AD2016
The 7th International Conference on
Algorithmic Differentiation

Programme and Abstracts

Monday 12th - Thursday 15th September 2016
Christ Church Oxford, UK

Conference Chairs:

Patrick Farrell (University of Oxford, UK)
Shaun A Forth (Cranfield University, UK)
Andreas Griewank (Yachay Tech, Ecuador)
Paul Hovland (Argonne National Laboratory, USA)
Jens-Dominik Müller (Queen Mary University of London, UK)

Invited Speakers:

Paul Barton (Massachusetts Institute of Technology, USA)
Jacques du Toit (Numerical Algorithms Group Ltd., UK)
Patrick Farrell (University of Oxford, UK)
Daniel Goldberg (University of Edinburgh, UK)
Siegfried Rump (TU Hamburg, Germany)
Andrea Walther (Universität Paderborn, Germany)
Copyright

Neither the Conference Chairs, nor their institutions, make any claim to the copyright of the material presented in this Programme. The copyright of each extended abstract remains with the authors (or their institutions/employers as appropriate), or with any organisation to which the authors assign copyright within their abstract.
Conference Chairs

Patrick Farrell  University of Oxford, UK
Shaun A Forth  Cranfield University, UK
Andreas Griewank  Yachay Tech, Ecuador
Paul Hovland  Argonne National Laboratory, USA
Jens-Dominik Müller  Queen Mary University of London, UK

Invited Speakers

Paul Barton  Massachusetts Institute of Technology
Jacques du Toit  Numerical Algorithms Group Ltd.
Patrick Farrell  University of Oxford, UK
Daniel Goldberg  University of Edinburgh
Siegfried Rump  TU Harburg
Andrea Walther  Universität Paderborn, Germany

Programme Committee

Christian Bischof  TU Darmstadt, Germany
Martin Bücker  Friedrich-Schiller-Universität Jena, Germany
Luca Capriotti  Credit Suisse, UK
Bruce Christianson  University of Hertfordshire, UK
Boris Diskin  National Institute of Aerospace, USA
Nicolas R Gauger  TU Kaiserslautern, Germany
Ralf Giering  FastOpt, Germany
Laurent Hascoët  INRIA, France
Patrick Heimbach  University of Texas at Austin, USA
Sri Hari Krishna Narayanan  Argonne National Laboratory, USA
Uwe Naumann  RWTH Aachen University, Germany
Boyana Norris  University of Oregon, USA
Valérie Pascual  INRIA, France
Eric T Phipps  Sandia National Laboratories, USA
Olivier Pironneau  Pierre and Marie Curie University, France
Detlef Stammer  University of Hamburg, Germany
Trond Steihaug  University of Bergen, Norway
Emmanuel Tadjouddine  University of Leicester, UK
Jean Utke  Allstate, USA
Andrea Walther  Universität Paderborn, Germany
Haibin Zhang  Beijing University of Technology, China
AD2016 - Programme and Abstracts

Foreword

AD2016 will be organised by, and held at, Christ Church, Oxford, UK under the direction of an international program committee. It is a follow up to the six previous conferences held in Breckenridge (USA) in 1991, in Santa Fe (USA) in 1996, in Nice (France) in 2000, in Chicago (USA) in 2004, Bonn (Germany) in 2008, and Fort Collins (USA) in 2012. The conference is targeted at users of algorithmic differentiation (AD) from all application areas (eg, optimization, inverse problems, uncertainty quantification), AD algorithm researchers and AD tool developers.

Advances since the last meeting in 2012 include: new work for the differentiation of piecewise-differentiable programs; embedding AD in domain-specific packages (e.g., AD Model Builder for parameter estimation); more extensive use of AD in applications outside of the mainstays of engineering and earth sciences especially computational finance. In addition to these recent advances conference topics include: the use of AD in optimization and inverse problems; its application to ODEs/DAEs and uncertainty quantification. Given the proximity of the venue to Londo’s global financial center, we intend to highlight the role of AD in computational finance.

This conference series is the only venue dedicated to the collaboration between researchers in the field and the users of AD. The four year interval between the conferences ensures a substantial amount of new results will be presented.

Oral Presentations

The abstract for each oral presentation has been reviewed by at least two members of the Programme Committee to ensure it is:

• Original - the work must not be previously published, nor must it have been submitted for publication elsewhere.

• Relevant - the work must contribute to the field of algorithmic differentiation in terms of the
  – theory of the differentiation of algorithms including: generalization to nonsmooth functions and infinite dimensional operators; interplay and relations with continuous adjoints, discretization schemes, interval inclusions and numerical linear algebra.
  – implementation of AD tools: source transformation; overloading; novel programming languages; novel architectures (eg, GPUs); high-level or application dependent algorithm specifications.
  – applications of AD techniques: novel applications; improvements in existing application areas; case studies showing limitations of present generation AD tools.

• Accessible - the International Conferences on Algorithmic Differentiation attracts a broad range of applied mathematicians, computer scientists, and computational scientists and engineers. The bulk of each presentation should be accessible to this diverse audience.

Poster Presentations

The abstract for each poster presentation has been reviewed by at least one member of the Programme Committee to ensure it is:

• Recent - the work should be novel or published/submitted elsewhere within the last 12 months.

• Relevant - the work must contribute to the field of algorithmic differentiation in terms of the
  – theory of the differentiation of algorithms including: generalization to nonsmooth functions and infinite dimensional operators; interplay and relations with continuous adjoints, discretization schemes, interval inclusions and numerical linear algebra.
– implementation of AD tools: source transformation; overloading; novel programming languages; novel architectures (eg, GPUs); high-level or application dependent algorithm specifications.

– applications of AD techniques: novel applications; improvements in existing application areas; case studies showing limitations of present generation AD tools.

• Accessible - the International Conferences on Algorithmic Differentiation attracts a broad range of applied mathematicians, computer scientists, and computational scientists and engineers. The bulk of each poster should be accessible to a significant number of this diverse audience.

Post-conference publication

Authors of accepted oral presentations and novel poster presentations are strongly encouraged to submit a full paper describing their work for post-conference publication to a special edition of *Optimization Methods and Software* to be edited by Andreas Griewank.
Acknowledgements

The Chairs of AD2016 wish to thank the following organisations for their financial support to AD2016:

- SIAM/NSF for travel grants to invited speaker Paul I. Barton, two early career researchers and two students
- SIAM United Kingdom and Republic of Ireland section for sponsoring a student prize
- Numerical Algorithms Group Ltd. for their support of invited speaker Jacques du Toit
- Taylor & Francis for post-conference publication of selected papers arising from AD2016 in their journal *Optimization Methods and Software*.

The Chairs also wish to express their thanks to:

- the members of the Programme Committee for their help in choosing the venue, selecting the Invited Speakers and refereeing the extended abstracts
- Joanna Malton, Conference and Events Administrator at Christ Church Oxford, and her team for their professional organisation of the conference.
Contents

Copyright ii
Conference Chairs iii
Invited Speakers iii
Programme Committee iii

Foreword v
Oral Presentations v
Poster Presentations v
Post-conference publication vi

Acknowledgements vii
List of Participants xv
Programme xvii

1 Monday 12th September 2016 xvii
2 Tuesday 13th September 2016 xviii
3 Wednesday 14th September 2016 xix
4 Thursday 15th September 2016 xx

Abstracts of Invited Speakers 1

Paul I. Barton, Kamil A. Khan, Jose A. Gomez, Kai Höffner, Peter G. Stechlinski and
Harry A. J. Watson
Computationally Relevant Generalized Derivatives: Theory, Evaluation and Applications
Presenter: Paul I. Barton 3

Jacques du Toit, Klaus Leppkes and Johannes Lotz
Applying AD in the Finance Industry, and an Operator Overloading Tool for “Handwritten” Adjoint
Presenter: Jacques du Toit 5

Patrick Farrell
Automatic parallel adjoints of finite element simulations with dolfin-adjoint
Presenter: Patrick Farrell 8

Daniel Goldberg, Sri Hari Krishna Narayanan, Laurent Hascœt and Jean Utke
Automatic differentiation of current ice sheet models
Presenter: Daniel Goldberg 9

Siegfried M. Rump
Reliable Computing and AD in Matlab/Octave
Presenter: Siegfried M. Rump 11

Andrea Walther and Sri Hari Krishna Narayanan
Covering resilience: A recent development for binomial checkpointing
Presenter: Andrea Walther 12

Abstracts of Oral Presentations 17
AD2016 - Programme and Abstracts

Siamak Akbarzadeh, Jan Hückelheim and Jens-Dominik Müller
  Consistent treatment of incompletely converged iterative linear solvers in reverse-
  mode AD
  Presenter: Siamak Akbarzadeh .......................................................... 19

Mladen Banovic, Salvatore Auriaemma, Orest Mykhaskiv, Andrea Walther, Herve Legrand
  and Jens-Dominik Müller
  Automatic Differentiation of a CAD System applied in an Adjoint CFD Method
  Presenter: Mladen Banovic ............................................................. 23

Atilim Güneş Baydin, Barak A. Pearlmutter and Jeffrey Mark Siskind
  DiffSharp: An AD Library for .NET Languages
  Presenter: Atilim Güneş Baydin .................................................... 27

Atilim Güneş Baydin, Barak A. Pearlmutter and Jeffrey Mark Siskind
  Tricks from Deep Learning
  Presenter: Barak A. Pearlmutter .................................................... 31

Bradley Bell and Kasper Kristensen
  The Newton Step Method for Algorithmic Differentiation with Implicit Functions
  Presenter: Bradley Bell ............................................................... 35

Martin Bücker and Johannes Willkomm
  First-order Derivatives of Associated Legendre Functions
  Presenter: Martin Bücker ............................................................. 38

H. Martin Bücker, Arno Rasch, Ralf Seidler and Andreas Wolf
  Transforming OpenMP Programs by Automatic Differentiation: Beyond Preserving
  the Parallelization Specified in the Original Program
  Presenter: Daniel Walther ............................................................ 42

Isabelle Charpentier and Jens Gustedt
  Arbogast - Higher order AD for special functions with Modular C
  Presenter: Jens Gustedt ............................................................... 46

Isabelle Charpentier and Bruno Cochelin
  A full higher-order AD continuation and bifurcation framework
  Presenter: Isabelle Charpentier .................................................... 50

Bruce Christianson
  Differentiating through Conjugate Gradient
  Presenter: Bruce Christianson ..................................................... 54

Sabrina Fiege, Andrea Walther and Andreas Griewank
  A Method for the Minimization of Piecewise Smooth Objective Functions
  Presenter: Sabrina Fiege .............................................................. 58

Shaun Forth
  A Matlab implementation of the Minpack-2 Test Problem Collection
  Presenter: Shaun Forth ............................................................... 62

Tosio Fukushima
  Algorithmic Computation of Finite Difference
  Presenter: Tosio Fukushima .......................................................... 66

Laurent Hascöet and Mathieu Morlighem
  Source-transformation Differentiation of a C++-like Ice Sheet model
  Presenter: Laurent Hascöet .......................................................... 70

Richard Hasenfelder, Andreas Griewank and Tom Streubel
  An Algorithmic Piecewise Linearization based Method for Nonsmooth Dynamical
  Systems
  Presenter: Richard Hasenfelder ..................................................... 74

Shahadat Hossain and Nasrin Hakim
  Pattern Graph for Sparse Hessian Matrix Determination
  Presenter: Shahadat Hossain ........................................................ 78
Paul Hovland
A Forward Mode Version of the CENA Method for Roundoff Error Estimation
Presenter: Paul Hovland ................................................................. 82

Alexander Hück and Christian Bischof
Operator Overloading Compatibility for AD - A Case Study of Scientific C++
Codes
Presenter: Alexander Hück ............................................................ 87

Jan Hückelheim, Paul Hovland, Michelle Strout and Jens-D Müller
Exact replication of OpenMP-parallelism in reverse-mode AD for loops with symmetric memory access
Presenter: Jan Hückelheim ............................................................. 91

Dominic Jones
Block Scope Differentiation
Presenter: Dominic Jones ............................................................... 95

Benjamin Jurgelucks and Leander Claes
Optimisation of triple-ring-electrodes on piezoceramic transducers using algorithmic differentiation
Presenter: Benjamin Jurgelucks .................................................... 99

Kamil Khan
Branch-locking AD techniques for composite nonsmooth functions and nonsmooth implicit functions
Presenter: Kamil Khan ................................................................. 103

Koichi Kubota
Simple implementation and examples for piecewise linearization with abs-normal form
Presenter: Koichi Kubota .............................................................. 107

Kshitij Kulshreshtha and Sri Hari Krishna Narayanan
Interfacing source transformation AD with operator overloading libraries
Presenter: Kshitij Kulshreshtha ...................................................... 111

Kshitij Kulshreshtha and Sri Hari Krishna Narayanan
Wrappers for ADOL-C in scripting languages using SWIG
Presenter: Kshitij Kulshreshtha ...................................................... 115

Lisa Kusch, Tim Albring, Andrea Walther and Nicolas R. Gauger
Semi-automatic transition from simulation to one-shot optimization with equality constraints
Presenter: Lisa Kusch ................................................................. 119

Viktor Mosenkis and Uwe Naumann
On Lower Bounds for the Optimal Jacobian Accumulation Problem on linearized DAGs
Presenter: Viktor Mosenkis ......................................................... 123

Mahesh Narayananurthi, Torsten Bosse, Sri Hari Krishna Narayanan and Paul Hovland
AD-Suite - A Test Suite for Algorithmic Differentiation
Presenter: Mahesh Narayananurthi ............................................... 127

Sri Hari Krishna Narayanan and Laurent Hascoët
Interfacing OpenAD and Tapenade
Presenter: Laurent Hascoët .......................................................... 131

Uwe Naumann
Adjoint Code Design Patterns
Presenter: Uwe Naumann .......................................................... 135

Richard Neidinger and Ben Altman
Comparing High-Order Multivariate AD Methods
Presenter: Richard Neidinger ...................................................... 139
Valérie Pascual and Laurent Hascœet
Mixed-language Automatic Differentiation
Presenter: Valérie Pascual ........................................... 142

Eric Phipps and H. Carter Edwards
Operator Overloading-based Automatic Differentiation of C++ Codes on Emerging Manycore Architectures
Presenter: Eric Phipps ............................................. 146

John Pryce and Nedialko Nedialkov
How AD Can Help Solve Differential-Algebraic Equations
Presenter: John Pryce .............................................. 150

Feng Qiang, Cosmin Petra, Miles Lubin, Joey Huchette and Mihai Anitescu
On efficient Hessian computation using the edge pushing algorithm in Julia
Presenter: Feng Qiang ............................................... 154

Jarrett Revels, Miles Lubin and Theodore Papamarkou
Forward-Mode Automatic Differentiation in Julia
Presenter: Jarrett Revels ........................................... 158

Mohammad Ali Rostami, Martin Bucker, Michael Lülfesmann and Andreas Wolf
Bidirectional Partitioning is no Better than Unidirectional Partitioning when Computing the Diagonal Entries of a Sparse Jacobian Matrix
Presenter: Mohammad Ali Rostami .............................. 162

Max Sagebaum, Tim Albring and Nicolas R. Gauger
Expression templates for primal value taping in the reverse mode of AD
Presenter: Max Sagebaum ......................................... 165

Marc Schwalbach, Tom Verstraete and Nicolas R. Gauger
Developments of a Discrete Adjoint Structural Solver for Shape and Composite Material Optimization
Presenter: Marc Schwalbach ...................................... 169

Jeffrey Mark Siskind and Barak A. Pearlmutter
Binomial Checkpointing for Arbitrary Programs with No User Annotation
Presenter: Jeffrey Mark Siskind .................................. 173

Jeffrey Mark Siskind and Barak A. Pearlmutter
Efficient Implementation of a Higher-Order Language with Built-In AD
Presenter: Jeffrey Mark Siskind .................................. 177

Filip Srajer, Zuzana Kukelova and Andrew Fitzgibbon
A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Machine Learning and Computer Vision
Presenter: Filip Srajer ............................................... 181

Tom Streubel, Andreas Griewank and Richard Hasenfelder
Newton-like Methods for Solving Piecewise Smooth Systems of Equations Based on Successive Piecewise Tangent or Secant Linearization
Presenter: Tom Streubel .......................................... 185

Markus Towara and Uwe Naumann
SIMPLE Adjoint Message Passing
Presenter: Markus Towara ........................................ 188

Mu Wang, Guang Lin and Alex Pothen
Enhancing sparsity of Hermite polynomial expansions with Automatic Differentiation
Presenter: Alex Pothen ............................................. 182

Mu Wang and Alex Pothen
Evaluating High Order Derivative Tensors in Reverse Mode of Automatic Differentiation
Presenter: Mu Wang ............................................... 196
Mu Wang, Assefaw Gebremedhin and Alex Pothen
   Performance Evaluation of Automatic Differentiation Algorithms for Hessian Computation
   Presenter: Mu Wang ........................................................................................................... 200

Abstracts of Poster Presentations 205
   Tim Albring, Christian Bischof, Nicolas R. Gauger, Alexander Graf, Alexander Hück,
   Benjamin Jurgelucks, Max Sagebaum and Jean Utke
   A Case Study of ADOL-C and CoDiPack applied to the Ice Sheet System Model
   Presenter: Alexander Hück ................................................................................................. 207
   Bernhard Bachmann, Willi Braun, Martin Flehmig, Rüdiger Franke, Leo Gall, Benjamin
   Jurgelucks, Kshitij Kulshreshtha, Kilian Link, Ulf Markwardt, Vitalij Ruge, Andrea
   Walther, and Niklas Worschech
   Parallel Algorithmic Differentiation in OpenModelica
   Presenter: Benjamin Jurgelucks ......................................................................................... 210
   Torsten Bosse, Sri Hari Krishna Narayanan and Laurent Hascoët
   Piecewise Linear AD via Source Transformation
   Presenter: Laurent Hascoët ................................................................................................. 212
   Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Michael Betancourt,
   Sebastian Weber and Rob Trangucci
   Algorithmic Differentiation in the Stan Math C++ Library
   Presenter: Bob Carpenter ................................................................................................. 215
   Jonathan Hüsér, Jens Deussen and Uwe Naumann
   KKT-based integration of differential-algebraic equations with optimality criteria
   Presenter: Jonathan Hüsér ............................................................................................... 217
   Robert Kelly, Barak A. Pearlmutter and Jeffrey Mark Siskind
   Evolving the Incremental Lambda Calculus Into a Model of Forward AD
   Presenter: Barak A. Pearlmutter ..................................................................................... 220
   Johannes Lotz, Klaus Leppkes, Viktor Mosenkis and Uwe Naumann
   Adjoint NAG Library
   Presenter: Viktor Mosenkis ............................................................................................. 223
   Johannes Lotz, Uwe Naumann and Sumit Mitra
   Mixed Integer Programming for Call Tree Traversal
   Presenter: Johannes Lotz ................................................................................................. 225
   Olivier Mullier, Alexandre Chapoutot and Julien Alexandre Dit Sandretto
   Validated Computation of the Local Truncation Error of Runge-Kutta Methods
   with Automatic Differentiation
   Presenter: Olivier Mullier ................................................................................................. 227
   Gilles Pagès, Olivier Pironneau and Guillaume Sal
   Vibrato and Automatic Differentiation for High Order Derivatives and Sensitivities
   of Financial Options
   Presenter: Guillaume Sal ................................................................................................. 230
   Gilberto Pérez, Eric Larour, Jean Utke, Anton Bovin and Mathieu Morlighem
   An Approach to Computing Discrete Adjoints for MPI-Parallelized Models Applied
   to the Ice Sheet System Model
   Presenter: Jean Utke ......................................................................................................... 234
   Manuel Radons
   Additions to an AD-Based Linearization Framework
   Presenter: Manuel Radons ............................................................................................... 236
   Arindam Sen, Markus Towara and Uwe Naumann
   Discrete adjoint OpenFoam for surface and volume sensitivities
   Presenter: Arindam Sen ................................................................................................. 239
Jean Utke, Sabrina Fiege, Andrea Walther and Kshitij Kulshreshtha
A Nonsmooth Optimization Case Study
Presenter: Jean Utke . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 241
List of Participants

Alphabetical by first (given) name.

Alex Pothen  Purdue University
Alexander Hück  TU Darmstadt
Ameli Gottstein  Universität Heidelberg
Andrea Walther  Universität Paderborn
Andreas Griewank  Yachay Tech
Arindam Sen  RWTH Aachen
Atılım Güneş Baydin  University of Oxford
Barak Pearlmutter  Maynooth University
Benjamin Jurgelucks  Universität Paderborn
Bradley Bell  University of Washington
Bruce Christianson  University of Hertfordshire
Christian Bischof  TU Darmstadt
Daniel Goldberg  University of Edinburgh
Daniel Walther  Friedrich-Schiller-Universität Jena
David Humphris  Numerical Algorithms Group Ltd.
Dominic Jones  CD-adapco
Eric Phipps  Sandia National Laboratory
Feng Qiang  University of Edinburgh
Gilberto Pérez  University of California Irvine
Guillaume Sall  Université Pierre et Marie Curie
Henri-Olivier Duche  Murex
Isabel Popple  Taylor & Francis
Isabelle Charpentier  University of Strasbourg
Jacob Barhen  Oak Ridge National Laboratory
Jacques du Toit  Numerical Algorithms Group Ltd.
Jan Holland  Springer
Jan Hückelheim  Queen Mary University of London
Jarrett Revels  Massachusetts Institute of Technology
Jean Utke  Allstate Insurance
Jeffrey Siskind  Purdue University
Jens Gustedt  INRIA
Jens-Dominik Müller  Queen Mary University of London
Johannes Lotz  RWTH Aachen
John Pryce  University of Cardiff
Jonathan Hüser  RWTH Aachen
Justin Robinson  Taylor & Francis
Kamil Khan  Argonne National Laboratory
Klaus Leppkes  RWTH Aachen
Koichi Kubota  Chuo University
Kshitij Kulshreshtha  Universität Paderborn
Laurent Hascœet  INRIA
Lisa Kusch  TU Kaiserslautern
**List of Participants (continued)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahesh Narayananurthi</td>
<td>Virginia Tech</td>
</tr>
<tr>
<td>Manuel Radons</td>
<td>Humboldt-Universität zu Berlin</td>
</tr>
<tr>
<td>Marc Schwalbach</td>
<td>von Karman Institute for Fluid Dynamics</td>
</tr>
<tr>
<td>Markus Towara</td>
<td>RWTH Aachen</td>
</tr>
<tr>
<td>Martin Bücker</td>
<td>Friedrich-Schiller-Universität Jena</td>
</tr>
<tr>
<td>Max Segebaum</td>
<td>TU Kaiserslautern</td>
</tr>
<tr>
<td>Mike Dewar</td>
<td>Numerical Algorithms Group Ltd.</td>
</tr>
<tr>
<td>Miles Lubin</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>Mladen Banovic</td>
<td>Universität Paderborn</td>
</tr>
<tr>
<td>Mohamed Ali Rostami</td>
<td>Friedrich-Schiller-Universität Jena</td>
</tr>
<tr>
<td>Mu Wang</td>
<td>Purdue University</td>
</tr>
<tr>
<td>Nicolas Gauger</td>
<td>TU Kaiserslautern</td>
</tr>
<tr>
<td>Olivier Mullier</td>
<td>ENSTA ParisTech</td>
</tr>
<tr>
<td>Patrick Farrell</td>
<td>University of Oxford</td>
</tr>
<tr>
<td>Paul Barton</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>Paul Hovland</td>
<td>Argonne National Laboratory</td>
</tr>
<tr>
<td>Pavanakumar Mohanamuraly</td>
<td>Queen Mary University of London</td>
</tr>
<tr>
<td>Philip Srajer</td>
<td>Czech Technical University of Prague</td>
</tr>
<tr>
<td>Ralf Giering</td>
<td>FastOpt</td>
</tr>
<tr>
<td>Richard Hasenfelder</td>
<td>Humboldt-Universität zu Berlin</td>
</tr>
<tr>
<td>Richard Neidinger</td>
<td>Davidson College</td>
</tr>
<tr>
<td>Sabrina Fiege</td>
<td>Universität Paderborn</td>
</tr>
<tr>
<td>Salvatore Auriemma</td>
<td>OPEN CASCADE</td>
</tr>
<tr>
<td>Shahadat Hossain</td>
<td>University of Lethbridge</td>
</tr>
<tr>
<td>Shaum Forth</td>
<td>Cranfield University</td>
</tr>
<tr>
<td>Siamak Akbarzadeh</td>
<td>Queen Mary University of London</td>
</tr>
<tr>
<td>Siegfried Runup</td>
<td>TU Hamburg</td>
</tr>
<tr>
<td>Sri Hari Krishna Narayanan</td>
<td>Argonne National Laboratory</td>
</tr>
<tr>
<td>Tom Streibel</td>
<td>Humboldt-Universität zu Berlin</td>
</tr>
<tr>
<td>Toshio Fukushima</td>
<td>National Astronomical Observatory of Japan</td>
</tr>
<tr>
<td>Trond Steihaug</td>
<td>Universitetet i Bergen</td>
</tr>
<tr>
<td>Uwe Naumann</td>
<td>RWTH Aachen</td>
</tr>
<tr>
<td>Valérie Pascual</td>
<td>INRIA</td>
</tr>
<tr>
<td>Viktor Mosenkis</td>
<td>Numerical Algorithms Group Ltd.</td>
</tr>
</tbody>
</table>
# Programme

## 1 Monday 12th September 2016

All events in Blue Boar Lecture Theatre, Christ Church, Oxford or adjacent refreshment area unless stated otherwise.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Registration and Opening</td>
</tr>
<tr>
<td>08:50</td>
<td><strong>Piecewise linear and non-smooth I</strong> - Chair Andrea Walther</td>
</tr>
<tr>
<td>09:10</td>
<td>Richard Hasenfelder - An Algorithmic Piecewise Linearization based Method for Nonsmooth Dynamical Systems</td>
</tr>
<tr>
<td>09:30</td>
<td>Tom Streubel - Newton-like Methods for Solving Piecewise Smooth Systems of Equations Based on Successive Piecewise Tangent or Secant Linearization</td>
</tr>
<tr>
<td>09:50</td>
<td>Break</td>
</tr>
<tr>
<td>10:20</td>
<td><strong>Invited Presentation</strong> - Chair Andreas Griewank</td>
</tr>
<tr>
<td>11:20</td>
<td><strong>Piecewise linear and non-smooth II</strong> - Chair Andreas Griewank</td>
</tr>
<tr>
<td>11:40</td>
<td>Koichi Kubota - Simple implementation and examples for piecewise linearization with abnormal form</td>
</tr>
<tr>
<td>12:00</td>
<td><strong>Poster Presentations I</strong> - Chair Shaun Forth</td>
</tr>
<tr>
<td>12:20</td>
<td>Lunch in the Great Hall</td>
</tr>
<tr>
<td>13:50</td>
<td><strong>Invited Presentation</strong> - Chair Paul Hovland</td>
</tr>
<tr>
<td>14:50</td>
<td>Markus Towara - SIMPLE Adjoint Message Passing</td>
</tr>
<tr>
<td>15:10</td>
<td>Daniel Walther - Transforming OpenMP Programs by Automatic Differentiation: Beyond Preserving the Parallelization Specified in the Original Program</td>
</tr>
<tr>
<td>15:30</td>
<td>Break</td>
</tr>
<tr>
<td>16:00</td>
<td>Jan Hückelheim - Exact replication of OpenMP-parallelism in reverse-mode AD for loops with symmetric memory access</td>
</tr>
<tr>
<td>16:20</td>
<td>Mahesh Narayananmurthi - AD-Suite - A Test Suite for Algorithmic Differentiation</td>
</tr>
<tr>
<td>16:40</td>
<td>Filip Srajer - A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Machine Learning and Computer Vision</td>
</tr>
<tr>
<td>17:00</td>
<td>Shaun Forth - A Matlab implementation of the Minpack-2 Test Problem Collection</td>
</tr>
<tr>
<td>17:20</td>
<td>Close</td>
</tr>
</tbody>
</table>
2 Tuesday 13th September 2016

All events in Blue Boar Lecture Theatre, Christ Church, Oxford or adjacent refreshment area unless stated otherwise.

<table>
<thead>
<tr>
<th>Time</th>
<th>Speaker</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Bradley Bell</td>
<td>The Newton Step Method for Algorithmic Differentiation with Implicit Functions</td>
</tr>
<tr>
<td>08:50</td>
<td>Toshio Fukushima</td>
<td>Algorithmic Computation of Finite Difference</td>
</tr>
<tr>
<td>09:10</td>
<td>Barak A Pearlmuter</td>
<td>Tricks from Deep Learning</td>
</tr>
<tr>
<td>09:30</td>
<td>Paul Hovland</td>
<td>A Forward Mode Version of the CENA Method for Roundoff Error Estimation</td>
</tr>
<tr>
<td>09:50</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>10:20</td>
<td>Daniel Goldberg</td>
<td>Invited Presentation - Chair Laurent Hascoët</td>
</tr>
<tr>
<td>11:20</td>
<td>Bruce Christianson</td>
<td>Iterative Systems - Chair Laurent Hascoët</td>
</tr>
<tr>
<td>11:40</td>
<td>Siamak Akbarzadeh</td>
<td>Consistent treatment of incompletely converged iterative linear solvers in reverse-mode AD</td>
</tr>
<tr>
<td>12:00</td>
<td>Jonathan Hüsner</td>
<td>Poster Presentations II - Chair Jens-Dominik Müller</td>
</tr>
<tr>
<td></td>
<td>Johannes Lotz</td>
<td>KKT-based integration of differential-algebraic equations with optimality criteria</td>
</tr>
<tr>
<td></td>
<td>Barak A. Pearlmuter</td>
<td>Evolving the Incremental Lambda Calculus Into a Model of Forward AD</td>
</tr>
<tr>
<td>12:20</td>
<td>Lunch in the Great Hall</td>
<td></td>
</tr>
<tr>
<td>13:50</td>
<td>Kshitij Kulshreshtha</td>
<td>AD for Other Languages and Interoperability I - Chair Jean Utke</td>
</tr>
<tr>
<td>14:10</td>
<td>Dominic Jones</td>
<td>Efficient Implementation of a Higher-Order Language with Built-In AD Siskind</td>
</tr>
<tr>
<td>14:30</td>
<td>Valérie Pascual</td>
<td>On efficient Hessian computation using the edge pushing algorithm in Julia</td>
</tr>
<tr>
<td>14:50</td>
<td>Laurent Hascoët</td>
<td>Interfacing OpenAD and Tapenade</td>
</tr>
<tr>
<td>15:10</td>
<td>Jarrett Revels</td>
<td>Forward-Mode Automatic Differentiation in Julia</td>
</tr>
<tr>
<td>15:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>16:00</td>
<td>Jeffrey Siskind</td>
<td>AD for Other Languages and Interoperability II - Chair Valérie Pascual</td>
</tr>
<tr>
<td>16:20</td>
<td>Feng Qiang</td>
<td>Efficient Implementation of a Higher-Order Language with Built-In AD Siskind</td>
</tr>
<tr>
<td>16:40</td>
<td>Atilim Güneş Bay</td>
<td>On efficient Hessian computation using the edge pushing algorithm in Julia</td>
</tr>
<tr>
<td>17:00</td>
<td>Kshitij Kulshreshtha</td>
<td>Kul-Wrappers for ADOL-C in scripting languages using SWIG</td>
</tr>
<tr>
<td>17:20</td>
<td>Close</td>
<td></td>
</tr>
</tbody>
</table>
3 Wednesday 14th September 2016

All events in Blue Boar Lecture Theatre, Christ Church, Oxford or adjacent refreshment area unless stated otherwise.

<table>
<thead>
<tr>
<th>Time</th>
<th>Speaker(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Alex Pothen</td>
<td>Enhancing sparsity of Hermite polynomial expansions with Automatic Differentiation</td>
</tr>
<tr>
<td>08:50</td>
<td>Uwe Naumann</td>
<td>Adjoint Code Design Patterns</td>
</tr>
<tr>
<td>09:10</td>
<td>Jeffrey Mark Siskind</td>
<td>Binomial Checkpointing for Arbitrary Programs with No User Annotation</td>
</tr>
<tr>
<td>09:30</td>
<td>Viktor Mosenkis</td>
<td>On Lower Bounds for the Optimal Jacobian Accumulation Problem on linearized DAGs</td>
</tr>
<tr>
<td>09:50</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>10:20</td>
<td>Andrea Walther</td>
<td>Covering resilience: A recent development for binomial checkpointing</td>
</tr>
<tr>
<td>11:20</td>
<td>Martin Bücker</td>
<td>First-order Derivatives of Associated Legendre Functions</td>
</tr>
<tr>
<td>11:40</td>
<td>Mohammad Ali Rostami</td>
<td>Bidirectional Partitioning is no Better than Unidirectional Partitioning when Computing the Diagonal Entries of a Sparse Jacobian Matrix</td>
</tr>
<tr>
<td>12:00</td>
<td></td>
<td>Poster Presentations III - Chair Jens-Dominik Müller</td>
</tr>
<tr>
<td>12:20</td>
<td>Alexander Hück</td>
<td>A Case Study of ADOL-C and CoDiPack applied to the Ice Sheet System Model</td>
</tr>
<tr>
<td>12:40</td>
<td>Guillaume Sal</td>
<td>Vibrato and Automatic Differentiation for High Order Derivatives and Sensitivities of Financial Options</td>
</tr>
<tr>
<td>12:50</td>
<td>Gilberto Pérez</td>
<td>An Approach to Computing Discrete Adjoints for MPI-Parallelized Models Applied to the Ice Sheet System Model</td>
</tr>
<tr>
<td>12:20</td>
<td></td>
<td>Lunch in the Great Hall</td>
</tr>
<tr>
<td>13:50</td>
<td>Jacques du Toit</td>
<td>Applying AD in the Finance Industry, and an Operator Overloading Tool for “Handwritten” Adjoints</td>
</tr>
<tr>
<td>14:50</td>
<td>John Pryce</td>
<td>How AD Can Help Solve Differential-Algebraic Equations</td>
</tr>
<tr>
<td>15:10</td>
<td>Alexander Hück</td>
<td>Operator Overloading Compatibility for AD - A Case Study of Scientific C++ Codes</td>
</tr>
<tr>
<td>15:30</td>
<td></td>
<td>Break + Wednesday Posters</td>
</tr>
<tr>
<td>16:20</td>
<td>Eric Phipps</td>
<td>Operator Overloading-based Automatic Differentiation of C++ Codes on Emerging Manycore Architectures</td>
</tr>
<tr>
<td>16:40</td>
<td>Max Sagebaum</td>
<td>Expression templates for primal value taping in the reverse mode of AD</td>
</tr>
<tr>
<td>17:00</td>
<td>Laurent Hascœt</td>
<td>Source-transformation Differentiation of a C++-like Ice Sheet model</td>
</tr>
<tr>
<td>17:20</td>
<td></td>
<td>Close</td>
</tr>
<tr>
<td>18:15</td>
<td></td>
<td>Drinks in the Masters Garden</td>
</tr>
<tr>
<td>19:00</td>
<td></td>
<td>Conference Dinner in the Great Hall</td>
</tr>
</tbody>
</table>
4 Thursday 15th September 2016

All events in Blue Boar Lecture Theatre, Christ Church, Oxford or adjacent refreshment area unless stated otherwise.

<table>
<thead>
<tr>
<th>Time</th>
<th>Speaker</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td>Mladen Banovic</td>
<td>Automatic Differentiation of a CAD System applied in an Adjoint CFD Method</td>
</tr>
<tr>
<td>08:50</td>
<td>Lisa Kusch</td>
<td>Semi-automatic transition from simulation to one-shot optimization with equality constraints</td>
</tr>
<tr>
<td>09:10</td>
<td>Marc Schwalbach</td>
<td>Developments of a Discrete Adjoint Structural Solver for Shape and Composite Material Optimization</td>
</tr>
<tr>
<td>09:30</td>
<td>Benjamin Jurgelucks</td>
<td>Optimisation of triple-ring-electrodes on piezoceramic transducers using algorithmic differentiation</td>
</tr>
<tr>
<td>09:50</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>10:20</td>
<td>Patrick Farrell</td>
<td>Automatic parallel adjoints of finite element simulations with dolfin-adjoint</td>
</tr>
<tr>
<td>11:20</td>
<td>Jens Gustedt</td>
<td>Arbogast - Higher order AD for special functions with Modular C</td>
</tr>
<tr>
<td>11:40</td>
<td>Mu Wang</td>
<td>Performance Evaluation of Automatic Differentiation Algorithms for Hessian Computation</td>
</tr>
<tr>
<td>12:00</td>
<td>Viktor Mosenkis</td>
<td>Adjoint NAG Library</td>
</tr>
<tr>
<td></td>
<td>Arindam Sen</td>
<td>Discrete adjoint OpenFoam for surface and volume sensitivities</td>
</tr>
<tr>
<td></td>
<td>Olivier Mullier</td>
<td>Validated Computation of the Local Truncation Error of Runge-Kutta Methods with Automatic Differentiation</td>
</tr>
<tr>
<td>12:20</td>
<td></td>
<td>Lunch in the Great Hall</td>
</tr>
<tr>
<td>13:50</td>
<td>Richard Neidinger</td>
<td>Comparing High-Order Multivariate AD Methods</td>
</tr>
<tr>
<td>14:10</td>
<td>Shahadat Hossain</td>
<td>Pattern Graph for Sparse Hessian Matrix Determination</td>
</tr>
<tr>
<td>14:30</td>
<td>Mu Wang</td>
<td>Evaluating High Order Derivative Tensors in Reverse Mode of Automatic Differentiation</td>
</tr>
<tr>
<td>14:50</td>
<td>Isabelle Charpentier</td>
<td>A full higher-order AD continuation and bifurcation framework</td>
</tr>
<tr>
<td>15:10</td>
<td></td>
<td>Closing Remarks</td>
</tr>
<tr>
<td>15:30</td>
<td></td>
<td>End of Conference</td>
</tr>
</tbody>
</table>
Abstracts of Invited Speakers

Abstracts are presented alphabetically by the surname of the invited, first author.
Computationally Relevant Generalized Derivatives: Theory, Evaluation and Applications

Paul I. Barton,* Kamil A. Khan†, Jose A. Gomez‡, Kai Höffner,§ Peter G. Stechlinski,* and Harry A. J. Watson∥

May 31, 2016

A variety of generalizations of the concept of the derivative to classes of continuous nondifferentiable functions have been proposed. Likewise, algorithms for nondifferentiable equation solving and optimization assume the ability to evaluate some element of a generalized derivative at points visited by the algorithm. However, not all generalized derivatives are equal in the sense that the particular generalized derivative element employed can have a large influence on the performance of algorithms. This leads to the notion of computationally relevant generalized derivatives. Until recently, it has not been possible to evaluate generalized derivative elements without an (often arduous) manual analysis of specific cases. Furthermore, in settings such as implicit functions and parametric ordinary differential equations, results enabling the evaluation of concrete, computationally relevant generalized derivatives have not been available. This talk will discuss a number of new theoretical results and algorithms that lead to automatic methods for the evaluation of computationally relevant generalized derivatives in several settings. We will also outline two important applications where these advances are having an enabling impact on simulation, sensitivity analysis and optimization.

We begin by introducing two of the most widely used generalized derivative notions: the B-subdifferential and the Clarke generalized Jacobian [1]. Both are set-valued mappings defined for a locally Lipschitz function on an open set. We then summarize some results on how the particular generalized derivative element employed influences the convergence rate of nondifferentiable equation solving methods, motivating the notion of computationally relevant generalized derivatives. However, the theoretical properties of these generalized derivatives create a number of obstacles to the development of automatic methods for evaluation of their elements. These obstacles are summarized.

Next we introduce Nesterov’s notion of the lexicographic derivative (L-derivative) [2], and the related notion of the lexicographic directional derivative (LD-derivative) [3]. The key property of the LD-derivative is that it satisfies a sharp chain rule in the form:

\[
[f \circ g]'(x; M) = f'(g(x); g'(x; M)),
\]

where \( g : \mathbb{R}^n \to \mathbb{R}^m \) and \( f : \mathbb{R}^m \to \mathbb{R}^p \) are lexicographically smooth (L-smooth), \( h'(z; N) \) is the LD-derivative of the function \( h \) at \( z \) in the directions \( N \), and \( M \) is a seed matrix defining the sequence of directions in which the LD-derivative is required. A key feature of this result that distinguishes it from previous results is that it gives a closed-form expression for a specific generalized derivative element of a composite function, for an arbitrary nonsingular or nonsquare seed matrix. This enables LD-derivatives to be propagated through arbitrary finite compositions of multiple functions, provided the LD-derivative can be computed for each individual function. This also leads immediately to a vector forward mode of AD for LD-derivative evaluation; essentially, if the vector forward mode of AD is interpreted as computing the LD-derivative in the directions of the seed matrix, and LD-derivative calculus rules for elementary nondifferentiable functions such as abs, min, max, \( \| \cdot \|_2 \), etc. are provided, LD-derivatives can be computed with minor modifications to existing forward mode AD tools. Moreover, at differentiable points the LD-derivative reduces to the classical directional derivative in the seed directions, so this a true generalization of the classical vector forward mode.

The class of L-smooth functions is closed under composition, and thus comprise a very general class of functions, including continuously differentiable functions, convex functions, nondifferentiable but piecewise differentiable functions (in the sense of Scholtes [4]), integrals of L-smooth functions, etc. [2, 3]. In addition, a number of new results describe closed-form expressions that motivate the development of numerical methods to compute LD-derivatives of many functions not normally expressible in closed form, for example:

- parametric ordinary differential equations (ODEs) with L-smooth right-hand sides have solutions that are L-smooth, and LD-derivatives may be computed solving an auxiliary ODE derived by “lexicographic differentiation” of the original ODEs [5]. This is the first result that yields an ODE in closed form that can compute a computationally relevant generalized derivative.

*Corresponding Author, MIT, pib@mit.edu
†Argonne National Laboratory, kkhan@anl.gov
‡MIT, jagomezr@mit.edu
§Genzyme, kai.hoeffner@sanofi.com
*MIT, pstechlin@mit.edu
∥MIT, hwatson@mit.edu
• implicit functions defined by the solutions of L-smooth equations are L-smooth, and an auxiliary nondifferentiable, nonlinear equation system in closed form can be solved to evaluate LD-derivatives.

• combined, these results enable the computation of LD-derivatives of the solutions of L-smooth differential-algebraic equations.

• the solutions of linear program and lexicographic linear programs (LLP) parametrized by their right-hand sides are L-smooth, and again auxiliary optimization problems can be derived in closed form that can be solved to evaluate LD-derivatives.

The reliable and accurate simulation and optimization of liquified natural gas (LNG) processes poses many challenges, primarily because of the many sources of nondifferentiability in the models. These include representation of the heat transfer limitations implied by the second law of thermodynamics in multi-stream heat exchangers [6], the modeling of thermodynamic phase changes in streams that occur as process variables vary, and the need to isolate physically correct roots of cubic equations of state for realistic physical property models. In process models, these sources of nondifferentiability can appear in complex compositions, and tailored solution algorithms can create nested equation solves that need to communicate generalized derivatives up the hierarchy of nested equations. We illustrate how the advent of practical methods for computing LD-derivatives is enabling the development of reliable and accurate simulation and optimization methods for LNG processes.

Dynamic flux balance (DFBA) analysis is an emerging modeling framework that enables the combination of genome-scale metabolic models of microorganisms with detailed steady-state or dynamic models of the extracellular environment [7]. DFBA models come in the form of dynamical systems with LLPs embedded [7]. For example, the dependence of the right-hand side of an ODE on the optimal values function of a LLP introduces nondifferentiability. Algorithms for boundary-value and optimization problems with DFBA models embedded require evaluation of suitable generalized derivatives, which is facilitated by the aforementioned advances.

References


1 AD in the Finance Industry

Over the past 8 years, the Numerical Algorithms Group (NAG) and RWTH Aachen have been involved in providing AD (and adjoint AD in particular) solutions to the finance industry through the STCE group’s dco/c++ tool. Although AD has been well-known in diverse fields for over 20 years, it was “rediscovered” in finance only in 2006 when Giles and Glasserman [1] showed how it could be used to calculate the so-called “Greeks” (mathematical derivatives) of non-trivial financial models very efficiently.

Since then, the finance industry has been slow to adopt AD. One might expect that AD would be embraced since it provides a dramatically more efficient way to address a core business need. But this has not been the case. Through a mixture of unfamiliarity, urban myths (you always run out of memory), misunderstanding, corporate culture, sometimes bewildering organisational structures, and massive highly “regulated” code bases, the adoption of AD has been slow. It is only now, thanks largely to the European Banking Regulator, that most financial players see clear benefits to AD.

In the first half of the talk we will discuss what it takes to apply AD in a conservative, heavily regulated but technically literate industry such as finance. It is a great leap to go from AD in theory to AD in practice and requires among others

- Effective communication and education
- Highly flexible, robust, “industrial strength” (this notion will be made precise) AD tools which are easy to use
- Professional product support and consulting
- AD-enabled maths library routines
- A certain degree of ingenuity when required to differentiate non-differentiable functions (which are very common in finance)

Now more than ever, finance needs people with strong technical backgrounds who are AD literate, which is good news for graduates with AD experience.

2 Operator Overloading Tool for “Handwritten” Adjoint

GPUs have started to gain traction in finance: there are certain very important models and numerical methods for which GPUs deliver compelling performance, and several banks have gone to production with them. In the second half of the talk we will explore one of the implications of this.

From an adjoint AD perspective, GPUs are a little awkward since there are upwards of 30,000 threads running on a card with only 6GB RAM (or 12GB if you’re lucky). Moreover, vectorisation is critical, which means that threads cannot be viewed as being independent. This makes tape-based adjoint AD difficult and unattractive for many problems. From a language point of view, GPUs also have some limitations: no static class members, no “shared memory” class members, a performance hit when using virtual functions, small stacks, and no binary compatibility between classes on the host and on the GPU. Then there are synchronisation issues: in adjoint methods data flow is reversed, which means that in at least one fundamental model (local volatility), one ends up with “stochastic” race conditions (threads access the same array but at random indices) which are not simple to handle.

Since no-one does GPU programming just for fun, all these problems need to be resolved efficiently (they are accelerators after all, they have to accelerate the code significantly to be worth the investment). Our test code for investigating these problems is a large local volatility model implemented on GPU [2]. By using dco/c++ on the host

---

*Corresponding Author, Numerical Algorithms Group Ltd, jacques@nag.co.uk
†STCE RWTH Aachen, leppkes@stce.rwth-aachen.de
‡STCE RWTH Aachen, lotz@stce.rwth-aachen.de
and a handwritten adjoint kernel on the GPU [3], we showed that the entire gradient of 450 entries could be obtained in half a second. The adjoint kernel itself was tricky to write and difficult to read, and would have a tough time being allowed into a bank’s quant library.

This prompted us to look for a tool based solution. C++11 introduced many new language features, among them the keyword auto. It turns out that it is possible to use these to produce a metaprogram which can instantiate an adjoint code at compile time for large classes of primal codes. Moreover, when done carefully, the performance of this metaprogram-produced adjoint code is often the same as that of the handwritten adjoint (this depends on how good the platform C++11 compiler is at optimising code – some are much better than others). In particular, when applied to the test code, it gives the same runtime on Linux for CPU and GPU as the handwritten adjoint (on Windows it is slower but still a lot faster than tape).

We implemented this idea in a new set of dco/c++ types called dco_ntr. The code written using this new tool looks very similar to the primal code and is therefore easy to read and understand. The table below shows runtime in milliseconds of the local volatility test code (figures in brackets are runtime relative to passive):

<table>
<thead>
<tr>
<th></th>
<th>clang 3.6</th>
<th>gcc 4.7</th>
<th>nvcc 7.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>passive</td>
<td>1,461</td>
<td>1,406</td>
<td>18</td>
</tr>
<tr>
<td>handwritten</td>
<td>2,997 (2x)</td>
<td>2,808 (2x)</td>
<td>89 (4.9x)</td>
</tr>
<tr>
<td>dco_ntr</td>
<td>3,031 (2x)</td>
<td>3,025 (2.2x)</td>
<td>83 (4.6x)</td>
</tr>
<tr>
<td>dco/c++ tape</td>
<td>13,579 (9.3x)</td>
<td>11,011 (7.2x)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>clang 3.8</th>
<th>VS2015</th>
<th>Intel2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>passive</td>
<td>1,172</td>
<td>1,510</td>
<td>1,421</td>
</tr>
<tr>
<td>handwritten</td>
<td>1,906 (1.6x)</td>
<td>1,992 (1.3x)</td>
<td>1,876 (1.3x)</td>
</tr>
<tr>
<td>dco_ntr</td>
<td>4,384 (3.7x)</td>
<td>10,241 (6.9x)</td>
<td>11,671 (6.8x)</td>
</tr>
<tr>
<td>dco/c++ tape</td>
<td>16,125 (16x)</td>
<td>24,025 (15.9x)</td>
<td>18,833 (16x)</td>
</tr>
</tbody>
</table>

The tool therefore has a few benefits:

- As with all operator overloading approaches, there is only one code from which primal and adjoint are instantiated. Banks are nervous of separate adjoint codes which have to be kept in sync with primal codes
- The runtime performance of the tool appears to be close to handwritten speed. In all cases it is quite a bit faster than dco/c++’s tape-based adjoint.
- The tool itself uses no heap memory
- Since the tool is written in C++11, it should be able to handle arbitrary C++11 codes
- A key design constraint was that the metaprogram should compile quickly and not use excessive amounts of memory: this was achieved

As with most AD tools, the user still has to know what they’re doing, and still has to checkpoint appropriately.

3 Conclusion

The tool is still in development and has only been tested on three codes, so there is much work still to be done. In particular, we need to understand why performance on Windows is not as good. One of the problems with industrial code development is that you are tied to whatever compiler your client is using, and all banks require code to run under Windows, often with “outdated” Microsoft compilers. There is also work to do in understanding how the various compilers “understand” the metaprogram constructs that are used – it turns out the profiling a metaprogram is challenging.

The use case for the tool is rather specialised: performance-critical sections of code, or code where available memory is a problem (i.e. the primal code already uses almost all available RAM). Accelerator code usually has both these features.

This work was funded by NAG Ltd.

References

Automated adjoint simulations with FEniCS and dolfin-adjoint

Patrick E. Farrell∗

June 2016

In this talk, we describe the dolfin-adjoint system for deriving the adjoints of models written in the FEniCS finite element system. Its key property is its high level of abstraction: rather than applying algorithmic differentiation principles at the level of individual instructions, we apply them at the level of individual PDE solves. We show that this approach enjoys significant advantages for adjoint efficiency and parallelisation.

We further describe how the adjoints computed can be automatically employed in a PDE-constrained optimisation framework. It is crucial here to respect the function space structure of the underlying problem; this has implications for the correct representation of the automatically computed derivatives. We demonstrate that a naive application of AD causes nonconvergence in a standard benchmark problem, but convergence is rapid if the correct Riesz representation is employed.

∗Mathematical Institute, University of Oxford, patrick.farrell@maths.ox.ac.uk
Automatic differentiation of current ice sheet models

Daniel Goldberg* and Sri Hari Krishna Narayanan† and Laurent Hascoet‡ and Jean Utke§

June 2016

Adjoint models are used extensively in the field of climate modelling. Many weather forecasting frameworks make use of adjoint models to optimise the fit of atmospheric models to time-varying observations [1]. In the field of ocean models, adjoints have long been used to assimilate dynamic data [e.g., [2]] and to generate state estimates of the global ocean, leading to deeper understanding of the ocean general circulation and biogeochemical cycles [3]. And while in some cases adjoint models can be derived analytically, the complexity of an ocean general circulation model is such that Algorithmic Differentiation (AD) tools become a necessity.

Adjoint models have been used for several decades in the modelling of ice sheets as well, but in a slightly different manner. A classic problem in glaciology is to determine the basal frictional pattern that gives rise to an observed ice velocity field, assuming a fixed ice geometry. The equations governing the dependence of velocity upon friction lend themselves to an analytical adjoint that can be easily solved. The most challenging and pressing problems in glaciology today, however, deal with evolving ice geometry and data that is time-dependent, and the added complexity of time evolution demands AD for adjoint model generation. Applications might include assessing the sensitivity of ice sheet evolution to patterns of submarine ice sheet melting in order to determine the influence of ocean circulation on potential sea level rise (Fig. 1). In my talk I will discuss this relatively new application of AD tools, including the various approaches taken (both source-to-source and operator overloading) and considerations to be made. Moreover I will discuss the difficulties and successes of data assimilation using AD tools, and the challenges remaining – including assimilation of marine ice data, and questions regarding predictability and propagation of uncertainty.

Acknowledgements

This work was made possible in part through a SAGES (Scottish Alliance for Geoscience, Environment and Society) travel grant for early career exchange, NERC grant NE/M003590/1, ARCHER Embedded CSE support grant eCSE03-09, and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. We are grateful for valuable input from D Ham and B Christianson. Additionally the authors are grateful for valuable input from B. Smith, J. Brown and P. Heimbach.

References


*Corresponding Author, University of Edinburgh, dan.goldberg@ed.ac.uk
†Argonne National Laboratory, snarayan@mcs.anl.gov
‡INRIA, Laurent.Hascoet@inria.fr
§Allstate, jutke@allstate.com
Figure 1: A result from an ice sheet adjoint model. This figure shows adjoint sensitivity of loss of grounded ice volume (the component of an ice sheet that, if lost to the sea, contributes to sea level rise) to submarine melting under the floating ice shelves adjacent to Smith Glacier, West Antarctica (location shown in inset). Filled contours give modeled ice velocity where ice is grounded; red-white shading gives adjoint melt rate sensitivity under ice shelves. The thick black contour denotes the boundary of the ice shelves. From [4].
Reliable Computing and AD in Matlab/Octave

Siegfried M. Rump*

June 2016

Reliable Computing means to develop algorithms with result verification using solely floating-point arithmetic, so that every computed result is mathematically correct including all possible sources of errors. Besides discretization and conversion errors, in particular rounding errors have to be taken care of.

Algorithms with result verification have been successfully developed for a number of numerical problems including (real or complex) systems of linear and nonlinear equations, eigenproblems or differential equations. Several open problems have been solved with substantial use of verification methods, among them the proof and inclusion of the Feigenbaum constant, the verification of chaos in the Lorenz equation, the famous Kepler conjecture or the proof of eigenvalues of a Sturm-Liouville problem.

Needless to say that the computation of derivatives is very helpful if not mandatory for several of the mentioned problems. In this talk we will discuss certain aspects of automatic differentiation in Matlab and Octave, including result verification, covering gradients, Hessians, and Taylor expansions. All this is part of INTLAB, the Matlab/Octave toolbox for Reliable Computing.

One of the major problems in Matlab and Octave is the interpretation overhead. For the development of fast algorithms it is mandatory to use vector and matrix notation wherever possible in order to avoid a severe time penalty. This is one reason to choose the forward mode, besides the easy implementation using Matlab/Octave’s operator concept.

As in compiled languages the memory management contributes to a fast implementation. This does not only cover storing certain quantities columnwise, but also to compute and store only the upper part of a Hessian and more. Some of those details will be discussed.

In order to compute mathematically correct inclusions of the range of a function, interval arithmetic is a convenient tool. Care is necessary to diminish overestimation due to dependencies. One possibility to that end is affine arithmetic, also included in INTLAB.

As a result, given a multivariate function $f$ and an $n$-dimensional interval vector $X$, a result interval vector $Y$ is computed such that $f(x)$ is an element of $Y$ for all real or complex $x$ in $X$. This does not only hold for the function itself, but also an inclusion of the range of the gradient, the Hessian or a Taylor expansion over $X$ is computed. This offers the possibility to compute mathematically fully rigorous results in pure floating-point arithmetic.

The INTLAB implementation has been tested by third parties, showing that our Matlab/Octave implementation for automatic differentiation seems the fastest available. Other tests for affine arithmetic show similar results. However, still the interpretation overhead remains. As a result, the computational speed becomes the more favorable the larger the vectors and matrices are.

INTLAB [1] has been invented by the author in 1998, and since then continuously developed and extended, currently in Version 9. It comprises of several tool boxes and verification methods. In [2] five of the ten problems of the SIAM 100-digit challenge were solved using INTLAB. Background on the state of the art of verification methods can be found in [3].

References


---

*Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, Hamburg 21071, Germany, and Visiting Professor at Waseda University, Faculty of Science and Engineering, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan, rump@tuhh.de
Covering resilience: A recent development for binomial checkpointing *

Andrea Walther¹ and Sri Hari Krishna Narayanan²

June 12, 2016

1 Introduction

Nowadays, adjoint methods form a well established approach to compute gradient information in a very efficient way in terms of runtime. However, as soon as the considered process involves any kind of nonlinearity, the memory requirement to compute the corresponding adjoints is in principle proportional to the operation count of the underlying function, see, e.g., [1] Sec. 4.6. For this reason, several very different checkpointing techniques have been developed over the last decades. For a summary of checkpointing approaches see [1] Chap. 12. All these checkpointing strategies have in common that they use a small number of memory units (checkpoints) to store the system state at some intermediate states during the evaluation of the program that computes the function value. Subsequently, the recomputation of information that is needed for the adjoint computation but currently not available is performed using these checkpoints in an appropriate way. Hence, all checkpointing techniques represent a compromise between memory requirement and runtime increase.

For this paper, we assume that the evaluation of the function of interest has a time-step structure given by

\[ x_i = F_i(x_{i-1}, u_{i-1}), \quad i = 1, \ldots, l, \]

for a given \( x_0 \), where \( x_i \in \mathbb{R}^n, i = 0, \ldots, l \), denote the state of the considered system and \( u_i \in \mathbb{R}^m \) the control. The operator \( F_i : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n \) defines the time step to compute the state \( x_i \). The process to compute \( x_l \) for a given \( x_0 \) is also called forward integration. Applying the reverse mode of algorithmic differentiation (AD), one obtains an adjoint integration of the form

\[ \bar{u}_i, \bar{x}_i \text{ given } (\bar{x}_{i-1}, \bar{u}_{i}) = \bar{F}_i(\bar{x}_i, \bar{u}_i, x_{i-1}, u_{i-1}), \quad i = l, \ldots, 1, \]

where the operator \( \bar{F}_i \) represents the \( i \)th adjoint time step. It is important to note that the information of the forward integration [1] is needed for the adjoint computation [2] in reverse order. To provide this information within a given limited amount of memory and a minimal number of recomputations, the so-called binomial checkpointing approach was proposed, see, e.g., [2, 3]. We will use this strategy as a basis to develop a checkpointing approach that can also handle a breakdown of the computing system. For example, there might be a foreseeable suspension, where the application has to stop for example because a given time limit is reached. This scenario occurs when running our target application, the MIT General Circulation Model (MITgcm, [1, 5]) on ARCHER, a UK-based supercomputer. The MITgcm code executes around 351,000 time steps to simulate one year of physical time requiring around 24 hours of wall clock computation time. Because the mean time between failure (MTBF) of ARCHER is lower than 24 hours, administrative policies require applications to execute only for a fixed time duration significantly less than 24 hours at a time. Then they are suspended and have to restart. In this case, the application can be restarted from suspension if checkpoints containing suitable intermediate data are available and the application is aware of its position in the overall computation. However, it might also happen that the application is killed externally because of machine failure, i.e., there might be also an unforeseeable breakdown.

For the targeted large-scale applications, the checkpoints comprise a significant amount of data. Furthermore, checkpoints stored in memory can be lost in the case of an unforeseeable breakdown. Therefore, it is reasonable to assume that checkpoints may have to be stored to disk. Then, the access time to read or write a checkpoint is no longer negligible in contrast to the assumption frequently made for the development of checkpointing approaches. There are a few contributions to extend the available checkpointing techniques to a hierarchical checkpointing, see, e.g., [4, 7, 8]. However, to derive a first checkpointing technique that incorporates resilience we ignore this hierarchical nature and assume throughout that the writing or reading process for a checkpoint is performed asynchronously such that it does not interfere with the adjoint computation.

This paper has the following structure. In Sect. [2] we describe the functionality of the software revolve [3] that we will use a starting point for a checkpointing approach and describe the extensions of the binary checkpointing

---

*This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

¹Universität Paderborn, Germany, andrea.walther@uni-paderborn.de

²Argonne National Laboratory, snarayan@mcs.anl.gov
approach and therefore also of revolve that are required for resilience. First results with respect to the resulting
temporal complexity will be given in Sect. 3. Finally, we draw conclusions and give an outlook of future work in
Sect. 4.

2 Binomial Checkpointing for Resilience

It was shown in [3] that a checkpointing scheme based on binomial coefficients yields for a given number of checkpoints
the optimal temporal complexity, i.e., the minimal number of time steps that have to be recomputed. To apply
the binomial checkpointing strategy, the software revolve can be used to steer the adjoint computation. Then the
corresponding calculations are performed within a while-loop where revolve determines the next action to be performed.
We have to stress that revolve provides a so-called serial checkpointing. That is, only one forward time step or one
adjoint step is performed at each stage of the adjoint computation. This is in contrast to so-called parallel checkpointing
techniques where several forward time steps might be performed in parallel even in conjunction with one adjoint step.

Up to now it was always assumed that the adjoint computation using revolve is free of any failures. However, even
if we use serial checkpointing the computation of the forward step or the adjoint step may be performed heavily in
parallel, i.e., may be evaluated on a large scale computer system. This is precisely the situation where we have to
take resilience into account and therefore an appropriately adapted extension of binomial checkpointing approach is
required.

The ability to recover from a possible breakdown yields two additional aspects that have to be covered by the
checkpointing strategy. First, the distance between two checkpoints should not be too large such that a restart of
the computation is not too costly. This results in an upper bound on the distance of two checkpoints. That is, the
number of time steps between two consecutive checkpoints should not exceed this upper bound. Second, because a
breakdown could also happen during the actual adjoint computation the adjoint state has to be checkpointed as
well. However, due to the nature of the adjoint computation, only one adjoint state is required to restart the adjoint
computation such that only one checkpoint is required for the adjoint information and its content is overwritten by
the next adjoint state. Because the target application is of large scale, we have to cover the situation where it is not
time possible to checkpoint every adjoint state computed. Therefore, also a certain distance, i.e., number of adjoint steps,
between these so-called adjoint checkpoints has to be taking into account. To distinguish between the two types of
checkpoints, we will call the checkpoints storing the information on the state x state checkpoints.

Because the adjoint checkpoints do not influence the checkpointing strategy, only the maximal distance between
two state checkpoints interferes with the binomial checkpointing strategy. To limit the number of time steps between
two consecutive state checkpoints, we first compute the distance required for the optimal, i.e., binomial checkpointing.
Then, this number is compared with the upper bound on the distance. If the distance for the optimal binomial
checkpointing is smaller than the upper bound this value is taken. Otherwise, only the number of time steps given by
the upper bound is performed despite the fact that the overall checkpointing strategy might be only suboptimal.

As a first observation, one has to note that the maximal distance between two consecutive state checkpoints
denoted by d can not be chosen completely independently from the number of available checkpoints denoted by c and
the number of time steps denoted by l, because the equality

\[ l \leq d \cdot c \]

must hold. Hence, this equation limits the distance d for resilience purposes from below by \([l/c]\). If l is close
to the upper bound \(d \cdot c\) defined by the values of c and d, many recomputations have to be performed because the
resulting checkpointing scheme becomes similar to the strategy of complete recomputation for large parts of the adjoint
computation.

3 First Results with Respect to Complexity

To illustrate the influence of the checkpointing also for resilience, we consider the small academic test case given by

\[
\min J(x, u) \quad \text{with} \\
\text{s.t. } x_1'(t) = 0.5x_1(t) + u(t), \quad x_1(0) = 1 \\
x_2'(t) = x_1(t)^2 + 0.5u(t)^2, \quad x_2(0) = 0 \\
t \in [0, 1].
\]

For this optimization problem, the adjoint can be derived analytically yielding

\[
\lambda_1'(t) = -0.5\lambda_1(t) - 2x_1(t)\lambda_2(t) \quad \lambda_1(1) = 0
\]

\[
\lambda_2'(t) = 0 \quad \lambda_2(1) = 1,
\]

such that it is possible to verify the correctness of the adjoint computation also for the extended version of revolve
covering resilience.
We tested and verified the binomial checkpointing with resilience for up to 100,000 time steps and breakdowns occurring at numerous different places. As representative observation, the left-hand side of Fig. 1 illustrates the additional recomputations needed by the optimal binomial checkpointing with resilience as solid line for 100,000 steps and a varying number of checkpoints. The additional recomputations needed by the binomial checkpointing approach that incorporates resilience are illustrated with dotted lines for the resilience distances of 5000, 1000, 500, and 250 time steps if no breakdown occurs. Hence, this picture illustrates the penalty that we face for taking resilience into account.

As can be seen, there is a very high number of additional time steps required for a \(c, d\)-combinations where the product of both values is close to \(l\). In these cases the temporal complexity of the resulting checkpointing strategy deviate a lot from the optimal one due to the reason explained at the end of the last section. On the other hand, for this example it can be also seen that the binomial checkpointing with resilience does not interfere with the optimality of the binomial checkpointing if the product \(d \cdot c\) is reasonably larger than \(l\).

This observation might be a little surprising, especially because the checkpoint distance for resilience is in numerous cases smaller than the checkpoint distance chosen by the current version of \textit{revolve} approach as illustrated on the right-hand side of Fig. 1 for the four first checkpoints to compute the adjoint of 100,000 steps and a varying number of checkpoints.

To explain the surprisingly good temporal complexity of the checkpointing for resilience, the strategy to place the next state checkpoint \(l\) has to be examined. As shown in [3], for almost all cases there are numerous possibilities to set the next state checkpoint as illustrated by the large diamond in Fig. 2. Here, the number of time steps \(l\) varies such that

\[
\beta(c, r - 1) < l \leq \beta(c, r) \equiv \left(\frac{c + r}{c}\right)
\]

holds for the given number of checkpoints \(c\) and an integer value \(r\). The approach chosen by \textit{revolve} is such that the number of times a state checkpoint is written is minimized, see [3, Prop. 2]. The corresponding possible choices to place the next state checkpoint are illustrated by the two small diamonds in dark gray in Fig. 2. The distance between two consecutive state checkpoints has absolutely no influence on this choice. To minimize this distance, as alternative strategy one can chose the next state checkpoint according to the solid black line in Fig. 2.

This alternative strategy will be analyzed in more detail in the full paper. That is, first theoretical results with respect to the optimality of the proposed checkpointing for resilience will be shown for a large range of \(d - c\) combinations. In addition to these theoretical considerations, we will present and analyze a performance study for the targeted large-scale application, i.e., the MITgcm code.

## 4 Conclusions and Future Work

We have presented a modified binomial checkpointing algorithm for resilience such that a restart of the adjoint computation after a breakdown of the computing system becomes possible. The modified algorithm maintains the optimality of binomial checkpointing while limiting the maximum distance between successive forward and adjoint checkpoints for numerous situations. The required changes were integrated in the software package \textit{revolve} for binomial checkpointing for a first complexity study. The performance of this new checkpointing strategy will be analyzed for the MITgcm code, i.e., a large application running on a large-scale computing machine.

Future work has to examine the question where checkpoints are stored and how this might influence the checkpointing strategy if the time to write and read a state or adjoint checkpoint is no longer ignored.
Figure 2: Possible places for the next state checkpoint $\hat{l}$

References


Abstracts of Oral Presentations

Abstracts are presented alphabetically by the surname of the first author. If an author is first author on multiple abstracts then alphabetical order of the first significant word of the title is used to order such papers.
Consistent treatment of incompletely converged iterative linear solvers in reverse-mode AD

Siamak Akbarzadeh∗ Jan Hückelheim† Jens-Dominik Müller ‡

March 2016

1 Introduction

In algorithmic differentiation [1] it is a common practice not to differentiate the linear solver, due to accuracy or because it is used from an external library, and replace it by a new set of linear equations [2]. However, the treatment is limited on the assumption that the solver is converged to machine accuracy. In many practical applications it is not required or feasible to fully converge the system. In this work we show that this particularly leads to inaccurate adjoint gradient calculation and present a correction to remove this inaccuracy in this type of treatment. The method is proven to be correct for relaxation-type linear solvers such as SSOR or Jacobi, but numerical experiments show that it is also beneficial for other linear solvers such as CG or GMRES.

2 Conventional Linear Solver Treatment

Let us consider a nonlinear numerical model, \( y \leftarrow F(x) \), in which \( x \in \mathbb{R}^n \) and \( y \in \mathbb{R}^m \) are the input and output respectively. Inside this nonlinear system a linear system of equations should be solved, \( A \cdot x = b \), that can be regarded as an inverse matrix product \( x = A^{-1} \cdot b \). In practice what it should be implemented as a treatment for linear solver subroutine in a code is depicted in Fig.1.

![Figure 1: Forward (left) & reverse (middle) conventional linear solver treatment, proposed correction treatment for incomplete converged solvers (right)](image)

The value of \( \bar{x} \) after the adjoint linear solver is set to zero. There is an intuitive explanation for this: \( \bar{x} \) after the adjoint linear solver contains the sensitivity of the cost function with respect to the value in \( x \) before the primal linear solver. This is nothing else but the sensitivity of the cost function to the initial guess given to the linear solver. If the solver was fully converged, that sensitivity must be zero. But in practice, the iterative linear solvers are used for large systems of linear equations that arise for example in Finite Volume or Finite Element schemes and the residual of adjoint/tangent linear systems are preferred, as primal, to be dropped by a few orders of magnitude (eg. \( x^* \) and \( incr^* \)). In the next section we show how this effects the accuracy of gradients.

3 Consistent Adjoint Linear Solver Treatment

If an iterative solver is used and not fully converged, it cannot be assumed that the initial guess has no influence on the final solution, and thus it is incorrect to set \( x = 0 \) at the end of the adjoint linear solver. For the example of relaxation-type solvers such as a Jacobi iterative solver, we present in this section a reverse-differentiation that is consistent with the forward-differentiation and correctly takes into account the influence of the initial guess of the primal system on the adjoint derivatives.

---

*Corresponding Author, Queen Mary, University of London, s.akbarzadeh@qmul.ac.uk
†Queen Mary, University of London, j.c.hueckelheim@qmul.ac.uk
‡Queen Mary, University of London, j.mueller@qmul.ac.uk
The Jacobi iterations can be considered as an iterative solver scheme. Given a matrix $A$, we define matrices $D$ and $R$ such that $D$ will hold all diagonal entries of $A$, and $R$ will hold all off-diagonals of $A$. The Jacobi iterative solver is guaranteed to converge if $A$ is irreducibly diagonally dominant. The iterative relaxation scheme can be written as

$$x^{k+1} = D^{-1} \cdot (b - R \cdot x^k)\quad (1)$$

Assuming that the exact solution to the system is $x$ we obtain a relation between the iteration error defined by

$$x^{k+1} - x = \delta x^{k+1} = -(D^{-1} \cdot A - I) \cdot \delta x^k\quad (2)$$

Therefore after $N$ iterations starting from an initial guess $x_0$, the solution can be written as

$$x^N = (D^{-1} \cdot A - I)^N \cdot (x - x^0) + x\quad (3)$$

Ideally when after sufficient number of iterations the solver is fully converged the final iteration error, $\delta x^N$ is zero and $x^N = x$. The forward differentiation of Jacobi iterations, Eq.1, is in the following form

$$\dot{x}^{k+1} = D^{-1} \cdot (\dot{b} - \dot{R} \cdot x^k - R \cdot \dot{x}^k) - D^{-1} \cdot \dot{D} \cdot x^{k+1}\quad (4)$$

and following the same analysis the tangent linear error can be identified as

$$\dot{x}^{k+1} - \dot{x} = \delta x^{k+1} = D^{-1} \cdot (\dot{b} - \dot{R} \cdot x^k - \dot{R} \cdot x^k) - D^{-1} \cdot \dot{R} \cdot \delta x^k - D^{-1} \cdot \dot{D} \cdot \delta x^{k+1}\quad (5)$$

Clearly the error of derivative calculation in tangent mode with Jacobi iteration include two types of errors. Firstly the error of primal calculation, $P_{Error}$, and secondly the error of Jacobi iteration for gradient computation, $(D^{-1} \cdot A - I) \cdot \delta \dot{x}$. This means even if the exact solution to the primal linear system was available the Jacobi iteration error would remain. In linear solver treatment, the primal is already solved before solving the tangent linear system. The last iteration error of primal, $\delta x^N$ is fixed in the tangent linear sytem and remains the same for all Jacobi iterations. If both primal and tangent linear systems were converged to machine precision there would be almost no error in the gradient calculation and after sufficient $N$ iterations

$$\delta x^{k+1} = -(D^{-1} A - I) \cdot \delta x^k - D^{-1} \cdot \dot{A} \cdot \delta x^N\quad (6)$$

$$\dot{x}^N = ((D^{-1} A - I)^N + I) \cdot \dot{x} - (D^{-1} A - I)^N \cdot \dot{x}^0\quad (7)$$

Note that as the expression $(D^{-1} A - I)^N$ vanishes, this means that the influence of the initial guess vanishes and the derivative of the final solution with respect to the correct solution becomes an identity matrix. Knowing the relation between $\dot{x}^N$ and $\dot{x}^0$ from Eq.7, the reverse differentiation relation can also be identified as

$$\dot{x}^0 = -(D^{-1} A - I)^{N,T} \cdot \dot{x}^N = (I - A^T \cdot D^{-T})^{N,T} \cdot \dot{x}^N\quad (8)$$

On the other hand, in reverse linear solver treatment to solve $A^T \cdot \dot{b} = \dot{x}$ we iterate,

$$\dot{b}^{k+1} = D^{-T} \cdot (\dot{x} - R^T \cdot \dot{b}^k)\quad (9)$$

and as it was shown before, Eq.3, after $N$ Jacobi iterations, starting with initial guess $\dot{b}^0 = 0$, yields a result

$$\dot{b}^N = (I - (A^T \cdot D^{-T} - I)^{N,T}) \cdot \dot{b}\quad (10)$$

and the residual of the linear solver after $N$ iterations can be calculated as

$$Res = \dot{x} - A^T \cdot \dot{b}^N$$

$$= \dot{x} - A^T \cdot \dot{b}^0 - A^T \cdot (I - (A^T \cdot D^{-T})^{N,T} \cdot \dot{b})$$

$$= (I - A^T \cdot D^{-T})^{N,T} \cdot \dot{x}\quad (11)$$
This means that if we use a relaxation method (linear) and the same number of iterations in forward and reverse, then the computed residual after the last adjoint iteration is the correct sensitivity of the cost function with respect to the primal initial guess.

\[ \dot{x}^0 = Res = x - A^T \cdot \bar{b}^N \]

4 Nonlinear Loop

Let us again consider the nonlinear function, \( y \leftarrow F(x) \). A nonlinear computational algorithm like Picard Iteration (indicated by superscript \( m \)) and its derivative is shown in Fig.2.

\[
\begin{align*}
A(x^m) &\leftarrow \text{funcA} (x^m) \\
b(x^m) &\leftarrow \text{funcB} (x^m) \\
x^{m+1} &\leftarrow \text{solve}(A, x^m, b) \\
y &\leftarrow \text{funcF}(x^{m+1})
\end{align*}
\]

\[
\begin{align*}
\dot{A}(x^m, \dot{x}^m) &\leftarrow \text{funcA}_d (x^m, \dot{x}^m) \\
\dot{b}(x^m, \dot{x}^m) &\leftarrow \text{funcB}_d (x^m, \dot{x}^m) \\
\dot{x}^{m+1} &\leftarrow \left\{ \begin{array}{l}
    rhs = b - \dot{A}x^{m+1} \\
    A_{m+1} = \dot{A} \\
    \end{array} \right.
\end{align*}
\]

\[
\begin{align*}
x^{m+1} &\leftarrow \text{funcF}_d (x^{m+1}, \dot{x}^{m+1}) \\
y &\leftarrow \text{funcF}_d (x^{m+1}, \dot{x}^{m+1})
\end{align*}
\]

\[
\begin{align*}
A_{m+1} b, \dot{x}^m &\leftarrow \left\{ \begin{array}{l}
    A^T \cdot \text{incr} = x^{m+1} \\
    \bar{b}^* \leftarrow \text{incr} \\
    \bar{A} = \text{incr} \cdot x^{m+1} \\
    \bar{x} = x^{m+1} - A^T \cdot \text{incr}
    \end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\dot{x}^{m+1} &\leftarrow \text{funcF}_d (x^{m+1}, \dot{y} \leftarrow 1) \\
x^{m+1} &\leftarrow \text{funcF}_d (x^{m+1}, \dot{x}^{m+1})
\end{align*}
\]

Figure 2: The nonlinear loop algorithm, primal(left), tangent linear(middle) and adjoint(right)

Although, in previous section we showed that in both primal and forward mode of AD, there would an error if the linear system was converged incompletely (the incomplete solutions are represented by the notation with star superscript like x* and x* in Fig.2 ) but if those error go to zero after sufficient number of iterations the tangent linear would be consistent with finite difference. However, in adjoint mode this is not the case if the conventional linear solver treatment is applied. If the system is not fully converged and the expression in the box is set to zero then the consistency is violated and wrong gradient has been reported by other researchers [4].

5 Numerical Examples

5.1 1D Nonlinear Heat Equation

The first validation study is the finite-difference solution to a nonlinear 1D steady state heat conduction problem.

\[
\frac{\partial}{\partial x} (k \frac{\partial T}{\partial x}) = 0
\]

where the conduction coefficient is a simple linear function of temperature, \( k = C_1 + C_2 T \). The model domain is 1cm length, Fig.3, and is discretised by 10 nodes, \( n=10 \), with \( \delta x_i = \frac{1}{n+1} \). The boundary node temperatures , \( T_{L, R} \) is defined as design variable and cost function is evaluated as the temperature value in the middle of domain, \( J = T \) \((n=5)\). The TAPENADE source transformation AD tool [3] is used to differentiate the code. The different methods sensitivities are validated against second order finite difference method and the results are tabulated in Tab.1.

\[
\begin{align*}
T_L &\rightarrow T_R \\
i=1 &\rightarrow i=2 \rightarrow i=3 \rightarrow \cdots \rightarrow i=10
\end{align*}
\]

Figure 3: 1D nonlinear steady state heat transfer

5.2 3D S-Bend Duct

The second validation study is the steady state laminar air duct flow, \( Re_{inlet} = 300 \), provided by Volkswagen AG, Fig.4. The design variables are defined as surface node coordinates in the S-Bend part of duct and the cost function is total pressure loss between inlet and outlet. The in-house SIMPLE [5] based incompressible discrete adjoint solver,
gpde [4] is used for primal and sensitivity calculation. The solver uses bi-conjugate gradients (Bi-CGSTAB) and conjugate gradient (CG) methods to solve the linear systems in momentum and pressure equations. The comparison of sensitivity in one of the surface nodes is tabulated in Tab. 2 showing the improvement of results even in Krylov-type solvers.

6 Conclusion and Outlook

In this paper we considered the linear solver treatment of iterative solvers and the inconsistency of the treatment in ADJ mode of AD when the solver is not fully converged. We introduced a correction, based on the residual of adjoint of linear system that removes the inaccuracy of gradient calculation. We proved the correction of method for relaxation-type solvers like Jacobi iterations. Moreover, we showed with numerical examples that the method is also beneficial in Krylov-type solvers. However the proof of suggested correction for general iterative solvers might be the subject of interesting future research.

Acknowledgements

This work is conducted within the About Flow project (http://aboutflow.sens.qmul.ac.uk) which has received funding from the European Unions Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 317006.

References

Automatic Differentiation of a CAD System applied in an Adjoint CFD Method

Mladen Banovic\textsuperscript{1∗}, Salvatore Auriemma\textsuperscript{2†}, Orest Mykhaskiv\textsuperscript{3‡}, Andrea Walther\textsuperscript{1}, Herve Legrand\textsuperscript{2} and Jens D. Mueller\textsuperscript{3}

March 2016

1 Introduction

Shape optimization of a three-dimensional CAD-based model using the adjoint method requires the calculation of the geometrical sensitivities with respect to the design parameters of the model. These sensitivities are usually not provided within a CAD system. One way of obtaining this information can be done by applying Finite Difference method to the CAD system which leads to numerical errors. Another possibility is to apply Automatic Differentiation if the CAD sources are available [1].

This paper explains the automatic differentiation of the open-source CAD kernel OpenCASCADE Technology (OCCT) applying the forward traceless mode of the AD software tool ADOL-C (Automatic Differentiation by OverLoading in C++). For the first time, a fully developed CAD system is differentiated. The differentiated OCCT kernel is validated in a derivative-based optimization of the pressure losses in a squared U-bend pipe.

2 Automatic Differentiation of OCCT

The automatic differentiation of the OCCT code has been considered by Dannenhoffer and Haimes [2], but not yet implemented due to a large complexity of the sources. This paper demonstrates that the code differentiation is feasible by integrating ADOL-C into OCCT.

ADOL-C is a software tool that facilitates the computation of first and higher derivatives of vector functions that are defined by computer programs written in C or C++. As noted in the acronym, this software tool is based on the operator overloading concept (instead of source transformation) and therefore it is not generating intermediate source codes [3]. ADOL-C features the following modes:

- Forward (options: trace-based or traceless),
- Reverse (trace-based).

Applying the trace-based variants, operator overloading is used to generate an internal representation of the function to be differentiated. The difference between traceless and trace-based mode is that in traceless mode the derivative computation is propagated directly on the function evaluation. For computing and obtaining the derivatives in the trace-based mode, one has to call driver functions provided by ADOL-C. The traceless forward differentiation computes first order derivatives (higher order derivatives are only available by using the trace-based mode) in scalar mode and in vector mode, and it is very suitable for the purpose of this article.

Integration of ADOL-C library into a certain code is done by injection of its specific adouble type instead of native real type. Replacing the declaration types of all relevant real variables with the specific ADOL-C type is not simple when facing complicated object-oriented code like OCCT. The goal is to find a possible way (or ways) of adouble injection.

Therefore, several approaches of source code modification were considered during ADOL-C integration as it will be explained in the paper. Only one was taken as a way to proceed with the full sources - the typedef approach. The idea is to replace all doubles by adoubles, by using an existing typedef which is named Standard_Real in OCCT. The main advantage of this approach is that code modification should be as minimal as possible, while the drawback is about sacrificing memory and efficiency to some extend because all double variables, even ones not needed for differentiation, will be adouble objects.

Although the idea looks simple, it is not as straightforward as one would expect. There was a lot of code modification introduced (which is contradictory to the idea) and even after successful compilation, there were a lot of run-time errors faced during the testing phase related to the parts of OCCT used for this application. These issues will be...
discussed in the paper. The major part of differentiated OCCT kernel is working, but the debugging phase is still in progress. For this reason, the list of working functionalities will be summarized in the full paper.

After successful integration, AD has been verified against Finite Differences (by a central difference scheme) in the required OCCT methods for the U-bend generation. Furthermore, the difference between the differentiated and the original OCCT sources in terms of memory consumption and computational time is compared.

3 U-bend parameterization

The U-bend under investigation is a typical internal cooling channel. The baseline geometry consists of a circular U-part with attached legs, i.e. inlet and outlet, that are not manipulated during the optimization. The baseline geometry is shown in Figure 1.

The parameterization is done on the U-part; it is based on a cross-sectional design approach - the lofting. This approach constructs the final B-Spline surfaces by taking as inputs the \( n \) curves/slices generated along the guiding pathline. Each slice lies on a plane which is orthogonal to the pathline. The pathline is described as a B-Spline curve. The slice consists of the 4 Bezier curves forming a closed wire and having in total 12 control points. Each control point has its own law of evolution along the pathline that determines its position in the specific plane. Therefore, parameters of the laws of evolution are actual design parameters considered in the optimization.

![Figure 1: U-bend geometry](image)

4 Verification of Differentiated OCCT

4.1 Comparing AD with Finite Differences

As a representative example, the sensitivities with respect to one design parameter calculated by AD and Finite Differences are shown in Figure 2. By looking at the overall magnitude plots, one can state that these results coincide. Furthermore, the quantitative comparison between AD and Finite Differences will be discussed in the paper.

![Figure 2: U-part sensitivities evaluated by AD (left) and Finite Differences (right)](image)
4.2 Performance tests

An application used for constructing the U-bend only has been considered for comparing the performance between the original and differentiated OCCT sources. The average of the total computational time based on 10 measurements required for the U-bend construction is shown in Table 1. In the case of the differentiated OCCT sources, the derivative computation is done in one direction (scalar mode) as well as in 96 directions (vector mode), which is the total number of design parameters. By the theory [4], the run-time ratio between the derivative computation and function evaluation should be in range \([1 + p, 1 + 1.5p]\), where \(p\) is the number of directions. Comparing this with the results in Table 1, one can state the following: for the scalar mode (1 direction), the run-time ratio is approximately equal to 12, which is not good by the theoretical expectations; whereas in the case of 96 directions, the run-time ratio is approximately equal to 56, which is, surprisingly, way below the theoretical lower range boundary \((1 + p)\). Probably the scalar mode run-time ratio of 12 is so high because of the way how overloaded operators are written: for each operator, there is a for loop that iterates \(p\) times in order to compute the derivative in each direction. This could be the reason why the scalar mode is not so effective as the vector mode. Possible improvement here would be to modify the overloaded operators in order to remove the for loops if the scalar mode is used.

Furthermore, the same application has been profiled in all of three cases with the profiling tool Massif (which is a part of the Valgrind tools for debugging and profiling). The maximum heap memory size and the total number of CPU instructions are shown in Table 2. It is important to note here that this is not the total memory required by the application, Massif only investigates the heap memory, which is the memory allocated/deallocated dynamically during the run-time. The charts, as well as the total memory requirements, will be discussed in the paper.

![Table 1: U-bend construction time difference between original and AD sources](image)

![Table 2: U-bend application profiling summary](image)

5 Application of CAD-sensitivities in aerodynamic shape optimization

Taking to account that the main subject of this paper is the automatic differentiation of the CAD-system, only a brief overview of the gradient-based shape optimization loop will be presented here, more details can be found in [1] and [5]. The aerodynamic performance of a given CAD-geometry usually could be described with a scalar cost function \(J\) (drag, lift, total pressure loss, etc.). In order to optimize a given shape, we consider an optimization problem of the CAD parameters \(\alpha\):

\[
\min_\alpha \ J(U(X(\alpha)), X(\alpha), \alpha)
\]

(1)

\[
R(U(X(\alpha)), X(\alpha)) = 0.
\]

(2)

Equation (2) describes the flow field within the domain of interest (includes or completely defined by CAD-model). Usually (2) is written in terms of system of Reynolds-Averaged Navier-Stokes equations, with the state variable \(U\) and computational mesh coordinates \(X\), which depend on design parameters \(\alpha\). In case of large amount of design parameters (usually the case in industrial applications) the adjoint method proves to be computationally efficient and will be followed here. Application of a chain rule to the system (1)-(2) yields:

\[
\frac{dJ}{d\alpha} = \left[ \frac{dJ}{dX} + \nu^T f \right] \frac{\partial X}{\partial \alpha},
\]

(3)

where

\[
f = -\frac{\partial R}{\partial X}.
\]

(4)

Here \(\nu\) represents the solution of adjoint equations:

\[
\left( \frac{\partial R}{\partial U} \right)^T \nu = \frac{\partial J}{\partial U}.
\]

(5)

After computing the solution of primal and adjoint equations one can map the obtained volume sensitivity onto design surfaces. For instance, using a spring-based analogy as the volume-surface deformation algorithm allows to write the sensitivity in terms of surface nodes perturbations:

\[
\frac{dJ}{d\alpha} = \frac{dJ}{dX_S} \frac{dX_S}{d\alpha}.
\]

(6)
The first term in (6) corresponds to flow sensitivity in the surface grid point $X_S$, while the second represents the geometrical (CAD) derivative w.r.t. design parameters. These terms could be computed independently from each other, during flow solver run and CAD subroutines available in the differentiated CAD-kernel. After composing flow and CAD derivatives obtained sensitivity is used in iterative gradient-based optimization loops:

$$\alpha^{(n+1)} = A(\alpha^{(n)}, \frac{dJ}{d\alpha}(\alpha^{(n)})),$$

with $A$ as an iterative optimization algorithm.

6 Further work

Since traceless forward differentiation mode succeeded, the next step is to use ADOL-C trace functionality in order to benefit in improved efficiency by using the reverse mode of AD at appropriate places.

7 Acknowledgements

The authors are very thankful to Mr. Sergey Slyadnev (OpenCASCADE) for his support related to the AD part and the CAD itself, and Dr. Tom Verstraete (Queen Mary University of London) for his support related to the U-bend parameterization.

This research is a part of the IODA project - **Industrial Optimal Design using Adjoint CFD**. IODA is Marie Skłodowska-Curie Innovative Training Network funded by European Commission.

References


DiffSharp: An AD Library for .NET Languages

Atılım Güneş Baydin∗ Barak A. Pearlmutter† Jeffrey Mark Siskind‡

April 2016

Introduction

DiffSharp is an algorithmic differentiation (AD) library for the .NET ecosystem, which is targeted by the C# and F# languages, among others. The library has been designed with machine learning applications in mind [1], allowing very succinct implementations of models and optimization routines. DiffSharp is implemented in F# and exposes forward and reverse AD operators as general nestable higher-order functions, usable by any .NET language. It provides high-performance linear algebra primitives—scalars, vectors, and matrices, with a generalization to tensors underway—that are fully supported by all the AD operators, and which use a BLAS/LAPACK backend via the highly optimized OpenBLAS library. DiffSharp currently uses operator overloading, but we are developing a transformation-based version of the library using F#'s “code quotation” metaprogramming facility [2]. Work on a CUDA-based GPU backend is also underway.

The .NET platform and F#

DiffSharp contributes a much needed advanced AD library to the .NET ecosystem, which encompasses primarily the languages of C#, F#, and VB in addition to others with less following, such as C++/CLI, ClojureCLR, IronScheme, and IronPython. In terms of popularity, C# is the biggest among these, coming fourth (after Javascript, SQL, and Java; and before Python, C++, and C) in the 2015 Stack Overflow developer survey; and again fourth (after Java, C, and C++; and before Python, PHP, and VB) in the TIOBE index for March 2016. Initially developed by Microsoft, the .NET platform has recently transformed into a fully open source endeavor overseen by the .NET Foundation, and it is currently undergoing a transition to the open source and cross platform .NET Core project, supporting Linux, OS X, FreeBSD, and Windows.

F# is a strongly typed functional language—also supporting imperative and object oriented paradigms—that originated as an ML dialect for .NET and maintains a degree of compatibility with OCaml. F# is gaining popularity as a cross-platform functional language, particularly in the field of computational finance. Languages that support higher-order functions are particularly appropriate for AD as the AD operators are themselves higher-order functions. We have developed implementation strategies that allow AD to be smoothly integrated into such languages, and allow the construction of aggressively optimizing compilers. F# allows DiffSharp to expose a natural API defined by higher-order functions, which can be freely nested and curried, accept first-class functions as arguments, and return derivative functions. The library is usable from F# and all other .NET languages. We provide an optional helper interface for C# and other procedural languages.

Project organization and example code

The code for DiffSharp is released under the GNU Lesser General Public License (LGPL) and maintained in a GitHub repository. The user community has been engaged in the project by raising issues on GitHub and joining in the Gitter chat room.

The library is structured into several namespaces. The main AD functionality is in the DiffSharp.AD namespace, but numerical and symbolic differentiation are also provided in DiffSharp.Numerical and DiffSharp.Symbolic.

∗Corresponding Author, Dept of Computer Science, National University of Ireland Maynooth, gunes@cs.nuim.ie
†Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net
‡School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu
1http://diffsharp.github.io/DiffSharp/
2http://www.openblas.net/
3For a full list, see: https://en.wikipedia.org/wiki/List_of_CLI_languages
5http://www.tiobe.com/tiobe_index
6https://dotnet.github.io/
8The LGPL license allows the use of unmodified DiffSharp binaries in any (including non-GPL or proprietary) project, with attribution.
9https://github.com/DiffSharp/DiffSharp
10https://gitter.im/DiffSharp/DiffSharp
All of these implementations share the same differentiation API (Table 1) to the extent possible, and have support both 32-bit and 64-bit floating point. (Lower precision floating point is of particular utility in deep learning.) The `DiffSharp.Backend` namespace contains the optimized computation backends (currently OpenBLAS, with work on a CUDA backend underway). These namespaces and functionality are directly usable from C# and other .NET languages. For making the user experience even better for non-functional languages, we provide the `DiffSharp.Interop` interface that wraps the AD and numerical differentiation functionality, automatically taking care of issues such as conversions to and from `FSharp.Core.FSharpFunc` objects.\(^{11}\)

Extensive documentation on the library API,\(^ {12}\) along with tutorials and examples, are available on the project website. The examples include machine learning applications, gradient-based optimization algorithms, clustering, Hamiltonian Markov Chain Monte Carlo, and various neural network architectures.

### Key features and contributions

#### Higher-order functional AD API

The fundamental elements of DiffSharp’s API are the `jacobianv` and `jacobianTv` operations, corresponding to the Jacobian-vector product (forward mode) and the Jacobian-transpose-vector product (reverse mode), respectively. The library exposes differentiation functionality through a higher-order functional API (Table 1), where operators accept functions as arguments and return derivative functions. For instance, for a function \( f : \mathbb{R}^n \to \mathbb{R}^m \), the `jacobianTv` operation, with type \( (\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}^n \to (\mathbb{R}^m \times (\mathbb{R}^m \to \mathbb{R}^n)) \), evaluates the function at a given point, and returns the function value together with another function that can be repeatedly called to compute the adjoints of the inputs using reverse mode AD. The API also includes specialized operations (e.g., `hessianv` for Hessian-vector product) to cover common use cases and encourage modular code. This allows succinct implementation of differentiation-based algorithms. For instance, Newton’s method for optimization can be simply coded as:

```csharp
// eps : threshold, f: function, x: starting point
let rec argminNewton eps f x =
    let g, h = gradhessian f x
    if DV.12norm g < eps then x else argminNewton eps f (x - DM.solvSymmetric h g)
```

Note that the caller of `argminNewton` need not be aware of what, if any, derivatives are being taken within it. DiffSharp provides a fixed-point-iteration operator, with appropriate forward and reverse AD rules [5]. The forward mode is handled by iterating until convergence of both the primal and the tangent values, while reverse mode\(^ {13}\) uses the “two-phases” strategy [6].

#### Nesting

All the AD operators can be curried or nested. For instance, making use of currying, the internal implementation of the `hessian` operator in DiffSharp is simply

```csharp
let inline hessian f x = jacobian (grad f) x
```

resulting in a forward-on-reverse AD evaluation of the Hessian of a function at a point.

In another example, we can implement \( z = \frac{d}{dx} \left( x \left( \frac{d}{dy} (x + y) \bigg|_{y=1} \right) \right) \bigg|_{x=1} \) in F# as

```csharp
let z = diff (fun x -> x * (diff (fun y -> x + y) (D 1.))) (D 1.)
```

This can be written in C#, using `DiffSharp.Interop`, as

```csharp
var z = AD.Diff(x => x * AD.Diff(y => x + y, 1), 1);
```

Correctness of AD in the presence of nesting requires avoiding perturbation confusion [7]. For instance, in the above example of nested derivatives, DiffSharp correctly returns 1 (val z : D = D 1.0), while an implementation suffering from perturbation confusion might return 2. We avoid perturbation confusion by tagging values to distinguish nested invocations of the AD operators. See [8, 3, 4, 9] for further discussion.

#### Linear algebra primitives

One can automatically handle the derivatives of linear algebra primitives using an “array-of-structures” approach where arrays of AD-enabled scalars would give correct results for derivatives, albeit with poor performance and high memory consumption. This approach was used in DiffSharp until version 0.7, at which point the library was rewritten using a “structure-of-arrays” approach where vector and matrix types internally hold separate arrays for their primal

\(^{11}\)http://diffsharp.github.io/DiffSharp/csharp.html

\(^{12}\)http://diffsharp.github.io/DiffSharp/api-overview.html

\(^{13}\)Currently, when using reverse mode, closed-over variables in the functional argument to the fixed-point operator should be exposed by manual closure conversion. We hope to lift this restriction soon.
The main motivation for this is our interest in efficiently implementing convolutional neural networks.

We provide benchmarks measuring the AD runtime overhead of the differentiation operations in the API. For application domains heavily using linear algebra, such as training a neural network, functions $f$ and derivative values, and the library recognizes linear algebra operations such as matrix multiplication as intrinsic functions [10]. This allows efficient vectorization of AD, where the underlying linear algebra operations can be delegated to highly optimized BLAS/LAPACK libraries. This approach to AD with linear algebra primitives has been adopted, for example, for the GPU-based C++ reverse AD implementation of Grens et al. [11]. It is interesting to note that for application domains heavily using linear algebra, such as training a neural network, 14 applications typically spend more than 90% of their running time in external BLAS/LAPACK libraries. The AD library’s role in a setting like this is reduced to the intelligent plumbing of primal and derivative arrays to the external library.

### Benchmarks

We provide benchmarks measuring the AD runtime overhead of the differentiation operations in the API. The code for the benchmarks is available in the GitHub repository and we also distribute a command line benchmarking tool with each release. We intend to add memory consumption figures to these benchmarks in the upcoming release.

### Current work

#### Generalization to tensors

DiffSharp currently provides scalar (D), vector (DV), and matrix (DM) types. We are working on generalizing these to an n-dimensional array type, with capabilities similar to those of the Torch Tensor class 17 or the NumPy ndarray. The main motivation for this is our interest in efficiently implementing convolutional neural networks.

#### Source transformation

The library is currently implemented using operator overloading. One of the reasons why F# is an interesting language for AD is its advanced metaprogramming features. The “code quotations” feature 18 allows one to programmatically

---

18http://docs.scipy.org/doc/numpy-1.10.0/reference/arrays.ndarray.html
read and generate abstract syntax trees of functions passed as arguments. The symbolic differentiation module in DiffSharp already makes use of code quotations. We are developing a source-transformation-based AD implementation using this feature, which should result in both speedups and simplification of the API.

**GPU backend**

The backend interface that we defined while vectorizing DiffSharp allows us to plug in other computation backends that the user can select to run their AD code. Our current work on DiffSharp includes the implementation of a CUDA-based backend using cuBLAS for BLAS operations, custom CUDA kernels for non-BLAS operations such as element-wise function application, and cuDNN for convolution operations.

**The Hype library**

DiffSharp will be maintained as a basis library providing an AD infrastructure to .NET languages, independent of the application domain. In addition to setting up this infrastructure, we are interested in using generalized nested AD for implementing machine learning models and algorithms. For this purpose, we started developing the Hype library\(^{19}\) which uses DiffSharp. Hype is in early stages of its development and is currently shared as a proof-of-concept for using generalized AD in machine learning. It showcases how the combination of nested AD and functional programming allows succinct implementations of optimization routines\(^{20}\) (e.g., stochastic gradient descent, AdaGrad, RMSProp), and feedforward and recurrent neural networks. Upcoming GPU and tensor support in DiffSharp is particularly relevant in this application domain, as these are essential to modern deep learning models.

**Conclusions**

Although DiffSharp started as a vehicle for conducting research at the intersection of AD and machine learning, it has grown into an industrial-strength AD solution for F# in particular and the cross-platform .NET platform in general. Its functional API, combined with the ability to freely nest constructs, allows for the convenient implementation of highly modular AD-using software, as seen in the Hype library. We aim to finalize our work on the GPU backend and tensors before September 2016. Readers are invited to refer to the online documentation and code for more in-depth information.

**Acknowledgments**

This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/12637 and by NSF grant 1522954-IIS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

**References**

5. Sebastian Schlenkerich and Andrea Walther. Differentiating fixed point iterations with ADOL-C. Presentation at the 2nd European Workshop on Automatic Differentiation, 2005.

\(^{19}\)http://hypelib.github.io/Hype/

\(^{20}\)https://github.com/hypelib/Hype/blob/master/src/Hype/Optimize.fs
Tricks from Deep Learning

Atılım Güneş Baydin∗ Barak A. Pearlmutter† Jeffrey Mark Siskind‡

April 2016

Introduction

The deep learning [1, 2, 3] community has devised a diverse set of methods to make gradient optimization, using large datasets, of large and highly complex models with deeply cascaded nonlinearities, practical. Taken as a whole, these methods constitute a breakthrough, allowing computational structures which are quite wide, very deep, and with an enormous number and variety of free parameters to be effectively optimized. The result now dominates much of practical machine learning, with applications in machine translation, computer vision, and speech recognition. Many of these methods, viewed through the lens of algorithmic differentiation (AD), can be seen as either addressing issues with the gradient itself, or finding ways of achieving increased efficiency using tricks that are AD-related, but not provided by current AD systems.

The goal of this paper is to explain not just those methods of most relevance to AD, but also the technical constraints and mindset which led to their discovery. After explaining this context, we present a “laundry list” of methods developed by the deep learning community. Two of these are discussed in further mathematical detail: a way to dramatically reduce the size of the tape when performing reverse-mode AD on a (theoretically) time-reversible process like an ODE integrator; and a new mathematical insight that allows for the implementation of a stochastic Newton’s method.

The Deep Learning Mindset

The differences in mindset between the AD and deep learning communities are rooted in their different goals and consequent practices. To grossly caricature the situation, the communities have different typical workflows.

<table>
<thead>
<tr>
<th>A Typical AD Workflow</th>
<th>A Typical Deep Learning Workflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primal computation (e.g., climate simulation) is given.</td>
<td>1. Construct primal process whose derivatives are “nice”, meaning easy to calculate and to use for optimization.</td>
</tr>
<tr>
<td>2. Calculate exact derivatives (e.g., the gradient) automatically and efficiently.</td>
<td>2. Manually code approximate (e.g., stochastic) gradient calculation.</td>
</tr>
<tr>
<td>3. Use these derivatives (e.g., for sensitivity analysis or in a standard gradient optimization method).</td>
<td>3. Use in custom manually-assisted stochastic gradient optimization method.</td>
</tr>
<tr>
<td>4. Iterate to improve end-to-end accuracy throughout.</td>
<td>4. Iterate to improve generalization on novel data.</td>
</tr>
</tbody>
</table>

Given these different settings, it is understandable that the deep learning community has developed methods to address two problems.

(I) Methods for making a forward process whose gradient is “nice” in an appropriate sense.

The important difference here from the usual situation in AD is that, in the deep learning community, the “primal” computational process being trained needs to have two properties: it needs to be sufficiently powerful to perform the desired complex nonlinear computation; and it has to be possible to use training data to set this primal processes’ parameters to values that will cause it to perform the desired computation. It is important to note that this parameter setting need not be unique.

When using gradient methods to set these parameters, there are two dangers. One is that the system will be poorly conditioned. An intuition for stiffness in this context is that, in a stiff system, changing one parameter requires precise compensatory changes to other parameters to avoid large increases in the optimization criterion. The second danger is that the gradient will be uselessly small for some parameters. Such small gradients would be expected in deeply nested computational process relating inputs to outputs. After all, each stage of processing has a Jacobian. These Jacobians have spectra of singular values. If these singular values are all less than one, then the gradient will be washed out, layer by layer, during the backwards sweep of reverse AD. And if the singular values exceed one, the gradients will instead grow exponentially, leading to a similar sort of catastrophe.

In response to these difficulties, the deep learning community has come up with a variety of architectural features which are added to the primal process to give it a better behaved gradient, and has also come up with

∗Corresponding Author, Dept of Computer Science, National University of Ireland Maynooth, gunes@cs.nuim.ie
†Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net
‡School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu
modifications of the reverse AD procedure resulting in the calculation of what we might call a pseudo-gradient, which is designed to be more useful than the true gradient for purposes of optimization.

(II) Faster and more robust stochastic gradient optimization methods.
Simply calculating the actual gradient is so slow in deep learning systems that it is impractical to use the true gradient for purposes of optimization. Instead much faster computations are used to calculate an (unbiased) estimate of the gradient, and these are used for optimization. Such stochastic optimization [4] were first proposed at the dawn of the computer age, but have in practice been something of a black art, requiring constant manual intervention to achieve acceptable performance. This unfortunate state of affairs has now been rectified [5], with the development of algorithms that are asymptotic improvements over their classical antecedents, as well as methods for automatically controlling the stochastic optimization process not just in its terminal asymptotic quadratic phase (which is in practice of little interest in deep learning systems) but also during the so-called initial transient phase.

Deep Learning Methods

We proceed to discuss methods developed by the machine learning community to address each of these issues, in turn.
First, gradient tricks, namely methods to make the gradient either easier to calculate or to give it more desirable properties. And second, optimization tricks, namely new methods related to stochastic optimization.

Gradient Tricks

A naïve description of the basic setup in deep learning would be a many-layers-deep network of perceptrons, that is trained by computing the gradient of the training error through reverse mode AD [6], usually with some layers of linear convolutions [7, 8] and occasionally recurrent structures [9, 2]. The reality, however, is not so simple, as such systems would exhibit a number of problems.

- Standard gradient descent with random initial weights would perform poorly with deep neural networks [10].
- Reverse AD blindly applied to the forward model would be highly inefficient with regard to space.
- Reverse AD, in most implementations, would unroll the matrix operations of the forward model rendering the adjoint calculations quite inefficient.
- The system would be very poorly conditioned, making standard optimization methods ineffective.
- Parameters would vary wildly in the magnitude of their gradient, making standard optimization methods ineffective, and perhaps even causing floating point underflow or overflow.

Starting with the breakthrough of layer-by-layer unsupervised pretraining (using mainly restricted Boltzmann machines) [11], the deep learning community has been developing many methods to make gradient descent work with deep architectures. These methods (related to model architectures and gradient calculation) are so numerous, that we cannot hope to even survey the current state of the art in anything shorter than a book. Here we discuss only a small sample of methods. Even these benefit from having some structure imposed by dividing them into categories. These categories are rough, with many methods spanning between a few.

Efficiency

Care is taken to fully utilize the available computational resources, and to choose appropriate tradeoffs between speed, storage, and accuracy.

(a) Commuting ∇ with ∑. The gradient operator is linear, and therefore commutes with sum, or equivalently with the average which we will denote with angle brackets (⟨⟩). In other words, \( \nabla_w E(x_p; w) \) is the error of the network with parameters \( w \) on training pattern \( p \). If the left hand side of this equation were coded and the gradient calculated with any AD system using reverse AD, the result would be highly inefficient, as the “forward sweep” would encompass the entire training set, and would consume enormous storage and would be hard to parallelize. The second form, in contrast, lends itself to parallelism across training cases, and each “little” gradient requires a constant and very manageable amount of storage.

(b) Stochastic Gradient. It is not really feasible to calculate the true gradient when there is a large dataset, as just calculating the true error \( E = \langle E_p \rangle = \frac{1}{|D|} \sum_{p \in D} E_p \) over all patterns \( p \) in the dataset \( D \) is impractically slow. Instead a sample is used. Calculating the gradient of the error of just a random sample from the dataset (or sequential windows, assuming the dataset has been randomly shuffled) is much faster. The expectation of that process is the true value, so this is admissible for use in stochastic gradient optimization.

(c) GPUs, and low-precision numbers. Most deep learning work uses GPUs, typically very large clusters of GPUs. 32-, instead of 64-bit, precision floating point is common; and fixed point, with 16-bit or even less precision, is feasible due to the error resiliency of neural networks [12]. No current AD systems can manage the delicate tradeoffs concerning which quantities require more or less precision in their representations. Note that derivatives often need more precision, or have a different numeric range, than the primal quantities they are associated with.

(d) Mini-Batches and data parallelism. In calculating the stochastic gradient, it is tempting to do the minimal amount of computation necessary to obtain an unbiased estimate, which would involve a single sample from the training set. In practice it has proven much better to use a block of contiguous samples, on the order of dozens. So instead of \( E_p \) for some \( p \), one uses \( n^{-1} \sum_{i=0}^{n-1} E_{p+i} \), for some \( p \). This has two advantages: the first is less noise, and the
second is that data-parallelism can be used, with favorable cache properties where every quantity in the network is replaced by an $n$-vector, utilizing SIMD on CPUs and GPUs. Conventional AD systems could not maintain this sort of allocation-free vector parallelism through the reverse AD transform.

(e) **Reversible learning.** In deep learning it is sometimes desired to perform bi-level optimization, usually in order to tune hyperparameters of the training process. Naively, since the primal process is $w(t+1) = w(t) - \eta \nabla_w E(x; w(t))$, this would require saving all the $w(1), \ldots, w(t), w(t+1), \ldots, w(T)$. Since $w(t)$ may consist of millions of parameters, and $T$ will typically be in the millions, this is not really feasible. Fortunately a clever technique has been developed to do reverse AD through this process with less storage [13].\(^1\) The trick is to note that this process looks suspiciously like integration of a time-reversible ODE. So we could try to run the system backwards while in the reverse AD reverse pass, calculating necessary quantities as needed: $w(t) = w(t+1) + \eta \nabla_w E(x; w(t+1))$. The would be approximate due to floating point inaccuracy as well as the use of $w(t+1)$ rather than $w(t)$ in $E(\cdot)$. However these inaccuracies would typically be very small, so the difference between these could be computed during the forward sweep, and encoded efficiently in a highly compressed form.

This same technique seems also applicable to performing reverse AD through any process which is (theoretically) time-reversible, such as most ODE or PDE integrators. Of course, the savings here amount to only a constant factor (albeit perhaps a very large one) over standard reverse AD. Combining the method with checkpoint reverse would seem natural, and would allow much larger leaf nodes in the checkpoint AD reverse computation graph.

### Vanishing or Exploding Gradients

In a multi-layered structure, one would expect the gradients of quantities at early layers to be nearly zero (assuming the gains at intermediate levels are below unity) or to be enormous (assuming the gains are above unity). Some methods avoid this. Others work around it, effectively doing a block-structured diagonal pre-conditioning of the gradient.

(a) **Rectified linear units instead of sigmoids.** Classic multi-layer perceptrons use the sigmoid transfer function $\xi \mapsto 1/(1 + \exp(-\xi))$, but this has a derivative which goes to zero when $\xi \gg 0$. That means that when a unit in the network receives a very strong signal, it becomes difficult to change. Using a rectified linear unit (ReLU) transfer function, $\xi \mapsto \max(0, \xi)$ overcomes this problem, making the system more plastic even when strong signals are present. Many variants of this have been proposed, often to avoid the lack of continuity in the derivative of the ReLU. One such are exponential linear units (ELUs) [14].

(b) **Long short-term memory (LSTM).** A recurrent network, unfolded in time, is simply a deep network with some invariants imposed on the parameter matrices. One technique that has proven useful in allowing gradient information to span long temporal spans in the context of recurrent networks, or equivalently many layers in a deep network, is the LSTM architecture [15]. This is essentially a hold-value unit which can have quantities gated in and out. The recent gated recurrent unit (GRU) model [16] is a simplification of this idea.

(c) **Gradient clipping.** In the domain of deep learning, there are often outliers in the training set: exemplars that are being classified incorrectly, for example, or improper images in a visual classification task, or mismarked examples, and the like. These can cause a large gradient inside a single mini-batch, which washes out the more appropriate signals. For this reason a technique called gradient clipping [17] is often used, in which components of the gradient exceeding a threshold (in absolute value) are pushed down to that threshold.

### Conditioning

Keeping the error surface well conditioned for gradient optimization has been one of the keys to the current widespread deployment of deep learning.

(a) **Dropout.** Imagine a network in which multiple units together represent some important feature, requiring a precise weighting of their values in downstream processing. This would make optimization quite difficult, as it sets up couplings between parameters which must be maintained. A technique to avoid such “unfortunate collusions” is dropout [18], in which each unit in the network is, on each training pattern pass, randomly “dropped out” with some probability (typically 50%) by holding its value at zero. This encourages detected features to be independently meaningful. (In “production mode” dropout is turned off, and the weights scaled to compensate, to minimize the noise when performance matters.)

(b) **Careful initialization.** Considering how the variances of activation values and gradients can be maintained between the layers in a network leads to intelligent normalized initialization schemes, which enable substantially faster optimization convergence [10].

### Optimization Tricks

(a) **Early stopping.** When fitting a dynamic system to data, as exact a match as possible is desired, so the true optimum is sought. This is not the case in machine learning, where the optimization is of error on a training set, while the primary concern is generally not performance on the training set, but on as-yet-unseen new data. There is often a tradeoff between the two, encountered after optimization has proceeded for a significant amount of time. This is addressed by early stopping [19], in which an estimate of performance on unseen data is maintained, and optimization is halted early when this estimated generalization performance stops improving, even if performance on the training set is continuing to improve. (An estimate of generalization is often obtained by holding back some training data and using it only for this purpose, and never for optimization.)

---

1. Another possibility, not used in the deep learning community, would be checkpoint reverse mode.
Many AD practitioners have had the demotivating experience of explaining how AD can be used to efficiently calculate Hessian-vector products, only to be asked whether it might be possible to instead calculate the Hessian-inverse-vector product. Although an efficient way to perform this calculation exactly remains unknown, a method has been discovered to efficiently calculate an unbiased estimate of the Hessian-inverse-vector product [20]! Note that we can express a matrix inverse as a series $H^{-1} = \sum_{i=0}^{\infty} (I - H)^i$ where we assume that the spectrum of $H$ allows convergence. We could obtain an unbiased estimate of this sum as follows. Let $p(i)$ be a distribution with support for all integers $i \geq 0$. Choose $i \sim p(i)$. Calculate $p(i)^{-1}(I - H)^i$. This is equal, in expectation, to $H^{-1}$. Similarly $p(i)^{-1}(I - H)^i \cdots ((I - H) v)$ is an unbiased estimate of $H^{-1} v$, and can be computed with $i$ Hessian-vector products. If $p(i)$ is chosen to have small mean, then $i$ will usually be small. If instead of an exact Hessian-vector product we can instead only compute an unbiased estimate of this product, the same procedure will work, except that each $H$ in the above expression becomes $\tilde{H}$, an operator that performs a stochastic unbiased Hessian-vector product. Further development improves the efficiency of the computation, and embeds it in a stochastic Newton’s method with proven efficiency properties.

Conclusion

Many advances in AD, both longstanding and recent, are of great potential utility to machine learning in general and deep learning in particular. Analogously, the machine learning community in general, and the deep learning community in particular, have been using computational derivatives “in anger” for quite some time. They have been forced to build very large systems whose optimization would be intractable using generic AD systems and batch gradient optimization. Necessity has been the mother of invention, and they have discovered a variety of novel methods which allow them to handle large systems and enormous datasets. Many of these methods are related to AD in some fashion, and it is our hope that the AD community will find them of interest.

Acknowledgments

This work was supported in part, by Science Foundation Ireland grant 09/IN.1/12637 and by NSF grant 1522954-IIS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

The Newton Step Method for Algorithmic Differentiation with Implicit Functions

Bradley M. Bell ∗ and Kasper Kristensen †
February 2016

1 Introduction

There are many circumstances where variables are defined implicitly and we need to calculate derivatives of functions that depend on these implicitly defined variables. One example is the implicit function approach to equality constrained optimization [1, Corollary 3.1]; e.g., applied to PDE constrained parameter estimation [2, eq. 2.2]. Another example is nonlinear mixed effects models where the optimal random effects are an implicit function of the fixed effects and the fixed effects objective depends on these random effects [3, eq. 3]. (Nonlinear mixed effects models, as an application of AD, are discussed in detail during this talk.) More generally consider bilevel programming [4], where the current point is such that the implicit function theorem applies to inner variables, and corresponding Lagrange multipliers, as a function of the outer variables; i.e., something similar to but stronger than [5, Assumption 3].

If the implicitly dependent variables are defined by nonlinear equations, they are usually solved by an iterative procedure. In this case there are well known alternatives to applying AD to the iterative procedure; for example, a forward mode AD algorithm that computes these derivatives, for any order, can be found in [6, Section 4.1]. During this talk we consider the Newton Step Method for computing derivatives of functions that are expressed in terms of implicitly dependent variables. This enables one to easily use forward or reverse mode and sparsity when calculating derivatives of functions that depend on implicitly defined variables.

2 Newton Step Representation

Given a function \( L : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^m \), we define the implicit function \( Y(x) \) by

\[
L(x, Y(x)) = 0.
\]

In this section we define a representation for \( Y(x) \) that uses the Newton step corresponding to solving this equation. We then state our main result: the derivatives up to order \( k \) are the same for \( Y(x) \) and the \( k \)-th Newton step representation of \( Y(x) \).

Notation 1 Given a function \( f : \mathbb{R}^f \rightarrow \mathbb{R}^{n \times m} \), and a multi-index \( p \in \mathbb{Z}_+^n \), we use the notation

\[
\partial^p f(x) = \frac{\partial^{p[1]}}{\partial x^{p[1]}} \cdots \frac{\partial^{p[l]}}{\partial x^{p[l]}} f(x).
\]

Definition 2 Given an arbitrary function \( Z : \mathbb{R}^n \rightarrow \mathbb{R}^m \), the corresponding Newton step \( N(Z) : \mathbb{R}^n \rightarrow \mathbb{R}^m \) is defined by

\[
N(Z)(x) = Z(x) - L_y(x, Z(x))^{-1} L(x, Z(x)).
\]

Theorem 3 Fix \( \bar{x} \in \mathbb{R}^n \), for \( k = 0, 1, \ldots \) define \( N_k : \mathbb{R}^n \rightarrow \mathbb{R}^m \) by \( N_0(x) = Y(\bar{x}) \) and

\[
N_{k+1}(x) = N(N_k)(x).
\]

It follows that for any multi-index \( p \in \mathbb{Z}_+^n \), and for any \( k \geq |p| \), \( \partial^p Y(\bar{x}) = \partial^p N_k(\bar{x}) \).

Remark 4 The fact that only one Newton step is required to get a first order accurate representation of \( Y(x) \) is well known; e.g., below [7, eq. 15]. It is worth noting that they, and Theorem 3, use the following definition for the first Newton step representation

\[
N_1(x) = Y(\bar{x}) - L_y(x, Y(\bar{x}))^{-1} L(x, Y(\bar{x})).
\]
If one uses this representation, one must differentiate the equation solution when computing first order derivatives. It is also possible to prove Theorem 3 with the following alternative definition for $N_1(x)$:

$$N_1(x) = Y(\bar{x}) - L_y(\bar{x}, Y(\bar{x}))^{-1}L(x, Y(\bar{x})).$$

This representation does not require one to differentiate the equation solution when using one Newton step.

**Remark 5** Checkpointing [8] is a technique for reducing the memory required by AD. Some operator overloading AD packages enable one to make one recording of the Newton step

$$(x, y) \rightarrow (x, y - L_y(x, y)^{-1}L_x(x, y))$$

and then use it to represent $N_{k+1}(x)$ in terms of $N_k(x)$; e.g., the **cppad.mixed** packages takes advantage of this technique [9].

### 3 Nonlinear Mixed Effects Models

There are many alternatives to estimating parameters in mixed effects models [10]. The Laplace approximation for the objective is considered difficult to optimize because it is expressed in terms of second derivatives of the model functions; see eq. (2) below. The approximation is often modified to use first order, instead of second order, derivatives; see [10, eq. 2.7]. Many researchers avoid the problem by restricting their attention to Gaussian linear models where the Laplace approximation is exact and can be computed in closed form; e.g., [11, eq. 2].

The Laplace approximation, and more general applications of mixed effects models, have gained popularity with the advent of better algorithmic differentiation tools that include the ability to automatically detect and use sparsity; e.g. [12]. Let $x$ represent the fixed effects, $y$ the random effects, and $H(x, y)$ the negative log likelihood of the data, random effects, and fixed effects. The Laplace approximation of the objective, for estimating the fixed effects $x$, is [12, eq. 4]

$$\frac{1}{2} \log \det (H_{yy}(x, Y(x))) + H(x, Y(x)).$$

(2)

were $Y(x)$ minimizes $H(x, y)$ with respect to $y$. If the Hessian $H_{yy}(x, y)$ is always positive definite, $Y(x)$ solves the implicit Equation

$$L(x, Y(x)) = H_y(x, Y(x))^T = 0.$$ 

In this case, we can compute the Hessian of $H(x, Y(x))$ with respect to $x$ using the one Newton step function $N_1(x)$, see [13, Theorem 2]. It is the log determinant term

$$D(x) = \frac{1}{2} \log \det (H_{yy}(x, Y(x))) ,$$

that requires the two Newton step function $N_2(x)$. Using some AD packages, one can make a recording of $H_{yy}(x, y)$ that in turn allows one to record the operations in

$$\tilde{D}(x, \bar{x}) = \frac{1}{2} \log \det (H_{yy}(x, N_2(x))) ,$$

with both $x$ and $\bar{x}$ included in the independent variable vector. This avoids the need to make a different recording for each $\bar{x}$. It follows that we can use AD to compute

$$D_{x, \bar{x}}(\bar{x}) = \tilde{D}_{x, \bar{x}}(\bar{x}, \bar{x}) .$$

This approach is implemented in the **cppad.mixed** package; see [9]. While this may be an easy way to compute this Hessian of $D(x)$, other alternates that may yield faster run times, or use less memory, or both.

Note that the Laplace approximate objective is expressed in terms of second derivatives of the model function $H(x, y)$. Thus, calculating the Hessian of the objective requires fourth order derivatives of the original model functions. For most random effects modeling applications, the Hessian

$$H_{yy}(x, y) = L_y(x, y)$$

is sparse, positive definite, and these calculations can be done for large problems. For example, when the random effects are a Gauss Markov random field where Hessian is closely related to the precision matrix [14, eq. 3]. Even if one does not use the Hessian of the objective during optimization, it may be used during calculation of the observed information.
REFERENCES


First-order Derivatives of Associated Legendre Functions

H. Martin Bücker* and Johannes Willkomm†

March 2016

1 Introduction

The scientific analysis for the solution of physical problems encounters a variety of special functions. Well-known examples of special functions include Hermite polynomials, Legendre polynomials, and Bessel functions [1]. Therefore, it is not uncommon that computer models in computational science and engineering employ these special functions in some way or the other. If such computer models are transformed by automatic differentiation (AD), the derivatives of these special functions will be necessary. If the source code for the evaluation of these functions is available it can be transformed in a black-box AD approach. That is, the code implementing the special function is treated in the same way as any other code without considering its semantics. However, it is well known that the semantics can and should be exploited to apply the chain rule more judiciously. The corresponding techniques are sometimes called hierarchical AD approaches [2, 3, 4, 5]. They are successfully used to increase performance in terms of computing time and storage in various different situations including the solution of systems of linear and nonlinear equations, or Fourier transforms to name a few.

This note is concerned with a hierarchical AD approach for the special functions called the associated Legendre functions. These functions frequently arise, for instance, from quantum physics during the solutions of the Schrödinger equation in spherical polar coordinates [6]. In MATLAB, Octave, and Scilab, the associated Legendre functions are part of the programming language. So, from a conceptual point of view, AD can treat them as “elemental functions.”

2 Associated Legendre functions

The associated Legendre functions of degree \( n \geq 0 \) consist of a set of \( n + 1 \) functions defined over \(-1 \leq x \leq 1\). They are given by

\[
P_m^n(x) = (-1)^m (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_n(x), \quad m = 0, 1, \ldots, n,
\]

where \( m \) is called the order and

\[
P_n(x) = \frac{1}{2^m m!} \left[ \frac{d^m}{dx^m} (x^2 - 1)^n \right]
\]

is the Legendre polynomial of degree \( n \). This definition shows that the associated Legendre functions of degree \( n \) reduce to the corresponding Legendre polynomials if the order is \( m = 0 \). That is,

\[
P_0^n(x) = P_n(x).
\]

In Figure 1, the associated Legendre functions \( P_m^n(x) \) are shown for the degree \( n = 3 \) where \( m = 0, 1, 2, 3 \).

The associated Legendre functions are commonly used with three different normalization schemes. The functions \( P_m^n(x) \) given in the scheme (1) are called unnormalized. A second normalization scheme consists of Schmidt seminormalized associated Legendre functions defined by

\[
S_m^n(x) = \begin{cases} (-1)^m \sqrt{\frac{2(n-m)!}{(n+m)!}} P_m^n(x) & \text{if } m \geq 1, \\ P_0^n(x) & \text{if } m = 0. \end{cases}
\]

Thus, the Schmidt seminormalized associated Legendre functions \( S_m^n(x) \) coincide with the unnormalized functions for \( m = 0 \) and, by (2), they also reduce to the Legendre polynomials \( P_n(x) \) in this special case. The third normalization scheme is given by

\[
N_m^n(x) = (-1)^m \sqrt{\frac{(n+1/2)(n-m)!}{(n+m)!}} P_m^n(x).
\]

Thus, these functions \( N_m^n(x) \) are called normalized associated Legendre functions.

*Corresponding Author, Friedrich Schiller University Jena, martin.buecker@uni-jena.de
†Work partially performed while employed at Technische Universität Darmstadt, johannes@johannes-willkomm.de
3 Analytic derivatives of the associated Legendre functions

It is well known [1] that the derivatives of the associated Legendre functions of degree \( n \geq 0 \) satisfy the recurrence relation

\[
(1 - x^2) \frac{dP_m^n(x)}{dx} = (n + m)P_{m-1}^n(x) - nx P_m^n(x), \quad m = 0, 1, \ldots, n, \tag{3}
\]

for \( n \geq 0 \). Here, the initial condition is given by

\[
P_{m-1}^n(x) \equiv 0, \quad m = 0, 1, \ldots, n,
\]

and the notation

\[
P_{m-1}^n(x) \equiv 0 \quad \text{if} \quad m = n
\]
is used. Equation (3) is sufficient to evaluate the derivatives \( dP_m^n(x)/dx \) for \( n \geq 0 \) in the domain \(-1 < x < 1\). In particular, we immediately find that these derivatives are zero for the degree \( n = 0 \). That is,

\[
\frac{dP_0^0(x)}{dx} \equiv 0.
\]

In contrast to (1) which is defined over \(-1 \leq x \leq 1\), the derivatives in (3) are not defined at \( x = \pm 1 \). However, we derive the one-sided limits of \( dP_m^n(x)/dx \) when \( x \to 1 \) from the left and \( x \to -1 \) from the right. More precisely, we prove the following.

**Lemma.** The derivatives of the associated Legendre functions \( P_m^n(x) \) of degree \( n \geq 0 \) and order \( m = 0, 1, \ldots, n \) evaluated at the boundary \( x = 1 \) are given by

\[
\frac{dP_m^n(x)}{dx} \bigg|_{x=1} = \begin{cases} 
\frac{n(n+1)}{2} & \text{if } m = 0, \\
\infty & \text{if } m = 1, \\
-(n-1)n(n+1)(n+2)/4 & \text{if } m = 2, \\
0 & \text{if } m = 3, 4, \ldots, n.
\end{cases}
\]

The corresponding derivatives evaluated at the boundary \( x = -1 \) satisfy

\[
\frac{dP_m^n(x)}{dx} \bigg|_{x=-1} = \begin{cases} 
(-1)^{n+1} \cdot n(n+1)/2 & \text{if } m = 0, \\
(-1)^n \cdot \infty & \text{if } m = 1, \\
(-1)^n \cdot (n-1)n(n+1)(n+2)/4 & \text{if } m = 2, \\
0 & \text{if } m = 3, 4, \ldots, n.
\end{cases}
\]
4 Derivatives of the MATLAB function legendre

In MATLAB, there is a function `legendre` to evaluate the associated Legendre functions at various points \( x_i \) simultaneously. The syntax of this function is given by

\[
P = \text{legendre}(n,X) \quad S = \text{legendre}(n,X,'sch') \quad N = \text{legendre}(n,X,'norm')
\]

which computes the functions \( P_n^m(x) \), \( S_n^m(x) \), and \( N_n^m(x) \), respectively, for all \( m = 0, 1, \ldots, n \). Here, the symbol \( n \) denotes the degree of the associated Legendre functions. The variable \( X \) stores the set of points \( x_i \) where the functions are evaluated. As usual in MATLAB, the variable \( X \) could be scalar or a multidimensional array \( X(i,j,k,\ldots) \). In the general case, the element \( P(m+1,i,j,k,\ldots) \) of the return value \( P \) contains the associated Legendre function \( P_n^m(x) \) for \( m = 0, 1, \ldots, n \) evaluated at \( X(i,j,k,\ldots) \). The situations for \( S_n^m(x) \) and \( N_n^m(x) \) are similar. The third parameter is optional and denotes the normalization scheme. By default, the function `legendre` computes the unnormalized associated Legendre functions.

From a conceptual point of view, the set of evaluation points \( x_i \) on input to `legendre` is represented by some vector \( x \) of length \( q \). Similarly, the corresponding output \( P_n^m(x) \) for \( m = 0, 1, \ldots, n \), where \( x \) is evaluated at all entries of \( x \), is some vector \( y \) of length \((n+1)q\). We use the linear indexing given by

\[
x = (x_1, \ldots, x_q) \quad \text{and} \quad y = (y_1, \ldots, y_{(n+1)q}).
\]

The way how the input data structure \( X(i,j,k,\ldots) \) is arranged in the conceptual vector \( x \) affects the meaning of the indexing. Likewise, the arrangement of the output data structure, say \( P(m+1,i,j,k,\ldots) \), in the vector \( y \) influences the meaning of its indices. Without loss of generality, we assume that this flattening of a multidimensional data structure onto a linear index follows the column-major order which is also used in MATLAB. That is, in the linear index, the columns of a multidimensional data structure are listed one after the other. For the sake of notational simplicity, we further assume that the variable \( X \) is indeed a vector with \( q \) entries in which case the variable \( P \) is an \((n+1) \times q\) matrix. So, the value \( P_n^m(x_j) \) is represented by \( y_i \) whose index is given by

\[
i = (n+1)(j-1) + m + 1. \tag{4}
\]

The function `legendre` implements a mathematical function that maps \( q \) scalar entries from \( x \) to \((n+1)q\) scalar entries in \( y \). Thus, its Jacobian

\[
J(x) := \frac{\partial y}{\partial x}
\]

is an \((n+1)q \times q\) matrix. The Jacobian entry at position \((i,j)\) corresponds to

\[
\frac{\partial y_i}{\partial x_j} = \frac{dP_n^m(x_j)}{dx_j},
\]

where \( i, j \) and \( m \) are related by (4).

Clearly, a value of an associated Legendre function \( y_i \) evaluated at some \( x_j \) does not depend on any other point of evaluation \( x_k \) with \( k \neq j \). Therefore, each row \( i \) of the Jacobian (5) contains at most a single nonzero entry. Furthermore, a point \( x_j \) can only have an effect on the associated Legendre values \( y_i \) with \( i \) from (4) where the order varies in the range \( m = 0, 1, \ldots, n \). Therefore, each column \( j \) of the Jacobian (5) contains at most \( n+1 \) successive nonzero entries.

In Figure 2, the pattern of nonzero elements of the Jacobian of \( P_n^m(x) \) is depicted. More precisely, this pattern belongs to the associated Legendre functions given in Figure 1 that are evaluated at \( q = 21 \) points. Here, the degree is \( n = 3 \) and the Jacobian (5) is a \( 84 \times 21 \) matrix. The entries of this Jacobian are shown graphically in Figure 3 that plots \( dP_n^m(x)/dx \) versus \( x \). In general, each row of the Jacobian contains a single nonzero element and each column contains four nonzero elements. However, there are exceptions in the positions \((42,11)\) and \((44,11)\). Column 11 corresponds to the point of evaluation \( x = 0 \) and, by inspection of (4), we find that rows 42 and 44 represent \( m = 1 \) and \( m = 3 \), respectively. The corresponding nonzero entries happen to be zero as is obvious from Figure 3.

Figure 2: Sparsity pattern of the \( 84 \times 21 \) Jacobian \( \partial y/\partial x \) of the associated Legendre functions given in Figure 1 with degree \( n = 3 \) and \( q = 21 \) evaluation points.
5 Concluding remarks

Hierarchical AD approaches, sometimes also called local preaccumulation, provide a mechanism for exploiting program structure at various levels of the underlying computational graph. A natural level consists of subroutines or mathematical functions that are accessible via well-defined interfaces. Here, we employ a hierarchical AD approach to compute the derivatives of the associated Legendre functions which are present in programming languages such as MATLAB. These special functions are defined on the interval $[-1,1]$. The benefit of this approach is that the one-sided limits of the derivatives at the endpoints of this interval can be treated rigorously. At the conference, we will not only present the proof of the Lemma, but we will also show implementation details of how this hierarchical approach is integrated into the software tool ADiMat [7] which implements AD for programs written in MATLAB.

References


1 Preserving a Given OpenMP Parallelization

Despite the ubiquity of parallelism in modern computer architectures there are currently only a few automatic differentiation (AD) tools that offer limited support for OpenMP parallelism. Most of current AD tools ignore OpenMP directives in the program code. That is, the AD transformations treat these directives as comments. Consider the OpenMP-parallelized code fragment given in Fig. 1a. Now suppose that this code fragment is differentiated in the forward mode propagating $N$ scalar directional derivatives. The resulting differentiated code is depicted in Fig. 1b. Here, the differentiated program is parallelized following the parallelization specified in the original program.

This AD transformation does not change the parallelization of the given OpenMP program. That is, the additional computational work for the derivative calculations is parallelized in the same way as the computational work in the original code. For example, the one-dimensional array variable $x$ of length $L$ in the original program is associated with a two-dimensional derivative object $g_x$ of dimension $N \times L$ in the differentiated program. Following the parallelization strategy of the original code, this two-dimensional derivative variable, $g_x$, is distributed among the OpenMP threads in a column-wise manner as depicted in Fig. 2a. Depending on the scheduling strategy applied to the outer loop iterations $i=1,L$, each thread is assigned to work on a certain number of columns of the derivative variables. In particular, the thread working on element $i$ of the array variable $x$ in the original code is working on the $i$-th column of the derivative variable $g_x$. In summary, preserving the original parallelization within the differentiated code as described above has the following properties:

- The synchronization patterns of the original and differentiated OpenMP codes are identical. But, since the computational work is increased in the differentiated code, the ratio of synchronization time and computing time typically decreases, possibly leading to improved scalability of the differentiated code.

- Depending on the actual problem at hand the overall amount of data to be accessed via the shared memory increases in the differentiated code, yielding increased data traffic consuming more memory bandwidth compared to the original code.

---

*Friedrich Schiller University Jena, martin.buecker@uni-jena.de
†Düsseldorf, Germany
‡Corresponding Author, Friedrich Schiller University Jena, ralf.seidler@uni-jena.de
§Technische Universität Darmstadt, Andreas.Wolf@hrz.tu-darmstadt.de
Figure 2: Work distributions of a two-dimensional derivative object $g_{x}$ on four threads on the physical processors $p_1, \ldots, p_4$. (a) One-dimensional distribution along the columns that preserves the existing parallelization. (b) One-dimensional distribution along the rows that ignores the existing parallelization. (c) Two-dimensional distribution along columns and rows that uses a nested parallelism.

- The memory access pattern of the differentiated code typically facilitates cache reuse and data prefetching. This may also positively affect the performance of the parallel code.

However, the scalability of this approach may be limited due to the limited scalability of the original parallelization, e.g., the granularity of parallelization in the original code may be limited by the nature of the problem. Yet, this approach is currently considered to be the strategy of choice for most if not all AD tools supporting OpenMP [1, 2].

2 Ignoring a Given OpenMP Parallelization

An alternative parallelization strategy for this AD transformation is to simply ignore the OpenMP directives from the original code and parallelize only the derivative computation. Ignoring the OpenMP directives in Fig. 1a and applying AD with the parallelization approach from [3] leads to the code fragment displayed in Fig. 3. Using an appropriate driver this code fragment is executed simultaneously by multiple threads, where each thread maintains a private pair of integers ($LB, UB$). Due to the redundant function evaluation suggested in this approach all variables related to the original code are private. The derivative objects, $g_{x}$, $g_{y}$, and $g_{z}$ by contrast, are shared among the threads. This parallelization strategy results in a row-wise partitioning of the derivative objects as shown in Fig. 2b.

```fortran
do i = 1, L
  do j = LB, UB
    g_tmp(j) = cos(x(i)) * g_x(j, i)
  enddo
  tmp = sin(x(i))
  do j = LB, UB
    g_z(j, i) = y(i) * g_tmp(j) + tmp * g_y(j, i)
  enddo
  z(i) = y(i) * tmp
enddo
```

Figure 3: Code resulting from applying the strategy described in [3] to the code given in Fig. 1a. The code is executed by multiple threads. Each thread is assigned a particular range of loop iterations over the derivative objects. This is achieved by assigning each thread a different pair of integer values ($LB, UB$).

Since the work on two-dimensional derivative objects is assigned to physical processors in a row-wise fashion, the size of the first dimension $N$, i.e., the number of independent variables, must be significantly larger than the number of processors. Otherwise, due to column-wise memory layout in Fortran, many cache lines are accessed by more than one thread, resulting in a phenomenon known as false sharing. False sharing occurs when different processors simultaneously write to a shared cache line but not at the same location. Each update of an individual element of a cache line by one processor requires the other processors to fetch a fresh copy of the line from memory (even though the element accessed has not been modified). Thus, false sharing can generate huge memory traffic and can degrade the overall performance of the parallel program significantly. In summary, the strategy has the following characteristics:

- The main advantage is the absence of any synchronization during the computation.
- The distribution of the multi-dimensional derivative objects is harmful and, under certain circumstances, is likely to cause a large number of false sharing events.
3 Nested Parallelism

The parallelization of the loops over the derivative entries is independent from the parallelization that is already present
in the original code. Thus, these two parallelization strategies can be combined. A first step in this direction is made
in [4], where AD is viewed as a code transformation naturally adding another level of parallelism to any OpenMP code.
This additional level adds appropriate OpenMP directives to the loops performing the vector linear combinations on
the derivative objects. More precisely, applying this approach given to an already parallelized original code results
in a parallel code with nested parallelism where the innermost level of parallelism corresponds to the parallelization
of the original code. As an example, consider the differentiated Fortran code depicted in Fig. 4 corresponding to the
original code given in Fig. 1a. Several instances of this code are invoked simultaneously by an appropriate driver.
Each instance, however, creates a new team of threads sharing the work in the parallel loop over \( L \), as indicated by the
\texttt{parallel do} construct in line 1. The goal of this nested parallelization strategy for differentiated OpenMP code

```c
C$OMP parallel do private(i,tmp) shared(L,x,y,z)
  do i = 1, L
    do j = LB, UB
      g_tmp(j) = cos(x(i)) * g_x(j, i)
    enddo
    tmp = sin(x(i))
    do j = LB, UB
      g_z(j, i) = y(i) * g_tmp(j) + tmp * g_y(j, i)
    enddo
  z(i) = y(i) * tmp
C$OMP end parallel do
```

Figure 4: Code resulting from applying the new strategy involving nested parallelism to the code given in Fig. 1a.

is to increase the scalability of the parallelization that is already present in the original code. This is achieved by
distributing the available threads on both levels of parallelism, reducing the disadvantageous effects that would occur
when all available threads are assigned to either one of the parallelization levels. This results in a two-dimensional
distribution of work on the derivative objects, as is schematically depicted in Fig. 2c. This new nested parallelization
approach is characterized by the following advantages:

- While each type of single-level parallelization is expected to yield good speedup for smaller number of threads
  and decreasing speedup for larger number of threads, we expect the combination of both strategies to perform
  better even for larger number of threads.
- The granularity of the original parallelization may be increased.
- For a fixed number of threads, the presented approach offers the opportunity for arbitrary distribution of the
  available threads on both levels of parallelism. However, the optimal distribution is problem-dependent.

4 Performance Results

The experiments presented below were carried out on a dedicated Sun Fire E25K system with 72 UltraSPARC IV
dual core processors. This system employs a cache-coherent non-uniform memory access (ccNUMA). This implies that
data locality plays an important role on this system since accessing data from remote memory is more expensive than
accessing local memory. The test code used in all experiments was generated by version 3.0 of Adifor [5] differentiating
a subroutine with \( 1098 \cdot k \) independent variables. Four different problem sizes have been considered, i.e., \( k \in \{1,2,3,4\} \).
OpenMP directives in the AD-generated code have been edited manually in order to provide correct scoping clauses
for the derivative objects. The execution times on a single processor are 2.63, 23.84, 91.33, and 213.0 seconds for
\( k = 1,2,3, \) and 4, respectively. Performance experiments with the two parallelization strategies presented above have been
carried out using up to 121 processing elements. In the following discussion, the approach from Section 1 preserving
the original parallelization in the differentiated code will be denoted by \textit{Preserve}. The nested parallelization strategy
presented in Section 3 will be referred to as \textit{Nested}.

The speedup for \textit{Preserve} is depicted in Fig. 5a. Reasonable speedup is achieved in particular for up to 8 threads,
independent of \( k \). For \( k = 1 \) or \( k = 2 \), this approach scales well even for greater number of threads because the small
amount of data fits into cache. When increasing the amount of data with \( k = 3 \) or \( k = 4 \), however, only a moderate
increase of speedup can be observed when the number of threads is increased between 10 and 120. Notice that, due
to the cache effects, the curves for \( k = 3 \) and \( k = 4 \) are significantly below those for \( k = 1 \) and \( k = 2 \).

Next, the speedup for \textit{Nested} is shown in Fig. 5b. The computation starts with \( n \) threads on the first level of
parallelization, i.e., in the outermost parallel region. Each of the \( n \) threads creates another team of \( n \) threads when
entering the second (nested) parallel region. So, the total number of threads is $n^2$. Using approach Nested, a reasonable speedup can be observed for up to 36 threads for $k = 1, 2, 3$ and 4. For $k = 1$, the number of threads that are required to achieve the maximum speedup of 18.5 is 36, which is to some extent similar to approach Preserve. Employing the strategy Preserve in this case, 36 threads yield speedup of approximately 17.5 while the absolute maximum speedup of 20.8 is achieved with 52 threads for this value of $k$. In Nested, for $k = 2$, about 64 threads are required in order to achieve a speedup of approximately 30, which is similar to approach Preserve. For $k = 3$ and $k = 4$ the speedup is generally larger than in the approach Preserve, independent on the number of threads. Moreover, to achieve a speedup of 30.2 or 34.7 for $k = 3$ or $k = 4$, respectively, 64 threads are required in Nested. In contrast, with the approach Preserve, 64 threads only yield a speedup of around 16 and 14, for $k = 3$ and $k = 4$, respectively.

![Graph](a) Approach Preserve

![Graph](b) Approach Nested

Figure 5: Speedup of the two approaches.

## 5 Concluding Remarks

It is commonly assumed that an AD approach for a parallelized program should preserve this given parallelization strategy in the AD-generated code. However, we give evidence by way of an OpenMP example that the performance of the parallelized AD-generated code can be improved by a novel approach that does not preserve the existing parallelization strategy in a one-to-one correspondence. The idea behind this approach is to preserve the given parallelization strategy only partially. More precisely, only a subset of the available threads follows the given parallelization strategy while the remaining threads are used in an orthogonal way to parallelize the loops of the derivatives propagated in the AD-generated code. In general, this approach involving nested parallelism scales better than the standard approach that preserves the existing parallelization strategy. At the conference, we will not only present a detailed comparison of these two parallelization strategies, but will also introduce various different strategies that we omitted to reduce the length of this extended abstract.

## References


Arbogast
Higher order AD for special functions with Modular C

Isabelle Charpentier* and Jens Gustedt†

April 2016

This toolbox for higher order AD is named after L. F. A. Arbogast (1759–1803), a French mathematician from Strasbourg (Alsace), for his pioneering work in derivation calculus [1 2]. He is also responsible for the law introducing the metric system in the French Republic.

1 Introduction

Special functions and their derivatives play a crucial in research fields of physics and mathematical analysis. Considerable research efforts have been directed at implementing special functions in numerical libraries, some of them are proposed in the GNU Scientific Library, without having developed genuine activities within the context of AD. Many special functions, Tab. 1, are solutions of the general second order ordinary differential equation (ODE)

\[ \alpha(z) \varphi^{(2)}(z) + \beta(z) \varphi^{(1)}(z) + \gamma(z) \varphi(z) = 0, \]

where functions \( \alpha(z) \), \( \beta(z) \), \( \gamma(z) \) determine the mathematical function \( \varphi(z) \). In their classical definitions [3], the input \( z \) is either a real or a complex variable. Assuming \( z = z(t) \) is a function depending on some variable \( t \), this ODE yields the general formulation [2],

\[ v^{(0)} = \varphi^{(0)}(z^{(0)}), \quad v^{(1)} = \varphi^{(1)}(z^{(0)})z^{(1)}, \quad v^{(2)} = -\gamma v^{(0)}(z^{(1)})^3 - \beta v^{(1)}(z^{(1)})^2 + \alpha v^{(2)}(z^{(1)}), \quad \frac{\alpha z^{(1)}}{z^{(1)}}, \quad \text{for } z^{(1)} \neq 0, \]

allowing for the higher-order differentiation of the compound function \( v(t) = \varphi \circ z(t) \) [4], the implementation of which may be of quadratic complexity [5]. The case \( z^{(1)} = 0 \) is discussed in [4]. As usual, the derivative and the Taylor coefficient of the function \( z \) at order \( k \) are denoted by \( z^{(k)} \) and \( z_k \), respectively.

One of the goals of this paper here is to show that equations (2) allow for the automatic generation of a higher-order automatic differentiation library comprising special functions satisfying (1). Only seeds \( \varphi^{(0)} = \varphi \) and \( \varphi^{(1)} \), and generating functions \( \alpha(z) \), \( \beta(z) \), \( \gamma(z) \) have to be provided.

<table>
<thead>
<tr>
<th>Function</th>
<th>Seeds</th>
<th>Generating functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \varphi(z^{(0)}) )</td>
<td>( \varphi^{(1)}(z^{(0)}) )</td>
</tr>
<tr>
<td>Faddeeva</td>
<td>( w(z^{(0)}) )</td>
<td>( -2z^{(0)}w(z^{(0)}) + 2i/\sqrt{\pi} )</td>
</tr>
<tr>
<td>Bessel</td>
<td>( J_{\nu}(z^{(0)}) )</td>
<td>( -J_{\nu+1}(z^{(0)}) + \nu/z^{(0)} J_{\nu}(z^{(0)}) )</td>
</tr>
<tr>
<td>Modified Bessel</td>
<td>( I_{\nu}(z^{(0)}) )</td>
<td>( I_{\nu+1}(z^{(0)} + \nu/z^{(0)} I_{\nu}(z^{(0)}) )</td>
</tr>
<tr>
<td>Hypergeometric</td>
<td>( 2F_1(a, b; c; z^{(0)}) ) ( (ab/c)2F_1(a + 1, b + 1; c + 1; z^{(0)}) )</td>
<td>( z(1 - z) - (a + b + 1)z - ab )</td>
</tr>
</tbody>
</table>

Table 1: Seeds and generating functions for some mathematical functions satisfying (1), where \( a \), \( b \), \( c \) and \( \nu \) are parameters [3].

2 Modular C as a tool for automatic code generation

Since decades, C is one of most widely used programming languages, see [6], and is used successfully for large software projects that are ubiquitous in modern computing devices of all scales. C is undergoing a continued process of standardization and improvement and over the years has added features that are important in the context of this study: complex numbers, variable length arrays, long double, type generic mathematical functions (all in C99), programmable type generic interfaces, choosable alignment and unicode support (in C11), see [7].

For many programmers, software projects and commercial enterprises, C has advantages (relative simplicity, faithfulness to modern architectures, backward and forward compatibility) that largely outweigh its shortcomings. Prominent among these shortcomings, is a lack of two closely related features: modularity and reusability. C misses to

*Corresponding Author, CNRS & ICube, University of Strasbourg, France. isabelle.charpentier@unistra.fr
†Inria & ICube, University of Strasbourg, France. jens.gustedt@inria.fr
encapsulate different translation units (TU) properly: all symbols that are part of the interface of a software unit such as functions are shared between all TU that are linked together into an executable.

To cope with these difficulties we recently proposed an extension to the C standard called Modular C, see [8]. It consists in the addition of a handful of directives and a naming scheme transforming traditional TU into modules. The change to the C language is minimal since we only add one feature, composed identifiers, to the core language. Other features of Modular C are implemented through CMOD directives. For our discussion here the important among these are:

import Our modules can import other modules as long as the import relation remains acyclic and a module can refer to its own identifiers and those of the imported modules through freely chosen abbreviations. Other than traditional C include, our import directive ensures complete encapsulation between modules. The abbreviation scheme allows to seamlessly replace an imported module by another one with equivalent interface.

snippet In addition to the export of symbols, we provide parameterized code injection through the import of “snippets”. This implements a mechanism that allows for code reuse, similar to so-called X macros or templates as other programming languages implement them. Listing 1 shows an excerpt of a snippet definition. The related slot directives are used to specify the parameters of a snippet. Then, fill directives are used to specialize these slots for a particular instantiation of the snippet, Listing 2.

foreach When implementing libraries that have to deal with similar code for different base types (here C’s six real and complex floating point types) usually almost identical code is just replicated and then adapted for the target type. Obviously, such a procedure is prone to subtle copy errors that are difficult to find by inspection. Modular C’s foreach directive avoids these problems by allowing parameterized code replication that is completely resolved at compile time.

Additional features of our proposal are a simple dynamic module initialization scheme, a structured approach to the C library, a migration path for existing software projects, and, last but not least complete Unicode integration.

3 The example of the 2nd order ODE

An excerpt of code for the second order ODE method [2] can be seen in Listing 1. This snippet code is then used in Listing 2 to provide the differential operator \( w \circ f_{t_0} \) (DO) for the Faddeeva function, where \( f_{t_0} \) is a real or complex function that is represented by its Taylor polynomial at point \( t_0 \). Another snippet (not shown) is used for families of functions that are themselves parameterized, such as \( J_v \circ f_{t_0} \), \( I_v \circ f_{t_0} \), \( T_n \circ f_{t_0} \) and \( 2 F_1^{(bc)} \circ f_{t_0} \).

We see that this snippet has 11 slots corresponding to identifiers that have to be specialized by each user of the snippet. The first five correspond to type parameters, complete types in the C jargon. The next five are the functions that will give the specifics of the DO that is to be implemented. In fact, the constraints on the right hand side of the sign specify that \( \varphi^0 \) and \( \varphi^1 \) must not necessarily be proper functions, they only have to allow for the evaluation of

1In Listing 1 these correspond to the identifiers segmented by the character ■. Such a character may be chosen for each module.
Listing 2: The Faddeeva DO for _Complex double (cd) implemented through the ODE in [1], code excerpt. On the left the implementation of the interfaces that are needed for the use of the snippet of Listing 1. On right the Modular C specific directives that fill the slots of that snippet.

```c
#define Faddeeva(Z) \ _pragma CMOD import wofz-cd = arbogast
_Generic(Z), \ _pragma CMOD f11 \ wofz-cd TP = tcd
 types tcf: func-cf, \ _pragma CMOD f11 \ wofz-cd Gamma = scd
 default: func-cd(Z) \ _pragma CMOD f11 \ wofz-cd Gamma = tcd
 #define wofz support cerf mw of z \ _pragma CMOD f11 \ wofz-cd Beta = tcd
 inline scd \ _pragma CMOD f11 \ wofz-cd Gamma = tcd
 \(\gamma\) \ _pragma CMOD f11 \ wofz-cd Gamma = tcd
 return \(\beta\) \_pragma CMOD f11 \ wofz-cd Gamma = tcd
 return \(\beta\) \_pragma CMOD f11 \ wofz-cd Gamma = tcd
 return \(\beta\) \_pragma CMOD f11 \ wofz-cd Gamma = tcd
 return \(\beta\) \_pragma CMOD f11 \ wofz-cd Gamma = tcd
 return \(\beta\) \_pragma CMOD f11 \ wofz-cd Gamma = tcd
```

\[ b = \varphi^0(a) \text{ at compile time. So, } \varphi^0 \text{ and } \varphi^1 \text{ can e.g be functions or type generic macros. The last slot, } \Phi, \text{ names the function that this snippet will produce, namely an } \textbf{extern} \text{ function that receives and returns a value of the Taylor polynomial type TP}. \]

The right half of Listing 1 shows the initial part of the C code of the DO itself. The first five variables \((z^1, z^2, z^0, r^1 \text{ and } v^0)\) represent the Taylor polynomial of the first and second derivative \(z^{(1)}\) and \(z^{(2)}\), the evaluation of \(\varphi^0\) and \(\varphi^1\) at \(z(t_0)\), and the initialization of the return value for the first two Taylor coefficients. Then, \(A, B \text{ and } \Gamma\) are the result of the compositions of \(\alpha, \beta \text{ and } \gamma\) with \(z(t)\). For instance, \(A\) is the Taylor polynomial in \(t_0\) of the composition \(\alpha \circ z(t)\).

The next two auxiliary variables \(\gamma\alpha \text{ and } \beta\alpha\) correspond to an algebraic reformulation of (2). As C does not allow to overload its operators, internally we use a "functional" notation for the operators that act on Taylor polynomials. The context of AD operations is therefore marked with special bracketing.

This construct then just rewrites an expression with operators \(*, /, + \ldots\) into the functional notation. Here in for the special case of the DO, the reformulation is done such that the problem parameters \(A, B \text{ and } \Gamma\) only occur in the divisions \((B/A):\text{ and } (\Gamma/A):\). Thereby, the system can take advantage of specific properties of these arguments and can avoid the division of polynomials, if possible. Special cases are detected at compile time when any of these is a constant function, or if \(B\) is even the 0-function.

Listing 2 shows the user side for the snippet, here in particular the implementation of the Faddeeva function, arbogast\_Faddeeva\_func\_cd. The suffix \_cd stands for complex double precision argument functions.

The left hand side shows the few C code that we have to provide for this implementation. The very first macro that is given at the beginning of Listing 2 provides a type generic interface: _Generic is C11’s new keyword for type based choices. Here it chooses either func-cf or func-cd according to Z’s type. This function is then applied to the same argument Z.

To specify the functions, first we use an implementation of the Faddeeva function from the libcerf library, and look up the derivative of the wofz function. Then, we observe that "functions" \(\alpha\) and \(\gamma\) are actually constant and can be implemented as simple macros. Function \(\beta\), specialized as \(\beta\)-cd, is just \(2\cdot z(t)\).

This specialization are then fed to the import of the snippet code, on the right hand side. This import is identified trough a name (wofz-cd), an additional import of the same snippet with another name, wofz-cf, to implement func-cf is omitted. The five types of the import are chosen to be scalar complex double (scd) for \(\Gamma\alpha\), \(\Gamma\beta\) and \(\Gamma\gamma\) as Taylor polynomial TP for \(T\alpha, T\beta\) and \(T\gamma\) (tcd). Observe, that thereby the division \(\text{div}(B, A)\) is just a division of two scalars and \(\text{div}(\Gamma, A)\) divides Taylor polynomial \(\Gamma\) by a scalar.

We can easily estimate the complexity of the whole. The only parts with quadratic complexity are the products and division for the computation of \(\gamma\alpha\) and \(\beta\alpha\) and the part for equating coefficients which is not shown.

With Modular C’s \textbf{foreach} directive, we can instantiate versions for other types. We just have to surround the above code by \#pragma CMOD foreach \textbf{TYPE} = cf cd and \#pragma CMOD done and do some adjustments to the naming. Then the code is repeated twice. In the first copy, all occurrences of the pattern \$\{\textbf{TYPE}\} will be replaced by cf, in the second by cd.

4 Properties of arbogast

- All DOs are implemented as functions (versus procedures or \textbf{void} functions) returning a Taylor polynomial.

Thereby nested mathematical expressions in user code that are to be differentiated can be reused in their...
• The implementation of the core of the library and all derived functions are written in C. There is no need to program core parts in some language that is closer to the compilers, such as AST, see [9].
• C’s natural assignment works flawlessly for our purpose. There is no need to overload the assignment operator.
• All code is readable and verifiable.
• Code can be re-used by means of the snippet directive:
  – The same code can be instantiated in three different versions. First, it can be a usual mathematical function (taking a double) or an AD function (taking a Taylor polynomial). For the latter, we can choose between the convergence criterion on the value (as for the double function), or a convergence criterion on the Taylor coefficients [10].
  – Internals of the library (e.g. the div DO) are implemented just once for all floating types.
  – User code can re-use predefined DOs such as our example ODE2nd.
• Interfaces can be programmed type generic. Several choices of floating point types can be maintained simultaneously.
• Interfaces can be programmed case generic, that is parameters with special properties (constants versus functions) are detected automatically and the generated code takes advantage of such a special case.
• Code is easily encapsulated through the module directive. Functions that are accessed through a local name (e.g wofz) can easily be replaced by another implementation.
• Each module can have an entry function that acts as its unit test for continuous integration of the code base.
• All DOs have at most quadratic complexity and are implemented efficiently. In particular, we are using fast convolution for the product formula, [11]. When the arbogast C code is compiled with FORTRAN’s relaxed floating point model (gcc option -fcx-fortran-rules) the code is up to 30% faster than the FORTRAN reference code from which we started.

4.1 Conclusions and further work

Conclusions. arbogast provides a highlevel toolbox for the calculus with Taylor polynomials. It is entirely based on a well defined extension of the C programming language, Modular C, and places itself between tools that proceed by operator overloading on one side and by rewriting, on the other. The approach is best described as contextualization or reinterpretation of code, because it permits the programmer to place his code in different types of contexts (usual math, AD, ...) to reinterpret it as usual C function or as a DO. Because of the type generic features of modern C, all specialization (e.g multiplication by a constant) can be delegated to the compiler.

Future work. C as base language does not provide the opportunity for operator overloading. Therefore, at the time of this submission, the use of arbogast on existing code needs to rewrite expressions as function calls. This rewriting completely remains at a syntactical level. Currently, there is no distinction for different types of contexts for rewriting, but we are evaluating the possibility of such a distinction. Also, we will consider this rewriting feature in the general context of Modular C, not only arbogast.

References

A full higher-order AD continuation and bifurcation framework

Isabelle Charpentier* and Bruno Cochelin†

March 22, 2016

1 Introduction

Numerical continuation and bifurcation analysis of nonlinear equation solutions are classical tools in many scientific areas. During the last thirty years, general-purpose software, free and commercial ones, have been proposed to engineers and scientists to draw bifurcation diagrams without embarking into the heavy task of programming their own continuation algorithm. As an alternative to first order predictor-corrector principles described in [1], solution branches may be efficiently approximated as higher-order truncated Taylor expansions for which the range of validity is estimated after the continuation step size and may often be applied without Newton corrections.

Let \( \mathcal{R}(U) \) define a nonlinear algebraic system comprising \( n \) equations,

\[
\mathcal{R}(U) = \mathcal{R}(u, \lambda) = 0,
\]

the unknown \( U \) of which comprises the state vector \( u \) of dimension \( n \) and the scalar control parameter \( \lambda \). The solutions of such an under-determined problem are one dimensional continua, referred to as solution branches, that may intersect at bifurcation points. Theses branches are locally parameterized with the so-called pseudo arc-length defined as

\[
a = \left( \frac{\partial U}{\partial a} \right)^* A(U(a) - U(0)),
\]

where \( A \) is a user-defined diagonal matrix. The default for \( A \) is the identity matrix. The pseudo-arc length continuation algorithm is a classical answer to the calculation of solution branches.

Under analyticity assumptions, the solution branches may be approximated as a collection of continuous parametric truncated Taylor expansions [3, 4],

\[
U(a) = \sum_{p=0}^P \frac{a^p}{p!} \frac{\partial U}{\partial a^p}(0) = \sum_{p=0}^P a^p U_p, \quad \forall a \in (0, a_{max}),
\]

where \( P \) is the truncation order of the series, \( U_p \) is the unknown Taylor coefficient of \( U(0) \) at order \( p \) and \( a_{max} \) is the range of validity for the pseudo-arc-length parameter \( a \). Series [3] are introduced in [1] to yield an equation for the Taylor coefficient at order \( p \),

\[
(\mathcal{R} \circ U)_p = \{ \mathcal{R}_1 \} U_p + \{(\mathcal{R} \circ U)_p U_p = 0 \} = 0,
\]

where the Jacobian \( \{ \mathcal{R}_1 \} \) of \( \mathcal{R} \) with respect to \( U \) is the same over the orders. The right-hand side term \( \{(\mathcal{R} \circ U)_p U_p = 0 \} \) is the Taylor coefficient at order \( p \) of \( (\mathcal{R} \circ U) \) evaluated with a null value for the unknown coefficient \( U_p \). The solution of [1] - [2] relies on the sequence of linear systems

\[
\begin{pmatrix}
\{ \mathcal{R}_1 \} U_p
\end{pmatrix} = \begin{pmatrix} -\{(\mathcal{R} \circ U)_p U_p = 0 \} \delta_{1p} \end{pmatrix}, \quad \text{for } p \geq 1,
\]

where \( \delta_{1p} \) is the Kronecker’s delta, \( U_1^* \) is the transpose of vector \( U_1 \). The range of validity \( a_{max} \) is deduced from the remainder of the series [5] [2],

\[
a_{max} = \left( \frac{\varepsilon}{\| \{(\mathcal{R} \circ U)_p U_p = 0 \} \|} \right)^{1/P},
\]

where the threshold \( \varepsilon \) is typically set to \( 1.10^{-6} \). This iterative sequence of linear systems [4] results in a general high level nonlinear solver, known as Diamant, the HOAD of which is comparable to the HOAD described in other Taylor-based nonlinear solvers (see [4] and the references therein). Thanks to AD generality, this solver is able to deal with any user-defined nonlinear analytical residual problem satisfying [1].

To date, Diamant does not provide a higher-order bifurcation analysis. This significant shortcoming is now addressed by building on the bifurcation strategies presented in [5] and [6] to propose a full higher-order AD implementation of bifurcation detection, location and branch switching with a high trade-off between accuracy and efficiency.
2 Bifurcation analysis

Let $U^b$ be a simple bifurcation point on the solution branches $B^1$ and $B^2$. Fig. 1a, the tangents of which are denoted by $U_1^{B^1}$ and $U_1^{B^2}$. At point $U^b$, the Jacobian $\{R_i^b\}$ admits two null right vectors $\phi^1$ and $\phi^2$, and a left null vector $\psi$ such that

$$\{R_i^b\}\phi^i = 0, \quad i = 1, 2, \quad \text{and} \quad \psi^*\{R_i^b\} = 0,$$

respectively. The null space and its orthogonal complement are denoted by $\mathcal{N}(R_1^b) = \text{span}\{\phi^1, \phi^2\}$ and $\mathcal{N}^\perp(R_1^b) = R^b \setminus \mathcal{N}(R_1^b)$. Vectors $\phi^1$, $\phi^2$ and $\psi$ allow to write the so-called Algebraic Bifurcation Equation (ABE) as

$$c_{11}\alpha^2 + 2c_{12}\alpha\beta + c_{22}\beta^2 = 0, \quad \text{with} \quad c_{ij} = \psi^*\{R_i^b\}\phi^j \quad \text{for} \quad i = 1, 2,$$

where the unknowns $(\alpha, \beta)$ satisfy $\{R_i^b\}(\alpha\phi^1 + \beta\phi^2) = 0$.

As demonstrated in [6], close to a bifurcation point, a geometric series emerges between successive Taylor coefficients. This particular event may be used for detection and branch switching. Four stages are necessary. First, the Taylor coefficient sequence is monitored to exhibit a possible geometric series. Second, the bifurcation locus is determined and the geometric series is taken out to restore a clean series with an optimal range of validity along the traveled branch. Third, the tangent to the second branch is computed. Fourth, at bifurcation point, optimal ranges of validity are computed for both directions.

Evidence for the geometric series — Choosing $\phi^1 = U_1^{B^1}$ and $\phi^2 = U_1^{B^2}$ allows to write the first order representation of branches $B^1$ and $B^2$ as

$$U(\alpha, \beta) = U_b + aU_1^{B^1} + \beta U_1^{B^2}.$$

In the perfect case, Fig. 1b, the ABE simplifies into $\alpha\beta = 0$, and either $\alpha$ or $\beta$ is null. This agrees with the particular choices done for $\phi^1$ and $\phi^2$: the approximate solution $U(\alpha, \beta)$ theoretically lies on one or the other branch. However, residual errors are not zero from a computer point of view. The ABE becomes $\alpha\beta = \mu$ where $\mu$ is a small real number, Fig. 1b. Using $\alpha = a - s$ and $\beta = \mu/(a - s)$ in (9) allows to write

$$U(a, s) = U_b - sU_1^{B^1} - \left(\frac{\mu}{s}\right)U_1^{B^2} + aU_1^{B^1} + \left(\frac{\mu}{s}\right)\left(\frac{a}{a - s}\right)U_1^{B^2},$$

where the shift $s$ represents the distance from $U(0, s)$ to $U_b$. Equation (10) provides a first order representation of the perturbed branch $B^1$ evaluated at a point $U_0$ close to the simple bifurcation under study. Since $\mu/s$ is very small, the contribution of $U_1^{B^2}$ is negligible except when $a$ is quite equal to $s$. In that case, the rational fraction $a/(a - s)$ may be written as the geometric series $\sum_k (a/s)^k$ with common ratio $a/s$. This happens more freely for a small distance $s$ and/or for a large residual error contribution $\mu$ at point $U^0$. The interested reader is referred to [7] for the analysis of functional singularities through Taylor series.

Bifurcation detection — The detection of a geometric series in the solution of (3) is based on collinearity and proportionality properties of the last four Taylor coefficients of $U$ computed at a some point $U_0$. The candidates as common ratio and scale factor are

$$\alpha = \frac{U_p^* U_p}{U_{p-1}^p U_p},$$

Figure 1: (a): Perfect case, (b): Numerical case, (c): Elastica’s bifurcation diagram, (d): RRR workspace boundaries.
and the last vector of the series $U_P$, respectively. A geometric series is detected when the following linearity and proportionality properties are satisfied,

$$\sqrt{\sum_{p=1}^{3} \left(1 - \frac{U_{p-p}^TU_P}{\|U_{p-p}\|^2 \|U_P\|^2}\right)^2} \leq \epsilon_c = 10^{-4} \quad \text{and} \quad \sum_{p=1}^{3} \left|\frac{(U_{p-p} - \alpha U_{p-p+1})^*U_{p-p}}{U_{p-p}^TU_{p-p}}\right| \leq \epsilon_p = 10^{-3}. \quad (12)$$

**Cleaning of the series** — The geometric series detected in the higher-order terms is taken off to avoid $(i)$ interactions between the two solution branches, $(ii)$ a possible amplification of computational errors, $(iii)$ small ranges of validity, or even $(iv)$ an undesired branch switching. Fig. 1b. Taylor coefficients of the clean series $\hat{U}$ are

$$\hat{U}_p = U_p - \alpha^{P-p}U_P, \quad \forall p = 0, ..., P - 1. \quad (13)$$

**Bifurcation location** — The clean series $\hat{U}(a) = \sum_{p=0}^{P-1} a^p \hat{U}_p$ provides the perfect branch of solutions in the current tangent direction. The range of validity of $\hat{U}$ may be written using $[9]$ at order $P - 1$. By construction, the bifurcation point satisfies $\hat{U}^b = \hat{U}(a)$, and an approximation of the tangent to the traveled branch at that point is $U_1^{B^2} = \frac{\partial \hat{U}}{\partial a} (a = \alpha)$. 

**Second tangent** — Differentiating $R(U(a)) = 0$ with respect to $a$ results in $R(U(0))U_1 = 0$ along the branch of solutions. This signifies that the Taylor coefficient $U_1$ of $U$ is tangent to the branch at point $U(0)$. At the bifurcation point $U^b$, the Taylor coefficient $U_1^{B^2}$ thus belongs to the kernel $\mathcal{N}(\{R_1^b\})$.

Differentiating twice $R(U(a)) = 0$ at point $U^b$ yields

$$\{R_1^b\}U_2^{B^2} + \{R_2^b\}(U_1^{B^2})^2 = 0, \quad (14)$$

where $\{R_1^b\}$ is the Hessian of $R$ evaluated at point $U^b$. This equation is multiplied $[8]$ by the left null vector $\psi$ to deduce the ABE

$$\psi^* \{R_2^b\}(U_1^{B^2})^2 = 0. \quad (15)$$

Taking $U_1^{B^2} = \alpha U_1^b + \beta \phi^2$ with $\phi^2$ belonging to $\mathcal{N}(\{R_1^b\})$ allows to compute $U_1^{B^2}$ from $(15)$.

**Ranges of validity** — The series at point $U^b$ may be written as

$$U(a) = U^b + aU_1^b + \sum_{p=2}^{P} a^p U^b_p, \quad (16)$$

where $U^b_p$ is one of the two tangents and $U^b_p$ are its unknown Taylor coefficients, for $p = 2, ..., P$. This series is introduced in $(1)$. Resulting equations are projected onto $\mathcal{N}(\{R_1^b\})$ and $\mathcal{N}^\perp(\{R_1^b\})$ to get the iterative sequence of linear systems satisfied by the Taylor coefficients $U^b_p$ ($p \geq 2$). The general formula comprises the projection $(17)$ onto $\mathcal{N}^\perp(\{R_1^b\})$:

$$\pi_\psi \left(\mathcal{R}_1^b U^b_p + \{R^b_p\}_{U^b_p = 0}\right) = 0, \quad (17)$$

where $\mathcal{R}_P^b$ is the Taylor coefficient of $\mathcal{R}$ at order $p$ computed at the bifurcation point, the projection $(15)$ onto $\mathcal{N}(\{R_1^b\})$

$$\psi^* \left(\{R^b_{p+1}\}_{U^b_p = 1, U^b_{p+1} = 0} U^b_p + \{R^b_{p+1}\}_{U^b_p = U^b_{p+1} = 0}\right) = 0, \quad (18)$$

the path equation contribution $[19]$

$$(U^b_1)^* U^b = 0. \quad (19)$$

The solution of $(17)$–$(19)$ is managed assuming

$$U^b_p = \alpha_p U_1^{B^1} + \beta_p U_1^{B^2} + \Upsilon_p^*, \quad (20)$$

where the unknowns $\alpha_p$ and $\beta_p$ are real parameters and $\Upsilon_p^* \in \mathcal{N}^\perp(\{R_1^b\})$. After some calculations to determine $\alpha_p$, $\beta_p$ and $\Upsilon_p^*$, one may deduce the series for $U^b$. Ranges of validity for the series are then deduced from the Taylor coefficient $U^b_P$ following $[6]$. 

52
3 Examples

For reproducibility purposes, we consider a clamped-clamped inextensible elastic beam subject to a uniaxial compression \([3,6]\) and the workspace boundaries of a RRR mechanism (3 bars, 3 revolute joints) \([10,11]\) since their bifurcation diagrams contain many interconnected solution branches, Figs. 1c and 1d, respectively.

**Beam** – Let \(s\) be the dimensionless curvilinear coordinate \(s \in (0, 1)\). The beam equilibriums satisfy

\[
x'(s) = \cos(\theta(s)), \quad y'(s) = \sin(\theta(s)), \quad \theta'(s) = m(s), \quad m'(s) = -4\pi^2(P\sin(\theta(s)) + T\cos(\theta(s))),
\]

where \((x(s), y(s))\) is the position of the beam centerline, \(\theta(s)\) the rotation of the beam cross-section and \(m(s)\) the bending momentum. Boundary conditions are \(x(0) = y(0) = \theta(0) = 0\) and \(m(0) = 2\pi C\) on the one side, and \(y(1) = \theta(1) = 0\) on the other. Variable \(P\) is the compressive load while \(T\) and \(C\) are the lateral and torque reactions at the boundary. The ODE system is discretized using an orthogonal collocation method with cubic interpolation.

**RRR mechanism** – Let \((\theta_1, \theta_2, \theta_3) \in [-\pi/3, \pi/3]^3\) be the angular positions between bars of length \(l_1 = 4\, cm\), \(l_2 = 2\, cm\) and \(l_3 = 1\, cm\). The behavior of the robot and the reachable positions are modeled following

\[
\begin{align*}
  x - (l_1\cos(\theta_1) + l_2\cos(\theta_1 + \theta_2) + l_3\cos(\theta_1 + \theta_2 + \theta_3)) &= 0, \\
  y - (l_1\sin(\theta_1) + l_2\sin(\theta_1 + \theta_2) + l_3\sin(\theta_1 + \theta_2 + \theta_3)) &= 0.
\end{align*}
\]

Workspace boundaries are singularities of some Jacobian of \([22]\) that may be followed by continuation \([10,11]\).

4 Conclusions and further work

The talk will discuss theoretical and implementation aspects governing this “full AD” general higher-order continuation and bifurcation framework, emphasizing both generality and efficiency through non-trivial case studies. The last version of the Diamanlab \([12]\) software may be downloaded [here](http://hal.archives-ouvertes.fr/hal-00853599/PDF/diamanlab_halb.pdf).

References


Differentiating through Conjugate Gradient

Bruce Christianson*

It is generally supposed to be problematic to differentiate naively through singularities\(^1\). We show that, although the Conjugate Gradient (CG) Algorithm \(^2\) has a singularity at the solution, it is straightforward to differentiate through this simply by re-declaring all the variables as truncated Taylor series, the type of active variable widely used in ADOL-C and elsewhere.

Suppose that we wish to use CG to solve the linear equations \(Ax = b\) for \(x\), and at the same time to compute the directional derivative \(\dot{x}\) satisfying

\[
A\dot{x} + \dot{A}x = \dot{b}
\]

The trick is to use a conventional implementation of CG, but to re-declare \(A, b, x, g, p, \alpha, \beta, \) etc. as truncated Taylor series: so for example

\[
x_i = x_i^{(0)} + x_i^{(1)}t + x_i^{(2)}t^2 + \ldots + x_i^{(k)}t^k
\]

and \(b^{(1)} = \dot{b}\) etc. As usual we assume \(A^{(0)}\) to be symmetric and positive definite.

The CG algorithm proceeds from an initial approximation \(x_0\): this, like all the other variables in the algorithm that follows, is a truncated Taylor series.

**Conjugate Gradient Algorithm**

start: set \(i := 0\)

\[
g_0 = Ax_0 - b
\]

\[
p_1 = -g_0
\]

loop: \(i := i + 1\)

choose \(\alpha_i\) to solve \(\alpha_i(p_i^TAp_i) = g_i^2\)

\[
x_i = x_{i-1} + \alpha_ip_i
\]

\[
g_i = Ax_i - b = g_{i-1} + \alpha_iAp_i
\]

if \(\|g_i\| < \epsilon\|g_0\|\) then terminate

choose \(\beta_i\) to solve \(\beta_i g_{i-1}^2 = g_i^2\)

\[
p_{i+1} = \beta_ip_i - g_i
\]

go to loop

---

\*University of Hertfordshire, Hatfield, England; b.christianson@herts.ac.uk

\(^1\)For the case of the Euclidean norm at the origin see [1] p. 357 Table 14.9 and p. 363 Exercise 14.1.
Theorem  At each stage, \( g_i \cdot g_j = 0 \) and \( p_i^T A p_j = 0 \) for all \( j : j < i \).

Proof  By the usual induction, applied to Taylor series. \( \square \)

Corollary  In the CG algorithm we may alternatively take \( \beta_i \) to be the solution of
\[
\beta_i p_i^T A p_i = g_i^T A p_i
\]

Proof  Writing \( P \) for \( p_i^T A p_i \) we have
\[
P \beta_i g_i^2 = (g_i^T A p_i) g_i^2 = g_i^T A p_i \alpha_i P = g_i^T (g_i - g_{i-1}) P = g_i^2 P
\]
\( \square \)

An Aside on Vector Norms  Whenever we apply tangent mode AD to an iterative algorithm, we must be careful in the stopping condition to use a vector norm that is appropriate for vectors whose elements are truncated Taylor series. In particular \( \|g\| \) must be non-zero and well-conditioned in the case when \( g^{(0)} \) is very close to 0 but \( g^{(1)} \) is not. As mentioned earlier, for the Euclidean norm we need to be careful, and we risk numerically unstable behaviour if we first use AD to calculate \( r = \sqrt{\bar{g} \cdot g} \) as a Taylor series, and then evaluate \( \|r\| \). The first step gives
\[
r = \|g^{(0)}\| + (\bar{g}^{(0)} \cdot g^{(1)}) t \quad \text{where} \quad \bar{g}^{(0)} = \frac{g^{(0)}}{\|g^{(0)}\|}
\]
and the term in \( t \), which results from the cross-terms in the inner product \( g \cdot g \), may take any value between 0 and \( \|g^{(1)}\| \): the moral is that we really do need to calculate the vector norm directly.

A Sobolev-style \( p \)-norm for \( 1 \leq p \leq \infty \) works well:
\[
\|g\|_{k,p} = \|g^{(0)}\|_p + \|g^{(1)}\|_p + \ldots + \|g^{(k)}\|_p
\]
and it doesn’t matter whether we take the absolute sum (or supremum) component-wise first or Taylor-order first, even for \( p = 2 \). Nothing in this aside is specific to CG.

In what follows we shall assume for the moment that we are interested only in first derivatives and, rather arbitrarily, take \( \|g\| = \|g^{(0)}\| + \|g^{(1)}\| \): although this is not strictly speaking a norm when \( k > 1 \), we shall see that it still does nicely as a stopping criterion.

Termination of Truncated Taylor CG  We shall show that, if arithmetic is exact, then the truncated Taylor CG algorithm terminates for some \( i = i_1 \), with \( g_i^{(0)} = g_i^{(1)} = 0 \).
Lemma If \( g_i^{(0)} = g_i^{(1)} = 0 \) then \( x_i^{(0)} \) is the solution of \( A^{(0)} x^{(0)} = b^{(0)} \) and \( x_i^{(1)} \) is the correct directional derivative \( \dot{x} \) of \( x \) for the given directional derivatives \( \dot{A} = A^{(1)}, \dot{b} = b^{(1)} \).

Proof Since at each stage \( g_i \) is the residual \( Ax_i - b \) we have
\[
A^{(0)} x_i^{(0)} = b^{(0)} \quad A^{(1)} x_i^{(0)} + A^{(0)} x_i^{(1)} = b^{(1)}
\]
as required.

We already know that the CG algorithm produces (if arithmetic is exact) an orthogonal sequence \( g_i^{(0)} \) and so, since there is only a finite number of dimensions, for some \( i_0 \) we shall have \( g_i^{(0)} = 0 \).

Lemma \( g_i^{(1)} \perp g_j^{(0)} \) for all \( j < i_0 \)

Proof The fact that \( g_i^{(0)} \cdot g_j^{(0)} = 0 \) for \( j < i_0 \) implies that the first order terms satisfy
\[
g_i^{(1)} \cdot g_j^{(0)} + g_i^{(0)} \cdot g_j^{(1)} = 0
\]
But \( g_i^{(0)} = 0 \) whence the result.

If the \( g_j^{(0)} : j < i_0 \) form a spanning set, then \( g_i^{(1)} = 0 \) by the first Lemma. But this will not be the case in general: for example \( A^{(0)} \) may have repeated eigenvalues, or \( x_i^{(0)} \) may have been a lucky guess.

However truncated Taylor CG does not terminate simply because \( g_i^{(0)} = 0 \). So suppose \( g_i^{(1)} \neq 0 \) and let us see what happens next: \( \beta_i^{(0)} = g_i^{(1)} / g_i^{(0)^2}, \) so \( \beta_i^{(0)} = \beta_i^{(1)} = 0, \) and then
\[
p_i^{(0)} = 0 \quad p_i^{(1)} = -g_i^{(1)}
\]
so \( p_i^{(1)} \) is non-zero. Now the CG calculation for \( \alpha_i^{(0)} + 1 \) gives
\[
\alpha_i^{(0)} = -\frac{\left(g_i^{(1)}\right)^2}{p_i^{(1)} A^{(0)} p_i^{(1)}} \neq 0
\]
so effectively we have a CG restart for the calculation of \( g_i^{(1)} \). The algorithm then continues for several more iterations, with \( g_i^{(0)} = p_i^{(0)} = 0 \). Note that the calculations for \( \alpha_i, \beta_i \) for \( i > i_0 \) involve taking the quotient of two truncated Taylor series with leading non-zero coefficients in the term \( t^2 \), which is why we require \( k \geq 2 \) even for first order directional derivatives.

Lemma Taking the sequence \( g_i^{(0)} \) until \( i = i_0 \) and then switching to \( g_i^{(1)} \) gives an orthogonal sequence.
Proof Since for $i > j$ we have $g_i \perp g_j$ it follows that the first and second order terms of the corresponding inner product vanish:

$$g_i^{(0)} \cdot g_j^{(1)} + g_i^{(1)} \cdot g_j^{(0)} = 0$$
$$g_i^{(0)} \cdot g_j^{(2)} + g_i^{(1)} \cdot g_j^{(1)} + g_i^{(2)} \cdot g_j^{(0)} = 0$$

so for $i \geq i_0 > j$ we have $g_i^{(0)} = 0$ whence $g_i^{(1)} \perp g_j^{(0)}$ from the first equation, whereas for $i > j \geq i_0$ we have $g_i^{(0)} = g_j^{(0)} = 0$ whence $g_i^{(1)} \perp g_j^{(1)}$ from the second equation.

\[\square\]

Theorem Eventually we get termination with the correct derivative.

Proof Eventually for some $i_1$ we have $g_i^{(1)} = 0$ by orthogonality, at which point $x_i^{(1)}$ is the correct directional derivative.

\[\square\]

The arithmetic used in AD is usually not exact, and so we must decide at which point to treat $g_i^{(0)}$ as zero: for example when calculating the truncated Taylor series $\beta_i = g_i^2 / g_i^{2-1}$ when do we take $\beta_i^{(0)} = (g_i^2)^{(0)} / (g_i^{2-1})^{(0)}$, and when do we use l’Hospital’s rule to set $\beta_i^{(0)} = (g_i^2)^{(2)} / (g_i^{2-1})^{(2)}$?

This is the same problem we face with conventional CG in deciding when to do a restart, and we may use the same heuristics to decide when to do this, for example when $g_i^{(0)}$ becomes small relative to, or loses orthogonality with $g_0^{(0)}$. Similarly we may decide to do a Taylor CG restart when $g_i^{(1)}$ becomes small relative to $g_0^{(1)}$ or loses orthogonality with $g_0^{(0)}$. More details and results will be included in the full paper.

Conclusion The interesting feature of this approach, beside conceptual and programming simplicity, is that both the solution and its directional derivative are obtained together in at most $n$ iterations, where $n$ is the dimension of $x$. Furthermore, as we shall see in the full version of the paper, the approach can readily be generalized to include simultaneous calculation of higher order derivatives than the first, without increasing this maximum iteration count.

References


A Method for the Minimization of Piecewise Smooth Objective Functions

Sabrina Fiege †, Andrea Walther ‡ and Andreas Griewank †

January 2016

1 Motivation

Nonsmoothness is a typical characteristic of numerous objective functions from both academic and real world applications. The nondifferentiabilities are often caused by the absolute value function. Therefore it is reasonable to consider piecewise smooth (PS) function in the sense of Scholtes [1, Chap. 4]. A well known example is the reformulation of a constrained optimization problem by adding $\ell_1$- and $\ell_\infty$-penalty terms of constraint violations to the original function.

There are still only few practical methods available for the minimization of Lipschitzian PS functions $f : \mathbb{R}^n \mapsto \mathbb{R}$. A common approach is to hope that the nondifferentiabilities do not effect standard algorithms designed for smooth problems too much. In fact as demonstrated by Lewis and Overton BFGS does amazingly well and can be complemented by a stochastic gradient sampling, see [2]. Another class of widely used methods are the bundle methods, see, e.g. [3]. But the behaviour of these approaches seems to be rather erratic. Because of these observations one may anticipate an improvement by gaining additional information via the exploitation of the structure caused by the nondifferentiabilities.

Our goal is the development of a minimization method that minimizes Lipschitz continuous, PS objective functions by successively generating piecewise linear local models and to efficiently solve these models by exploiting the polyhedral structure representing the nondifferentiabilities. The basis of our work is a method proposed by Hiriart-Urruty and Lemaréchal in [4, Chap. 8] that was regarded to be not implementable. By using an adapted version of the algorithmic differentiation tool ADOL-C, we are now able to provide all required information for our minimization method.

In the following the required components are briefly presented and their corresponding AD drivers are introduced. In the main part the algorithm, which consists of an outer and an inner routine, will be explained.

2 Notation and background

We will consider only objective functions $f : \mathbb{R}^n \mapsto \mathbb{R}$ that can be given by a sequence of elementary functions that are either Lipschitz continuously differentiable in an open domain $D \subset \mathbb{R}^n$ or the absolute value function. The correspondingly adapted evaluation procedure is given in Tab. 1. The number of absolute values occurring during the function evaluation will be denoted by $s \in \mathbb{N}$. The arguments $z_i$ of the absolute value functions cause the switches in the corresponding derivative values and therefore, $z = (z_i) \in \mathbb{R}^s$ is called switching vector. It also defines the signature vector $\sigma = (\sigma_i) \in \{-1,0,1\}^s$.

Selection functions and directional active gradients

The class of PS functions can be represented as

$$f(x) = \{f_\sigma(x) : \sigma \in \mathcal{E} \subset \{-1,0,1\}^s\} \quad \text{at} \quad x \in \mathbb{R}^n$$

as defined in [1, Chap. 4] where the selection functions $f_\sigma$ are continuously differentiable on neighborhoods of points where they are active. We will assume that all $f_\sigma$ with $\sigma \in \mathcal{E}$ are essential in that their coincidence sets $\{f(x) = f_\sigma(x)\}$ are the closures of their interiors. Furthermore, one has that

$$\partial f(x) \equiv \text{conv}(\partial L f(x)) \quad \text{with} \quad \partial L f(x) \equiv \{\nabla f_\sigma(x) : f_\sigma(x) = f(x)\},$$

where the elements of $\partial L f(x)$ are called limiting gradients of $f$ at $x$.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
$v_{i-n}$ & $x_i$ \\
\hline
$z_i$ & $\psi_i(v_j)_{j<i}$ \\
$\sigma_i$ & $\text{sign}(z_i)$ \\
$v_i$ & $\sigma_i z_i = \text{abs}(z_i)$ \\
$y$ & $\psi_s(v_j)_{j<s}$ \\
\hline
\end{tabular}
\caption{Adapted evaluation procedure}
\end{table}

---

*Paderborn University, Germany, sfiege@math.upb.de
†Paderborn University, Germany
‡Yachaytech, Urcuqui, Ecuador
A directionally active gradient \( g \) is given by
\[
g \equiv g(x; d) \in \partial^2 f(x) \quad \text{such that} \quad f'(x; d) = g^T d \quad (1)
\]
and \( g(x; d) \) equals the gradient \( \nabla f_x(x) \) of a locally differentiable selection function \( f_x \) that coincides with \( f \) on a set, whose tangent cone at \( x \) contains \( d \) and has a nonempty interior. Further information on directionally active gradient can be found in [5]. If the function evaluation is coded in C or C++, the driver
\[
directional_active_gradient(tag, n, x, d, g, sigma,g);
\]
of ADOL-C can be used to compute \( g(x; d) \) at a given argument \( x \) and a given direction \( d \).

### Piecewise linearization and abs-normal form

To obtain a piecewise linearization (PL) of the target function \( f \), one has to construct for each elemental function a tangent approximation as described in [6, Chap. 3]. For a given argument \( x \) and a direction \( \Delta x \), we will use the elemental linearizations that are already known from standard AD theory, extended by the following rule for \( \text{abs}() \) as a tangent approximation as described in [6, Chap. 3]. For a given argument \( x \) and a direction \( \Delta x \), we will use the elemental linearizations that are already known from standard AD theory, extended by the following rule for \( \text{abs}() \) as a tangent approximation as described in [6, Chap. 3].

For any PL, it follows by continuity that
\[
\Delta v_i = \text{abs}(v_j + \Delta v_j) - v_i \quad \text{for} \quad v_i = \text{abs}(v_j).
\]

This extension of the usual AD approach can be used to compute the increment \( \Delta f(x; \Delta x) \) and therefore the PL
\[
f_{PL,x}(\Delta x) = f(x) + \Delta f(x; \Delta x) \quad (3)
\]
of the original PS function \( f \) at a given point \( x \) with the argument \( \Delta x \). To exploit the nondifferentiabilities of the PL in the minimization algorithm, see Sec. 3, we rewrite the PL in abs-normal form. In [5] it was shown that any piecewise linear function \( y = f_{PL}(x) \) can be expressed using the argument \( x \) and the resulting switching vector \( z \in \mathbb{R}^s \) in the abs-normal form given by
\[
\begin{bmatrix}
z \\
y
\end{bmatrix} = \begin{bmatrix} c_x \\ c_y \\ Z \\ L \\ Y \\ J \end{bmatrix} \begin{bmatrix} x \\ \left| z \right| \\ \left| z \right|^2 \\ \left| z \right|^3 \end{bmatrix}, \quad (4)
\]
where \( c_x \in \mathbb{R}^s, c_y \in \mathbb{R}^m, Z \in \mathbb{R}^{s \times n}, L \in \mathbb{R}^{s \times s}, Y \in \mathbb{R}^{m \times n} \) and \( J \in \mathbb{R}^{m \times s} \). The matrices \( Y \) and \( J \) are row vectors in this optimization context, since we consider functions with \( m = 1 \). The matrix \( L \) is strictly lower triangular, i.e., each \( z_i \) is an affine function of absolute values \( |z_j| \) with \( j < i \) and the input values \( x_k \) for \( 1 \leq k \leq n \).

To generate the corresponding PL, one can use the AD-tool ADOL-C [7]. Therefore, several drivers were included into ADOL-C. One can extract the number of absolute value function evaluations using the call
\[
s = \text{get_num_switches}(tag);
\]
where \( s \) is defined in Tab. 1. This yields also the dimension of the vector \( z \). The function value \( y \) and the switching vector \( z \) according to Tab. 1 are provided by the call
\[
zos_pl_forward(tag,1,n,1,x,y,z);
\]
with the argument \( x \) as point at which the PL is performed. Subsequently, one can use the driver
\[
fos_pl_forward(tag, 1, n, x, deltax, y, deltay, z, deltaz);
\]
to actually compute the increment \( \Delta y = \Delta f(x; \Delta x) \). As output variables, one has also values \( z \) of the switching variables and their linearization \( \Delta z \). ADOL-C also provides the driver
\[
abs_normal(tag, m, n, s, x, sigma, y, z, cz, cy, Y, J, Z, L);
\]
to compute the components of the abs-normal form for a given PS function \( f \) and a given argument \( x \). It is important to note, that the parts \( c_z, c_y, Y, J, Z \), and \( L \) only depend on the considered PS function \( f \) and its argument \( x \).

### Structure of the decomposed domain

The structure of the PL is given by its decomposition into polyhedra \( P_\sigma \) and is caused by the nondifferentiable points. For any PL, it follows by continuity that \( P_\sigma \) must be open but possibly empty if \( \sigma = \text{definite} \) in that all its components are nonzero. Degenerated cases are possible as described in [8]. We certainly have by definition of \( \sigma = \sigma(x) \) for the closure of \( P_\sigma \)
\[
P_\sigma \subset \{ x \in \mathbb{R}^n : \Delta f(x) = f_\sigma(x) \},
\]
where identity must hold in the convex case. In the nonconvex case \( f_\sigma \) may coincidentally be active, i.e., coincide with \( \Delta f \) at points in other polyhedra \( P_\sigma \). In particular \( f_\sigma \) is essentially active in the sense of Scholtes [1, Chap. 4] at all points in \( P_\sigma \) provided \( \sigma \) is open. Whether or not it is essentially active somewhere outside of \( P_\sigma \) is irrelevant. To conform with the general concepts of PS functions we may restrict \( f_\sigma \) to some open neighborhood of \( P_\sigma \) such that it cannot be essentially active outside \( P_\sigma \). The corresponding signature vectors are called essential and are given by
\[
\mathcal{E} = \{ \sigma \in \{-1,0,1\}^s : \emptyset \neq P_\sigma \text{ open} \}.
\]

**Example 2.1.** We consider the piecewise smooth and nonconvex function
\[
f : \mathbb{R}^n \rightarrow \mathbb{R}, \quad f(x_1, x_2) = (x_2^2 - (x_1)_+) \quad \text{with} \quad y_+ \equiv \max(0,y).
\]
(5)
Its piecewise linearization evaluated at a base point \( \bar{x} \) and with the argument \( \Delta x \) is given by

\[
f_{\text{PL},x}(\Delta x) = \frac{1}{2} \left( \bar{x}_1^2 + 2\bar{x}_2 \Delta x_2 - \frac{1}{2} (\bar{x}_1 + \Delta x_1 + |z_1|) + \frac{1}{2} |z_2| \right) \tag{6}
\]

with \( z_1 = |\bar{x}_1 + \Delta x_1| \) and \( z_2 = \bar{x}_2^2 + 2\bar{x}_2 \Delta x_2 - \frac{1}{2} (\bar{x}_1 + \Delta x_1 + |z_1|) \).

The domain of the piecewise linearization (6) is decomposed by two absolute value functions into four open polyhedra.

Generally, we will describe the polyhedral structure primarily in terms of the signature vectors \( \sigma \), as shown in the example above. Further properties of the signature vectors and the polyhedral structure are explained in [9].

3 Minimization of Lipschitz continuous, piecewise smooth functions

Now the minimization method is described. It consists of an outer loop that successively generates piecewise linear local models that are complemented by a quadratic penalty term to ensure lower boundedness of the local model. These local quadratic models are minimized by the inner loop.

Minimization algorithm for piecewise smooth functions

As sketched already in [5], we propose the following algorithm to minimize Lipschitzian piecewise smooth functions:

**Algorithm 3.1 (LiPsMin).**

\( \text{LiPsMin} (x^0, q^0) \) // Precondition: \( x \in \mathbb{R}^n, x^0 = x, q^0 > 0 \)

for \( k = 0, 1, 2, \ldots \)

1. Generate a PL model \( f_{\text{PL},x}(.) \) at the current iterate \( x_k \).
2. Use PLMinDescent\((x_k, \Delta x_k, q_k)\) to compute \( \Delta x_k \), s.t. \( f(x_k + \Delta x_k) \leq f(x_k) \).
3. Set \( x^{k+1} = x_k + \Delta x_k \).
4. Compute

   \[
   q^{k+1} := q_k (x_k, \Delta x_k) \equiv \frac{|f(x^{k+1}) - f_{\text{PL},x_k}(\Delta x_k)|}{\|\Delta x_k\|^2}
   \]

   and set \( q^{k+1} = \max\{q^{k+1}, 0.9q_k + 0.1q^{k+1}, q^0\} \).

Return \( x^{k+1} \)

As can be seen, one main ingredient of the approach is the successive piecewise linearization and the solution of the local model by Algo. 3.2 as detailed in Sec. 3. So far, there is no termination criterion given such that the algorithm generates an infinite sequence of iterates \( \{x^k\} \) that can be exploited in the convergence analysis. A possible termination criterion is to check if \( \|\Delta x_k\| = 0 \) after step 2.

To prove that Algo. 3.1 converges, we assume that the PS function \( f \) has a bounded level set with \( x^0 \) the starting point. \( \{q^k\} \) is a bounded sequence, the sequences \( \{\Delta x^k\} \) and \( \{q^k\} \) are uniformly bounded and the routine PLMinDescent, explained in Sec. 3, terminates after finitely many iterations. With these assumptions and the simple version of the algorithm LiPsMin, one obtains that all cluster points \( x^* \) of the infinite sequence \( \{x^k\}_{k \in \mathbb{N}} \) generated by Algo. 3.1 satisfy the first order minimality condition \( f'(x^*, \cdot) \geq 0 \) for Lipschitzian PS problems.

Minimization algorithm for piecewise linear functions

The central part of the inner loop of the minimization method is Algo. 3.2. Its essential difference to the true descent algorithm introduced in [9, Algo. 4] is the solution of a quadratic subproblem instead of computing a critical step multiplier for a given direction. We use the base point \( x^k \) and the step \( \Delta x^k \) as in Algo. 3.1 for clarity. Besides the base point \( x^k \) and the quadratic coefficient \( q^k \) serve as input variables. The increment \( \Delta x^k \) is the output parameter.

By solving the quadratic subproblem of step 1

\[
\min_{\Delta x \in \mathbb{R}^n} f_{\sigma}(\Delta x_j) + \frac{q}{2} \|\Delta x_j\|^2, \quad \text{s.t.} \quad z_i(x^k) + \nabla z_i(x^k)^T \Delta x_j \begin{cases} 
\leq 0 & \text{if } \sigma_i^j < 0 \\
g \geq 0 & \text{if } \sigma_i^j > 0 
\end{cases} \text{ for } i = 1, \ldots, s \tag{7}
\]
for a fixed $\sigma^j$ where $z_i(x^k)$ is the $i$-th component of the switching variable of $z(x^k)$ and $\nabla z_i(x^k)$ the corresponding gradient, one can easily characterize the points $x$ in the extended closure $\tilde{P}_{\tau,j}$ as the solutions of the system of inequalities. All required components are available from the abs-normal form.

Algorithm 3.2 (PLMin).
PLMin$(x^k, \Delta x^k, q^k)$ // Precondition: $x^k, \Delta x^k \in \mathbb{R}^n$, $q^k \geq 0$. Set $\Delta x_0^k = 0$
For $j = 0, 1, 2, ...$
1. Determine solution $\Delta x_j$ of local QP on current polyhedron $P_{\sigma_j}$.
2. Compute bundle $G$ and direction $d$ by computeDesDir().
3. Identify new polyhedron $P_{\sigma_{j+1}}$ by direction $d$.
4. If $||d|| = 0$ STOP.
5. Update $\Delta x_{j+1} = \Delta x_j + \Delta x_j$.
return $\Delta x = \Delta x_j$

The direction $d$ and the bundle $G$ required in step 2 are computed similarly as proposed in [9, Algo. 2], where also the finite termination of this algorithm is shown. However, for the general case considered here, we only want to identify a polyhedron $P_{\sigma_{j+1}}$ that provides descent compared to the current polyhedron. Hence, we introduce here the additional multiplier $\beta \in (0, 1)$ to relax the descent condition compared to [9, Algo. 2].

However, if one recalls the requirements made in step 2 of Algo. 3.1, one finds that it is not sufficient to compute a stationary point of the quadratic subproblem. One also has to ensure that the value of the original target function does not increase. One has to distinguish two possibilities. First if there are one or more iterations of PLMin that achieve descent, one can fall back onto the last iterate that fullfilled the descent condition $f(x^k + \Delta x_{j+1}^k) < f(x^k + \Delta x_j^k)$ and stop. Second, if no descent is generated in the first iteration of PLMin, i.e. $f(x^k) < f(x^k + \Delta x_1^k)$, there is no possibility to fall back onto the last valid iterate. Therefore, we need an adjusted PLMin algorithm, that assures the required nonincrease. First, the termination criterion of Algo. 3.2 has to be extended hence detecting an increase of the function value is possible. Then we wrap Algo. 3.2 by another routine called subsequently PLMin_Descent that checks, if $f(x^k + \Delta x^k) < f(x^k)$ is valid. If this is not the case it increases $q^k$ by a multiplier $\gamma > 1$ and calls Algo. 3.2 again with the original piecewise linear model but the modified quadratic coefficient. One can show after finitely many increases of $q^k$ the condition $f(x^k) > f(x^k + \Delta x_1^k)$ is fulfilled and that therewith the inner minimization routine terminates after finitely many iterations as we assumed in the convergence proof of the outer routine, Algo. 3.1.

4 Results and Outlook

In [9] we presented first promising results, also in comparison with the nonsmooth optimization tools HANSO [2] and MPBNGC [3]. These results were improved by calculations with the refined algorithms introduced above.

In the future the new optimality condition from [8] and the resulting reflection method will be integrated. Both the optimality condition and the reflection method exploit additional information of the polyhedral structure and therewith they imply an additional improvement of the performance of the presented minimization method LiPsMin.

References

A Matlab Implementation of the Minpack-2 Test Problem Collection

Shaun A Forth*

8th April 2016

1 Introduction

Following the initial work of Rich and Hill [1] and the prolific effort of Verma [2] there have been numerous projects to develop AD tools for the Matlab™ problem solving environment [3]. There is a clear need for a set, or sets, of standard test cases in order to verify such AD tools. Such test cases must necessarily include the source code to be differentiated and preferably also verified derivative code so that derivative accuracy may be assessed without resort to finite differencing or the complex variable trick [4].

The Minpack-2 Test Problem Collection of Averick et al [5] was developed to test the performance of numerical optimization software. The collection contains Fortran 77 subroutines to define 24 optimisation problems of three problem types:

- **Unconstrained minimisation** - a single subroutine that returns: standard starting vector $x_s$, lower $x_l$ and upper $x_u$ bounds on $x$, objective function $f(x)$ and gradient $\nabla f(x)$.

- **Least squares minimisation** - a single subroutine that returns: standard starting vector $x_s$, lower $x_l$ and upper $x_u$ bounds on $x$, residual function $F(x)$ and Jacobian $J_F(x)$.

- **Systems of nonlinear equations** - a single subroutine that returns: standard starting vector $x_s$, lower $x_l$ and upper $x_u$ bounds on $x$, residual function $F(x)$ and Jacobian $J_F(x)$.

A subroutine argument `task` is set to specify which value is to be returned by the subroutine, eg, `task='XS'` returns the starting vector, `task='FJ'` returns the residual value and its Jacobian for a specified $x$.

Several of the problems are large-scale and for these subroutines to compute Hessian or Jacobian sparsity patterns, and Hessian- or Jacobian-vector products are also provided. The collection is widely used for performance and robustness testing of numerical optimization packages and the report [5] is referenced by 82 articles in the Scopus citation database. It is also widely used for verification of both Fortran and C/C++ AD tools (we presume the f2c utility is used prior to applying C/C++ AD tools).

1.1 Lenton’s Matlab Implementation of Minpack-2 Problems

In her MSc Dissertation, Lenton [6] hand-converted all the Minpack-2 problems to Matlab. This involved:

- Re-writing Fortran subroutines as Matlab functions

- Changes of syntax for loops, branches and array constructors

- Ensuring arrays have lower index bound 1 in Matlab and adjusting associated loop indices.

Her re-coding was verified by a 2-stage process:

1. A Fortran program was written: to create random vectors $x$; call the Fortran Minpack subroutines to compute $f(x)$, $\nabla f$, etc; write $x$, $f(x)$, $\nabla f$, etc to a formatted text file.

2. The text file data was then read into Matlab, and using the supplied $x$ recompute $f(x)$, $\nabla f$, etc using the Matlab versions of the Minpack-2 functions and compare results to the Fortran-computed results.

Results were found to agree to within input/output conversion and floating point round-off errors.

In this paper we detail further enhancements to our Matlab re-coding of the Minpack-2 test problem collection:

- Standardisation of the function interfaces to the problems - permitting easy scripting of test suites

- Vectorization of large-scale problems - permitting efficient overloaded AD regardless of problem-size

- Verification using Matlab’s unit-testing framework using intermediate binary files to eliminate input/output conversion errors.

*Centre for Simulation & Analytics, Cranfield University, Defence Academy of the United Kingdom, Shrivenham, SN6 8LA, S.A.Forth@cranfield.ac.uk
2 A Revised Matlab Implementation of Minpack-2 Problems

The original Fortran Minpack-2 subroutines, and Lenton’s Matlab coded functions, have inconsistent interfaces that vary depending on a problem’s physical parameters and whether the problem is defined on a 2-D mesh or not. We have standardised all functions for calculating \( f(x) \), \( \nabla f \), etc by encapsulating all problem-dependent data in a structure using a single function for each problem.

For example, the Human Heart Dipole (HHD) problem is of fixed problem size but comes in 6 versions with different model coefficients for each version. The function \( \text{MinpackHHD.Prob} \) depicted in Fig. 1 is used to set such data given a problem version number.

As a second example, the Minimal Surface Area (MSA) problem is to find the height \( z(x,y) \) on a uniform rectangular mesh of points on the unit square \( (x,y) \in [0,1] \times [0,1] \) to minimise the surface area given \( z(x) \) or \( z(y) \) on the four edges. For this case three forms of the problem definition may be performed:

1. \( \text{Prob} = \text{MinpackMSA.Prob}(nx,ny,bottom,top,left,right) \) we specify the number of interior grid points in \( x \), \( y \) directions, and vectors of \( z \) on the 4 boundaries

2. \( \text{Prob} = \text{MinpackMSA.Prob}(nx,ny) \) as for 1 but with default boundaries corresponding to Enneper’s problem ([7] referenced in [5])

3. [\( \text{Prob} \), nuse] = \( \text{MinpackMSA.Prob}(n) \) as for 2 but with \( nx = ny = \text{floor}(\sqrt{n}) \) and the returned nuse\( = nx \times ny \).

Having made uniform the problem specification it is now possible to make the function interfaces uniform to compute \( f(x) \), \( \nabla f \), etc. For example, for the HHD problem the function \( \text{MinpackHHD.FJ}(x,\text{Prob}) \) computes

- the residual vector \( F = \text{MinpackHHD.FJ}(x,\text{Prob}) \)
- or the residual and its Jacobian by \([F,J] = \text{MinpackHHD.FJ}(x,\text{Prob})\).

Internally to \( \text{MinpackHHD.FJ} \) the number of function output arguments is detected and only if there are two is the Jacobian calculated. The same function interface is then used for all other nonlinear equation problems in the collection, only the function name changes \( F = \text{MinpackCPF.FJ}(x,\text{Prob}) \), \( F = \text{MinpackCPR.FJ}(x,\text{Prob}) \), etc.

The ability to return both the residual and optionally the Jacobian is that required by nonlinear equation solvers using Newton’s method. In many cases the Jacobian cannot be hand-coded, which is why we need AD. In order to compare AD generated Jacobian performance with hand-coded we also supply a variant of the residual function that only computes the Jacobian and might be used as for \( \text{MinpackCPF.FJ}(x,\text{Prob}) \).

The fortran coding of the large-scale problems of Minpack-2, such as the MSA problem, uses loops over the arbitrary large mesh of unknowns. It is known that converting loops to equivalent vector-, matrix- or array-based operations may significantly improve performance in Matlab [8]. Additionally, for overloaded AD vectorisation reduces the number of operations that must be overloaded. For example, if \( A \) and \( B \) are \( n \times n \) arrays then the element-wise product \( X = A \star B \) involves just one overloaded operation involving \( n^2 \) floating point multiplication operations whereas an equivalent double loop would involve \( n^2 \) overloaded operations and the same number of multiplications. For this reason we supply fully vectorized versions of all of the large-scale Minpack-2 problems, as well as loop-based versions. For example, for the MSA problem we supply \( \text{MinpackMSA.Fvec} \) for the loop-based computation of the MSA problem’s objective function, and \( \text{MinpackMSA.Fvec} \) for the vectorized version.

2.1 Verification versus the Fortran Implementation

We wished to verify results from our Matlab coding against those from the original Fortran source code. Intel Visual Fortran can be used to compile Fortran \textit{Mex} subroutine interfaces to Fortran code rendering it callable from Matlab but such interfaces are difficult to write. To improve on the formatted text files used to communicate results between Fortran and Matlab calls of the Minpack-2 routines by Lenton [6] noted in Sec. 2.1 we use (unformatted) binary files to eliminate associated input/output decimal to/from binary conversion errors and stream input/output.
We used Matlab’s unit test framework [9] to verify the results from our Matlab functions against those from the Fortran subroutines for randomised inputs. Table 1 shows such results for the Minpack-2 nonlinear equation problems. The columns refer to: $x_0$ - standard starting point; $x_l/x_u$ - lower and upper bounds on $x$; $F$ - residual function; $J_F$ - Jacobian; $F + J_F$ - residual function and Jacobian; $S(J_F)$ - Jacobian sparsity pattern; $J_F \cdot v$ - Jacobian-vector product; $F_{\text{vec}}$ - vectorized residual function. An entry of n/a indicates that the computation is not applicable when: there are no upper or lower bounds on $x$; the problem is not large scale. A check mark indicates agreement of the Matlab and Fortran results to within a relative tolerance $\tau_{\text{rel}}$ and absolute tolerance $\tau_{\text{abs}}$. Although $\tau_{\text{rel}}$ and $\tau_{\text{abs}}$ were adjusted for each problem and each task the maximum values used were $\tau_{\text{rel}} = 2^7 \times \epsilon \approx 2.8 \times 10^{-14}$ for the residual from version 2 of the HHD problem and a $\tau_{\text{abs}} = 2^5 \times \epsilon \approx 7.1 \times 10^{-15}$ for the Jacobian of the Flow in a Driven Cavity (FDC) problem of size $n = 100$. The three crosses in Tab. 1 indicate those residual functions yet to be vectorized.

Table 1: Verification unit testing results for the Minpack-2 nonlinear equation problems

### 2.2 Some initial performance data

Table 2 gives the ratios of the averaged run times for Jacobian computations to the run time for the residual function computation for the small-scale Minpack-2 nonlinear equation problems. As well as the two Minpack-2 functions to compute residual and Jacobian, and Jacobian alone, we also give results using forward mode AD from the MAD package [10] with dense ($fmad$) and sparse ($fmad-sparse$) storage of derivatives. Computations were averaged over 100,000 repetitions except for the two MAD computations for which 500 repetitions were used. The large run-time ratios of the MAD computations are partly attributable to MAD using Matlab’s original, but now obsolete, form of objective oriented programming and so prohibiting use of the performance enhancing just-in-time compilation techniques of modern Matlab.

Table 3 gives run time ratios for two large scale problems - Flow in a Driven Cavity (FDC) and Flow in Channel (FIC). As these problems are large scale they may be run for different problem sizes $n$. For the FDC case the hand-coded Minpack-2 Jacobian ($F + J_F$ or $J_F$) is computed perhaps more slowly than expected (there are a maximum of 13 non-zeros in each row/column and a bandwidth of $5 \times \sqrt{n}$) - this may be due to the large amount of indexing into the sparse Jacobian storage which is performed row-wise whereas Matlab uses compressed sparse column storage. Without vectorisation ($fmad-sparse$) the MAD results are slow but vectorisation ($fmad-sparse(vec)$) improves performance by a factor of between 20 to 150. For the FIC case the hand-coded Minpack-2 Jacobian ($F + J_F$ or $J_F$) is computed in-line with expectations as it has 9 non-zeros per row/column and band-width of 9. Again applying MAD gives poor performance ($fmad-sparse$) unless it is applied to the vectorized source code ($fmad-sparse(vec)$) and then performance is only 3 to 10 times poorer than the hand coding.

Table 2: Jacobian computation run-time ratios for small-scale Minpack-2 nonlinear equation problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>n version</th>
<th>$x_0$</th>
<th>$x_l/x_u$</th>
<th>$F$</th>
<th>$J_F$</th>
<th>$F + J_F$</th>
<th>$S(J_F)$</th>
<th>$J_F \cdot v$</th>
<th>$F_{\text{vec}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPF</td>
<td>11</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CPR</td>
<td>5</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FDC</td>
<td>16,100</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>CPR</td>
<td>5</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>HHD</td>
<td>8 1-5</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FDC</td>
<td>16,100</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FDC</td>
<td>100</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FDC</td>
<td>400</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FDC</td>
<td>1600</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FIC</td>
<td>80</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FIC</td>
<td>320</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>FIC</td>
<td>1280</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

Table 3: Jacobian computation run-time ratios for two large-scale Minpack-2 nonlinear equation problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>$F + J_F$</th>
<th>$J_F$</th>
<th>$fmad$</th>
<th>$fmad-sparse$</th>
<th>$F_{vec}$</th>
<th>$fmad-sparse(vec)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDC</td>
<td>100</td>
<td>6.8 6.5</td>
<td>7100 100</td>
<td>0.45</td>
<td>85</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>FIC</td>
<td>80</td>
<td>6.7 3.3</td>
<td>1800 400</td>
<td>1.0</td>
<td>60</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>FIC</td>
<td>320</td>
<td>7.0 7.0</td>
<td>5700 1200</td>
<td>0.5</td>
<td>60</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>FIC</td>
<td>1280</td>
<td>12. 12.</td>
<td>7600 2400</td>
<td>0.15</td>
<td>60</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\)The constant $\epsilon \approx 2.2 \times 10^{-16}$ is machine relative epsilon for Matlab’s double precision variables.
3 Conclusions and further work

We have

• Detailed standardisation of the function interfaces to our Matlab re-coding of the Fortran Minpack-2 test problem
• Presented verification results to show that our Matlab implementation gives results agreeing to within floating point round-off errors with those from the original Fortran subroutines
• Demonstrated how vectorization of Minpack-2’s large-scale problems improves run-time performance of both the hand-coded derivative functions and overloaded AD.

Our full paper will also document our conversion of Minpack-2’s least squares and unconstrained minimisation problems, associated AD results, and results of applying the Matlab and TOMLAB [11] Optimization Toolboxes.

Once completed we will use the Minpack-2 text problems for verification and performance testing of updates and enhancements to our MAD package [10] and make available to others in the AD and optimization communities.

References


Algorithmic Computation of Finite Difference
Toshio Fukushima*
March 2016

1 Introduction

The algorithmic or automatic differentiation is a powerful technique to obtain a computer program to evaluate the derivatives of a given function if its computer program is available [1]. However, there are some cases where we need not the derivatives but the finite differences of the given function [2]. Let us explain an example [3].

Consider the numerical computation of Cauchy’s principal value of an improper definite integral [4] such as

\[ I = P \int_{-1}^{1} \frac{f(x)}{x} \, dx = \lim_{\varepsilon \to 0^+0} \left[ \int_{-1}^{-\varepsilon} \frac{f(x)}{x} \, dx + \int_{\varepsilon}^{1} \frac{f(x)}{x} \, dx \right], \tag{1} \]

where \( f(x) \) is assumed (i) to be analytic or (ii) to contain integrable singularities at \( x = 0 \) such as \( 1/\sqrt{|x|} \) or \( \ln |x| \).

This kind of integrals frequently appear in the boundary element methods, especially when computing the acceleration vector of a mass element inside a finite body caused by the self-gravitation or self-electromagnetic field [5].

One way to compute such an improper integral is the integrand reflection [6]. Noting a fact that

\[ P \int_{-1}^{1} \frac{dx}{x} = \lim_{\varepsilon \to 0^+0} \left[ \int_{-1}^{-\varepsilon} \frac{dx}{x} + \int_{\varepsilon}^{1} \frac{dx}{x} \right] = 0, \tag{2} \]

we reflect the integrand of \( I \) as

\[ I = \lim_{\varepsilon \to 0^+0} \int_{\varepsilon}^{1} \frac{f(x) - f(-x)}{x} \, dx. \tag{3} \]

Now that the singularity is (i) removed or (ii) relaxed and located at one end point of the integration interval, any integration method which can handle the integrable end-point singularities properly, such as the double exponential (DE) quadrature rule [7], may calculate the integral value accurately.

Unfortunately, there is a pitfall. The DE rule requires that the integrand is analytic. Thus, this approach will fail as long as the finite difference in the above integrand is computed by the subtraction, and therefore, is suffering from the catastrophic cancellation when the integral variable \( x \) approaches 0. In order to overcome this situation, we need a precise method to evaluate not the derivative but the finite difference of an arbitrary function, say at the level of machine epsilon. Of course, easy is the usage of the higher or multiple-precision arithmetic when the function is computable in such environment. However, this remedy is not always available and tends to be time-consuming even if it is feasible. Therefore, we seek for a different approach by tracking back the history of numerical computation.

Boole, who is famous for the Boolean algebra, is, perhaps except Newton himself, the first person who studied the finite differences extensively. He wrote a good text book on this theme [8] as a sequel of his treatise on the differential calculus. However, except the second chapter, he concentrated himself to the application of the finite differences to the numerical integration of ordinary and partial differential equations. This tendency was enhanced more and more by his followers and has resulted the establishment of the finite difference method. Consequently, few literature can be consulted for our purpose to compute the finite difference of given functions accurately. Therefore, we had to develop a formulation by ourselves. In this article, we report our trial briefly.

2 Theory

2.1 Operator Formalism of Finite Difference

Boole is the first mathematician regarded the manipulation of the finite difference as an operator [8, Chapter 1]. He defined it for a single-variable scalar function, \( f(t) \), as

\[ \Delta f \equiv f(t) - f_0, \tag{4} \]

*Corresponding Author, National Astronomical Observatory of Japan, 2-4-1, Ohsawa, Mitaka, Tokyo, 181-8588 Japan, Toshio.Fukushima@nao.ac.jp, https://www.researchgate.net/profile/Toshio_Fukushima/

66
where \( f_0 \equiv f(t_0) \) is the reference value corresponding to the reference argument \( t_0 \). Rigorously speaking, \( \Delta \) is expressed as a two-argument functional acting on the given function \( f \) as

\[
\Delta f = \Delta (f; t_0, h),
\]

where we write the argument difference as \( h \equiv t - t_0 \). However, unless the rigorous expression becomes necessary, we prefer a simpler notation introduced in the above. It is automatic to extend the definition for general vector/matrix-valued functions. Also, in the case of multiple-variable functions, the concept of the partial finite difference would be easily understood. Further, the computation of a higher order finite differences such as \( \Delta^n f \) becomes simple if repeatedly applying the basic rules of \( \Delta \), which will be explained in the next subsection.

### 2.2 Basic Rules

Some rules of \( \Delta \) are well known from the days of Newton, who actually invented the divided difference \( \Delta f/\Delta t \):

\[
\Delta a = 0, \quad \Delta t = h, \quad \Delta (a f + b g) = a \Delta f + b \Delta g, \quad \Delta (f g) = f \Delta g + g_0 \Delta f, \quad \Delta (g/f) = (f_0 \Delta g - g_0 \Delta f) / (f f_0),
\]

where \( a \) and \( b \) are numerical constants and \( f \equiv f_0 + \Delta f \). Notice the asymmetric appearance of the suffix 0 in the product rule. Meanwhile, the composite rule, which seems to be missing in the existing literature, is written as

\[
\Delta (f \circ g) = \Delta (f; g_0, \Delta g).
\]

This may require an explanation. Write a composite function as

\[
\phi(t) \equiv (f \circ g)(t) = f(g(t)).
\]

Assume that \( \Delta f, \Delta g, \) and \( \Delta \phi \) are expressed as explicit functions of \( t_0 \) and \( h \) as

\[
\Delta f = F(t_0, h), \quad \Delta g = G(t_0, h), \quad \Delta \phi = \Phi(t_0, h),
\]

then the above composite rule means that \( \Phi(t_0, h) \) is constructed from \( F(t_0, h), g(t), \) and \( G(t_0, h) \) as

\[
\Phi(t_0, h) = F(g_0, \Delta g) = F(g(t_0), G(t_0, h)).
\]

This rule is a key device to compute the finite difference of complicated functions from those of basic formulas as will be seen later. At any rate, the finite difference operations are significantly different from the differential ones.

### 2.3 Conditional Branching

There is an important factor in applying the rules of finite difference operation to the existing computer programs: the conditional branching. There are two kinds of branching in usual computer programs. One is the termination of do-loop. If the condition to terminate the computation of \( f \) depends on not \( t \) but \( f \), a care is needed: to inspect the terminate condition of \( \Delta f \), too. Indeed, there is a chance such that the computation of \( f \) is completed but that of \( \Delta f \) is not, and vice versa.

The other is a switch of computing procedures depending on the argument \( t \) and/or some intermediary variables. A simple example is a piecewise function defined as

\[
f(t) = \begin{cases} f_1(t) & (t \leq t_c) \\ f_2(t) & (t_c < t) \end{cases}
\]

where \( t_c \) is a critical argument, which we assumed to be fixed and known beforehand. In this case, assuming \( h > 0 \), we obtain the finite difference in three ways:

\[
\Delta f = \begin{cases} \Delta f_1 & (t_0 + h \leq t_c) \\ \Delta f_{12} & (t_0 < t_c \leq t_0 + h) \\ \Delta f_2 & (t_c < t_0 + h) \end{cases}
\]

where \( \Delta f_j \) for \( j = 1 \) and 2 are those obtained by simply applying \( \Delta \) to \( f_j \), respectively, and \( \Delta f_{12} \) is defined as

\[
\Delta f_{12} \equiv \Delta (f_1; t_0, h_1) + \Delta (f_2; t_c, h_2)
\]

where \( h_1 \equiv t_c - t_0 \) and \( h_2 \equiv t - t_c \). If the bit length of \( t_c \) is sufficiently small, then the subtraction operation to compute \( h_1 \) will cause no information loss. If not, one should first evaluate \( h_1 \), then compute \( h_2 \) so as to satisfy the condition \( h_1 + h_2 = h \) exactly. This resembles Kahan’s compensated summation [9].
2.4 Vector/Matrix Operations

Since $\Delta$ is a linear operator, it is trivial to extend its definition to the vector- or matrix-valued functions as

$$\Delta (u \cdot v) = u \cdot (\Delta v) + (\Delta u) \cdot v_0, \quad \Delta (u \times v) = u \times (\Delta v) + (\Delta u) \times v_0, \quad \Delta (PQ) = P (\Delta Q) + (\Delta P) Q_0,$$

$$\Delta \left( P^{-1} \right) = -P^{-1} (\Delta P) P^{-1}, \quad \Delta \left( P^{-1} u \right) = P^{-1} [\Delta u - (\Delta P) P^{-1} u_0],$$

where the last rule must be understood such that, if $x_0$ is the solution of a linear equation when $t = t_0$ as $P_0 x_0 = u_0$, then $\Delta x$ is obtained by solving an auxiliary linear equation defined as $P \Delta x = \Delta u - (\Delta P) x_0$.

2.5 Non-Trivial Formulas

Finally, we summarize some non-trivial formulas of $\Delta$ of basic and elementary functions:

$$\Delta (t^n) = h \sum_{j=0}^{n-1} t^j (t_0)^{n-1-j}, \quad \Delta (t^{-n}) = -h \sum_{j=0}^{n-1} t^j (t_0)^{n-1-j} / (tt_0)^n, \quad \Delta \left( \sqrt[1]{t} \right) = h / \sum_{j=0}^{n-1} \left( \sqrt[1]{t} \right)^{j-1} (\sqrt[1]{t_0})^{n-1-j},$$

$$\Delta (\exp(t)) = \exp(1) \exp(t_0), \quad \Delta (\ln(t)) = \log(1 \cdot t_0), \quad \Delta (\tan^{-1}(t)) = \tan^{-1} \left[ h / (1 + t_0) \right],$$

$$\Delta (\sin(t)) = 2 \sin(h/2) \cos(t_0 + h/2), \quad \Delta (\cos(t)) = -2 \sin(h/2) \sin(t_0 + h/2), \quad \Delta (\tan(t)) = \sin h / \left[ (\cos t)(\cos t_0) \right].$$

3 Examples

3.1 Composite Function

Let us apply the rules and formulas in the previous section to the construction of a procedure computing the finite difference of the given functions. Begin with a simple composite function [1]:

$$F(t) \equiv \exp \left( -\sqrt[1]{t} \right) \sin \left( t \ln \left( 1 + t^2 \right) \right).$$

First, we resolve $F(t)$ into pieces as

$$F \equiv F_1 F_2, \quad F_1 \equiv \exp G_1, \quad F_2 \equiv \sin G_2, \quad G_1 \equiv -\sqrt[1]{t}, \quad G_2 \equiv t H_2, \quad H_2 \equiv \ln I_2, \quad I_2 \equiv 1 + t^2.\eqno{17}$$

The algorithm to compute $F$ for $t$ is a backward sequence of these element operations as

$$I_2 = 1 + t^2, \quad H_2 = \ln I_2, \quad G_2 = t H_2, \quad G_1 = -\sqrt[1]{t}, \quad F_2 = \sin G_2, \quad F_1 = \exp G_1, \quad F_0 = F_1 F_2.\eqno{18}$$

where the suffix 0 means the reference values. This sequence is translated into that of the finite differences as

$$\Delta I_2 = h (t + t_0), \quad \Delta H_2 = \log(1 \cdot t_0), \quad \Delta G_2 = t (\Delta H_2) + h H_2, \quad \Delta F_2 = 2 \sin \frac{\Delta G_2}{2} \cos \left( G_2 + \frac{\Delta G_2}{2} \right),$$

$$\Delta G_1 = -h / \left( \sqrt[1]{t} + \sqrt[1]{t_0} \right), \quad \Delta F_1 = \exp(1) \Delta G_1 F_1, \quad \Delta F = (F_1 + \Delta F_1) \Delta F_2 + F_2 \Delta F_1.\eqno{19}$$

Thus, a computer program is algorithmically transformed into its finite difference version.

3.2 Recursion

Another illustration is a recurrence relation which is typically used to compute the classical orthogonal polynomials:

$$r_{n+1} = (a_n t + b_n) r_n + c_n r_{n-1}, \quad (n = 0, 1, \ldots)\eqno{20}$$

where $a_n$, $b_n$, and $c_n$ are certain numerical constants. The finite difference of $r_n$ are recursively computed as

$$\Delta r_{n+1} = (a_n t + b_n) \Delta r_n + c_n \Delta r_{n-1} + a_n h r_n \left( t_0 \right).\eqno{21}$$

For example, $\Delta T_n$, the finite difference of Chebyshev polynomial of the first kind, is recursively computed as

$$\Delta T_{n+1} = 2 t \Delta T_n - \Delta T_{n-1} + 2 h T_n \left( t_0 \right), \quad (n = 1, 2, \ldots)\eqno{22}$$

starting from the seed values $\Delta T_0 = 0$ and $\Delta T_1 = h$ where $T_n \left( t_0 \right)$ are prepared simultaneously or separately.

Also, an important extension of the recursion is the formulas to solve th ordinary differential equations such as

$$y_{n+1} = y_n + F \left( y_n, y_{n-1}, \ldots, y_{n-N}; t \right).\eqno{23}$$

This can be regarded as a predictor of the multistep method to integrate ordinary differential equations such as Adams method [10, Chapter 7]. If the finite difference of $F \left( y_n, y_{n-1}, \ldots, y_{n-N}; t \right)$ is algorithmically available, the numerical integrator can be also differented. This is useful in integrating the formation flight of space missions [11].
3.3 Nonlinear Equation

Move to the solution of a nonlinear equation. The elliptic Kepler equation is an important equation to be solved frequently in celestial mechanics and astrodynamics [12]:

\[ f(y; a, t) \equiv y - a \sin y - t = 0, \quad (0 \leq a < 1) \]  

(24)

where \( y \) is the unknown to be solved when the argument \( t \) and the parameter \( a \) are specified. Since its analytical solution does not exist, we must solve it numerically. The standard method is to solve it by the Newton method

\[ y_{n+1} = \frac{[t + a (\sin y_n - y_n \cos y_n)]}{(1 - a \cos y_n)}, \]

(25)

starting from a certain initial guess. By an appropriate transformation, the range of \( t \) is limited as \( 0 \leq t < \pi \). After this reduction, the Newton method is guaranteed to converge if starting from a simple initial guess, \( y_0 = \pi \).

At any rate, from the iterative procedure to obtain the solution \( y \), we algorithmically derive a procedure to compute \( \Delta y \), the finite difference of the solution corresponding to \( h \equiv \Delta t \), the finite difference in the argument. The resulting procedure is a nonlinear map expressed as

\[ \Delta y_0 = 0, \quad \Delta y_{n+1} = [(1 - ac_n) \Delta g_n + ag_{n0} \Delta c_n] / [(1 - ac_{n0}) (1 - ac_n)], \]

(26)

where the auxiliary quantities are defined as

\[ g_{n0} \equiv t_0 + a [s_{n0} - c_{n0} (y_{n0} + \Delta y_n)], \quad \Delta g_n \equiv h - a [(1 + y_{n0}) \Delta c_n + c_n \Delta y_n], \quad \Delta c_n = -2 \sin (h/2) \cos [y_{n0} + (\Delta y_n)/2], \]

\[ c_n \equiv \cos (y_{n0} + \Delta y_n), \quad s_{n0} \equiv \sin y_{n0}, \quad c_{n0} \equiv \cos y_{n0}, \quad y_{n0} \equiv y_n (t_0). \]

(27)

4 Conclusion

Triggered by a practical need [3], we developed a systematic method to compute the finite difference of an arbitrary function when its computing algorithm is explicitly given. The method is an employment of the basic rules of \( \Delta \), the finite difference operator, as well as the formulas of the finite differences of elementary and some special functions. If no recursion is contained in the given program or if their stability issue is positively resolved even when included, then, as in the case of automatic differentiation [1], it is straightforward to create a computer subprogram providing both the undifferenced and the differenced function values. Its practical implementation is, however, an open question.

References


Source-transformation Differentiation of a C++-like Ice Sheet model

Laurent Hascoët∗, Mathieu Morlighem†
March 2016

1 Introduction

Algorithmic Differentiation (AD) has become one of the most powerful tools to improve our understanding of the Earth System. It is routinely used to calculate model sensitivities to any model input, and to constrain numerical models using data assimilation techniques. If AD has been used by the ocean and atmospheric circulation modeling community for almost 20 years, it is relatively new in the ice sheet modeling community (e.g., [1]). The Ice Sheet System Model (ISSM) is a C++, object-oriented, massively parallelized, new generation ice sheet model that recently implemented AD to improve its data assimilation capabilities [2]. ISSM currently relies on Object Overloading through ADOL-C and AMPI. However, experience shows that Object Overloading AD on ISSM is significantly more memory intensive compared to the primal code. We want to investigate other AD approaches to improve the performance of the AD adjoint. Yet, to our knowledge, there is no source-to-source AD tool that supports C++.

To overcome this problem, we have developed a prototype of ISSM entirely in C, called Boreas, in order to test source-to-source transformation and compare the performance of these two approaches to AD. Boreas is a clone of ISSM, the main difference with ISSM is that all the objects are converted to C-structures and some function names have been adapted in order to be unique, but the code architectures are identical. The programming style of Boreas is a first attempt at defining a programming style of (or a sub-language of) C++ that source-transformation AD could handle. Boreas can be run in serial mode, or in parallel using MPI like ISSM. The adjoint of Boreas will also rely on AMPI for adjoin communication. To deal with parallel vectors and matrices and to solve linear systems, Boreas and ISSM rely on the Portable, Extensible Toolkit for Scientific Computation (PETSc).

The long term objective of this work is to use source-to-source transformation to differentiate ISSM. The first step, which we present here, consists in using Tapenade [3] to perform source-to-source transformation on Boreas, a C++-like C code. If Tapenade now officially supports C, differencing Boreas proved to be challenging. We present here some of these challenges and the strategies that we adopted to overcome them.

2 Extension of source transformation outside the differentiable code

File architecture of the source code, and in turn of the differentiated code, has been the first issue when applying AD to Boreas. Even though AD tools should ideally not be sensitive to the file architecture, we had to make some important changes in order for Tapenade to transform the primal code. For example, when the C source file is preprocessed before compilation (e.g. by \texttt{cpp}), it must also be preprocessed before differentiation. Consequently, the differentiated C code will be bound to one particular preprocessing output, coming from one set of preprocessing variable values. In other words, the differentiated C code will not contain \texttt{#ifdef} clauses, even if the source does. Similar issues apply to \texttt{#include} files. Even if we have reached today an acceptable strategy, it is still an open question whether the differentiated \texttt{include} files should contain definitions of differentiated objects only, or incorporate the original objects as well, and in this case should those be explicitly inlined or should they be added through an \texttt{#include} of the original \texttt{include} file.

We focus here on other issues that are more central to AD tool development. Namely, the automatic generation of a “calling context”, the static analysis of destinations of pointers and its relation to aliasing issues.

2.1 Pointer aliasing

The calling context in which the differentiable code is used has a clear influence on differentiation results. The most obvious illustration is aliasing, i.e. the possibility that the same storage location is referenced by apparently different syntactic elements. Aliasing seriously restricts static code transformation. For instance, a possible aliasing between $X$ and $Y$ will forbid a loop nest parallelization tool to detect the code

\[
\text{for}(i=2 \ ; \ i<\text{size} \ ; \ ++i) \ X[i] = Y[i-2];
\]

as parallel, although without aliasing this is just a simple array copy.

∗Corresponding Author, INRIA Sophia-Antipolis, laurent.hascoet@inria.fr
†University of California, Irvine, mathieu.morlighem@uci.edu
With former FORTRAN standards, aliasing was strongly discouraged, and AD tools often just required users to not use aliasing. The introduction of pointers in FORTRAN 90 makes this requirement more constraining. Forbidding aliasing becomes totally unrealistic in C, where aliasing is extremely common, if not encouraged. A source-transformation AD tool relies on accurate static data-flow analysis and, as such, needs a reliable detection of pointer aliasing. Tapenade is no exception and uses a static pointer destination analysis (“points-to” analysis) to determine whether two syntactic elements refer to the same memory location. This pointer analysis was already necessary to handle FORTRAN 90, and it becomes absolutely central for C/C++ [4]. The results of pointer analysis inform all following analyses of the function computation to handle correctly aliasing created during function computation.

However, a simulation code often separates the initialization phase that builds and initializes the data structures (e.g. mesh elements and the links between them) possibly setting aliasing at that stage, and the computation phase that takes values from and rewrites values into these data structures, implicitly relying on this aliasing. The computation phase is also generally followed by a post-processing phase. We refer to the union of the initialization and post-processing phases as the “context”.

Typically, AD should only focus on the actual computation phase. For instance, the end user specifies that they want the derivatives of the function implemented by the root procedure of the computation phase, more precisely of the dependent variables, with respect to the independent variables being among the arguments (formal parameters or globals) of that computation root. One could think that only the function computation code need to be passed to the AD tool. However, in doing so, the AD tool will be blind to aliasing created during the initialization phase. Figure 1 illustrates this pitfall for the case of Boreas. The computation phase corresponds to the function \texttt{FemModelSolve}, with independents X and dependents J. The computation copies X into the inputs component of the central object \texttt{FemModel}, then deals only with the values attached to the mesh elements to compute J without ever reading \texttt{FemModel->Inputs} explicitly. As a result, the AD data-flow analysis and in particular activity analysis finds that there is no differentiable link from X to J. The link appears only through aliasing of \texttt{FemModel->Inputs} with deep components of each of the mesh elements \texttt{Element->Inputs}, which is done during the initialization phase by the function \texttt{LinkDataSets} (which specifies that \texttt{Element->Inputs = FemModel->Inputs} for each element of the mesh). The issue can be solved in two ways:

- The AD tool can accept user directives that specify the existing aliasing upon entry of the computation root. Similarly, one can move part of the initialization that creates the aliasing inside the computation phase. When running on the computation phase, the pointer analyzer then correctly detects the aliasing. Consequently, activity analysis detects the differentiable link from X to J. This is the solution we have implemented so far. However, it disturbs the original code either by adding a number of possibly complex directives or by moving a structural, non-differentiable piece of code that initializes the mesh structure at a place where it does not belong.

- One can extend the pointer analyzer to the complete code, including the context. Aliasing information coming from the context would thus be exposed to AD additionally to aliasing coming from the computation code. This might imply some fine-tuning as the context contains more system calls, more problematic than what is usually found in the computation code, but this approach seems preferable in the long run. Still, this creates more complexity to the AD request: a given differentiation target (i.e. computation function plus dependent and independent variables) may produce different results for different contexts.

An immediate consequence of the second approach is that the complete simulation code must be exposed to the AD tool. Although this will certainly increase the memory size and run time of the AD tool, the benefits outweigh the cost. We will see in the next section that this is also beneficial to the initial execution, validation, and debugging of the differentiated code.

2.2 Generation of context code to call the differentiated code

The differentiated code of the computation root, obtained through AD, must be executed in an appropriate context. This context must call the differentiated code, providing it with the input derivative values in addition to the original inputs, and reading the output derivative values in addition to the original outputs. This context must also declare, allocate, and initialize the memory that hold these additional inputs and outputs, and must release this memory after differentiation. Although these tasks can be viewed as outside the realm of AD, it is not reasonable to leave them entirely to the end user. They are time-consuming, error-prone, and can be automated. Moreover, these tasks generally become manifest between the first successful run of the AD tool and the validation stage that should immediately follow. Starting the development of the calling context at that moment will handicap the user as they will lose the focus...
on their primary objective. Finally, automated generation of this context can also automate the setup for derivatives validation.

Here, we have extended the generation of the differentiated code to also create a calling context for the actual derivative code. This extension is triggered by adding the single command-line option `-context` to the AD tool invocation. This extension requires that all the original files that define the calling context of the computation root, up to and including the `main` procedure, are passed to the differentiation command. In other words, in addition to the code of the differentiable function, that must be passed to the AD tool in all cases, one must pass the context code that prepares for and calls the differentiable function. As a result, a new “differentiated” context code is produced that prepares for and calls the differentiated function. The new context code follows very closely the structure of the original context, performing no derivative computation, as it is outside the call tree of the differentiable function. Still, the context code declares, allocates and initializes all the data structures that will later hold the derivatives, and propagates them to the entry of the differentiated function. Upon return from the differentiated function, the context code cleans up and releases these data structures. These creation and destruction operations take place by mimicking the corresponding operations on the original data structures. These differentiated data structures and differentiated variables follow Tapenade’s association by name approach, but can be adapted quite easily to follow the alternative association by address where derivative containers are attached close to the original containers, deep at the level of the leaves of the data structures, therefore requiring no extra derivative variable names.

The main ingredient of this “context” functionality is a static data-flow analysis that runs over the complete code to find all allocation, initialization, and release operations of the original context that must have a differentiated counterpart. Insertion of appropriate declarations follows naturally from that. At each point in the code, we call `Req` the set of all variables for which the derivative variable is required downstream of that point, and that must therefore have been allocated and initialized upstream that point. The `Req` sets are computed by a data-flow analysis that runs backwards on the flow graph, i.e. in the direction opposite of execution. Similarly, we call `Avl` the set of all variables for which the derivative variables is available (i.e. allocated) and therefore for which the derivative variables must be released whenever they are released. The `Avl` sets are computed by a data-flow analysis that runs forwards on the flow graph, i.e. in the direction of execution. Like any data-flow analysis, both `Req` and `Avl` analyses deal with cycles in the flow-graph by running repeatedly until reaching a fixed point. They also propagate interprocedurally on the call graph, using a fixed point search to deal with recursive programs. On the call graph, both analyses first run a bottom-up sweep to compute summaries for each procedure, then run top-down, using these summaries when encountering a procedure call.

Let us focus here on the `Req` analysis, as the `Avl` analysis is straightforward. Regardless of the context functionality, the `Req` analysis is already needed for differentiation of the differentiable function code, because the AD model leads to introducing differentiated pointer variables. While activity analysis applies to variables of differentiable type, `Req` analysis extends it to pointers. An instruction `I` will be differentiated into some `I'` not only if its outputs are active but also if they intersect the `Req` set immediately after `I`. Only the differentiation rules are simpler. For instance, if `I` is the pointer arithmetic assignment:

\[ p = &A[2] + \text{offset}; \]

and assuming that the derivative of `p` is required downstream, one must generate the “derivative” assignment `I'`:

\[ p' = &A'[2] + \text{offset}; \]

where `p'` and `A'` are the differentiated variables of `p` and `A`. This defines the required `p'`, and in turn requires anterior definition of `A'`. Therefore, the data-flow equation of the `Req` analysis (backwards) across a statement `I` computes `Req` before `I` (noted `Req-^I(I)`) from `Req` after `I` (noted `Req^+^I(I)`): as:

\[
\begin{align*}
\text{Req}^-^I(I) &= \begin{cases} 
\text{Req}^+^I(I) \setminus \text{kill}'(I') & \text{if } \text{out}(I) \text{ is active or if } \text{out}(I) \cap \text{Req}^+^I(I) \neq \emptyset \\
\text{Req}^+^I(I) & \text{otherwise}
\end{cases}
\end{align*}
\]

where `use'(I')` (resp. `kill'(I')`) is the set of variables whose derivative variable is used (resp. totally overwritten) by the derivative instruction `I'` of `I`.

The novelty brought by the “context” functionality is that the `Req` and `Avl` analyses are now run on the context code as well. As there is no activity in any of these regions of the code, the data-flow equation above applies only when \(\text{out}(I) \cap \text{Req}^+^I(I) \neq \emptyset\), in other words, when the statement defines or modifies a pointer that may eventually be used in the differentiated part. The code generated for the context part is essentially the original code where every declaration, initialization, and propagation of a variable belonging to `Req` is followed by the same operation on the derivative variable. Similarly, propagation and release of a variable belonging to `Avl` is followed by the same operations on the derivative variable.

### 2.3 Validation and semi-automatic debugging of the differentiated code

When combined with the `-debugTGT` or the `-debugADJ` option of Tapenade, the context extension also adds into the differentiated context the infrastructure to validate the derivatives and to narrow down the faulty instructions if the derivatives are not valid. Here, we only discuss the validation of the tangent code.

In the differentiated context code, the call to the differentiated function is instrumented to implement two behaviors, depending on the value of a Unix shell variable. In the first behavior, the context perturbs the input of the differentiated function as \(X + \epsilon X\) for each independent input \(X\), with a random \(\epsilon\). In the second behavior, the differentiated function
is given the normal input $X$ and the same random $\dot{X}$ for the tangent derivative. In the derivative of the function computation part, test points are inserted by default upon entry into and exit from every procedure call. More test points can be inserted by the end user through directives. At each test point, the code that follows the first behavior writes into a FIFO file the value of the perturbed value for each active variable. Meanwhile for the second behavior, and for each active variable, the code reads the perturbed value from the FIFO file, computes the classical finite difference and compares it with the tangent derivative. Thus, running in parallel two instances of the differentiated executable, one for each behavior, executes the finite differences test to validate the tangent code. In case of an error, the test points prove very useful to narrow down the wrongly differentiated code.

3 Results and further work

We validate our tangent derivatives by comparison with centered finite differences with a well chosen $\epsilon$. We consistently obtain seven matching digits, which is satisfactory for a computation on double values. Some C primitives (e.g. memcpy) are “differentiated” by hand, as well as, classically, the calls to the linear solver. The tangent code runs approximately 1.7 times slower than the primal code.

To highlight the benefits of the AD tool’s activity analysis, we observe that the number of assignments in the tangent code is higher than in the original code by only 50%: many assignments on real values are detected as passive. To visualize this in another way, figure 2 shows the nesting of the original data structures, highlighting the only components that need a derivative in red. These are the only components that need to be computed, and that need storage space to hold the derivatives.

We have now started the next step, which is to use reverse AD to create the adjoint of Boreas. The programming style of ISSM and Boreas uses many intermediate arrays that are allocated, then released quickly after. This will raise the problem of storage of addresses in the adjoint code, i.e. addresses stored might point to released memory and thus may be no longer valid. To address this issue, we will use the experimental ADMM library, developed jointly with Argonne. It will then be meaningful to compare the overhead of overloading-based adjoint code of ISSM with the overhead of a source-transformation-based adjoint of Boreas, which might suffer from this intensive use of dynamic memory management. In particular, it is still unclear what the most efficient design options for ADMM should be. For instance, should ADMM recompute stored addresses when their underlying memory chunks were released then re-allocated at a different location, or should it implement its own malloc to make sure that re-allocating returns the same memory chunk? Only experiments on a large code such as Boreas/ISSM will tell us which option is best.

References


An Algorithmic Piecewise Linearization based Method for Nonsmooth Dynamical Systems

Richard Hasenfelder,* Andreas Griewank†, Tom Streubel‡

Even though the pleasing story that Bert Speelpenning [10] was told by Bill Gear to compute the Jacobians of stiff ODEs turned out to be a myth, the efficient numerical integration of dynamical systems was one of the earliest motivations for AD [8, 3, 9], which continues to be an almost indispensable tool especially for DAEs. Like in most of the literature it was assumed for the methods based on Taylor expansions [4] that the dynamical system is very smooth so that more or less arbitrary high convergence orders can be achieved.

For the many situations, where the right hand side is nonsmooth, i.e. has kinks, or jumps, a technique called event handling has been developed and implemented in MATLAB codes by Larry Shampine [1]. Here the user has to provide so called switching functions at whose roots in the state space the right hand side may undergo qualitative changes. The corresponding algorithmic modifications are somewhat heuristic and the resulting convergence theory is not very complete. When the switches come about by absolute values or conditional assignments of intermediate values, AD can automatically identify the switching surfaces, but unfortunately such an implementation has apparently not been done.

Here we pursue a different approach, namely the local approximation of nonsmooth right hand sides by piecewise linear models based on tangent or secant linearization. This leads to the construction of variants of the classical midpoint and Trapezoidal rules that maintain a local discretization error of order 3 and a global convergence order 2 in the presence of kinks on the right hand sides.

We want to study initial value problems

\[ \dot{x}(t) = F(x(t)), \quad x(0) = x_0 \]

where the right hand side \( F : \mathbb{R}^m \to \mathbb{R}^n \) is Lipschitz continuous and thus in realistic applications piecewise smooth. We assume it to be given by a computer model so it has to be composed of certain elementary functions (like sin, exp, sqrt...), which can as well as their derivatives be evaluated with machine precision. Additionally we allow for the usage of the absolute value function (and thus minimum and maximum) without demanding a derivative. We can then overload the right hand side function in an AD like fashion. This means we start with a base point \( \hat{x} \) and an increment \( \Delta x \) and propagate through the computational graph using the chain rule, such that for a smooth function \( F(x) \) we would end up with a directional derivative \( \Delta F(\hat{x}; \Delta x) = F'(\hat{x}) \Delta x \). Now if we encounter an absolute value in the computational graph, we set the derivative such that it holds \( v(\hat{x}) + \Delta v = \text{abs}(u(\hat{x}) + \Delta u) \) for \( v(x) = \text{abs}(u(x)) \). This gives us a piecewise linear approximation with a second order local error, just like the Fréchet derivative is on a smooth function.

\[ F(\hat{x} + \Delta x) - F(\hat{x}) - \Delta F(\hat{x}; \Delta x) = O(||\Delta x||^2) \]

We call this a piecewise linearization of the function.

There is a slight variation of this technique we call secant type piecewise linearization, where one replaces the partial derivatives \( \hat{c}_{ij} = \varphi'(\hat{v}_j) \) of \( v_i(x) = \varphi(v_j(x)) \) in the sense of Fréchet by a

*Corresponding Author, Humboldt University of Berlin, hasenfer@hu-berlin.de
†Yachay Tech, griewank@yachaytech.edu.ec
‡1st affiliation: Zuse Institute Berlin, 2nd affiliation: Humboldt University of Berlin, streubel@zib.de
secant slope in the two development points \( \hat{x}, \check{x} \) in the form:

\[
\dot{c}_{ij} = \begin{cases} 
\phi(v_j) & \text{if } \check{v} = \hat{v}, \\
v_i(\check{x}) - v_i(\hat{x}) & \text{else}.
\end{cases}
\]

If we do so we get a piecewise linear approximation \( \Delta F(\hat{x}, \check{x}; \Delta x) \) that fulfills the following bilinear approximation result:

\[
F(x) - \frac{1}{2}(F(\hat{x}) + F(\check{x})) - \Delta F(\hat{x}, \check{x}; x - \frac{\hat{x} + \check{x}}{2}) = \mathcal{O}(||x - \hat{x}|| ||x - \check{x}||).
\]

For further details and proofs we refer to [6].

The classical convergence theory of ODE integrators requires sufficient smoothness of the right hand side function. This means that the local discretization error of all classical methods, such as all Runge-Kutta methods, drops to \( \mathcal{O}(h^2) \) when integrating across a nondifferentiability. Thus we find that even in the best scenario, when the so called transversality condition is fulfilled, the global error of any Runge-Kutta method is bounded by \( \mathcal{O}(h^2) \). Transversality is equivalent to a finite number of nondifferentiabilities along the analytical solution trajectory and no tangential transitions through the surface of a nondifferentiability and the case we want to focus on here. We want to overcome this bound by proposing a method that maintains a local error of \( \mathcal{O}(h^3) \) everywhere without event location.

To do so we want to propose a new integration method that is a generalization of the trapezoidal rule. By using a secant type piecewise linear instead of the linear approximation used in the smooth method for the approximation of the integral we preserve the third order local discretization error in the nonsmooth case. The proposed method has an iteration scheme of the form

\[
\dot{x} = \hat{x} + h \int_{-\frac{1}{2}}^{\frac{1}{2}} (F(\hat{x}) + F(\check{x})) + \Delta F(\hat{x}, \check{x}; t(\hat{x} - \check{x})) dt,
\]

for a current point \( \hat{x} \) and a new point \( \check{x} \). We have proven the convergence order [6, 7] and validated it numerically [5]. A similar result can be achieved for the implicit midpoint rule [6, 5].

Since one cannot create an integration method with a local error better than \( \mathcal{O}(h^3) \) this way, a way to break the barrier of global second order convergence is Richardson extrapolation. With a standard Runge-Kutta method this does not yield any benefit, since the global error is bounded by the local error and this bound (of \( \mathcal{O}(h^2) \)) is already reached without any extrapolation, so that we cannot benefit from the local discretization error of order five on the smooth parts. We can however do so with the new piecewise linear integrator, since it has a local discretization error of order three instead of two on nondifferentiabilities, resulting in a global error of order three (see Table 1).

<table>
<thead>
<tr>
<th>Transversal Case</th>
<th>On Smooth Parts</th>
<th>On Kinks</th>
<th>Globally</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Rule</td>
<td>( \mathcal{O}(h^3) )</td>
<td>( \mathcal{O}(h^2) )</td>
<td>( \mathcal{O}(h^2) )</td>
</tr>
<tr>
<td>Generalized Rule</td>
<td>( \mathcal{O}(h^3) )</td>
<td>( \mathcal{O}(h^3) )</td>
<td>( \mathcal{O}(h^2) )</td>
</tr>
<tr>
<td>Class. w/ Romberg</td>
<td>( \mathcal{O}(h^5) )</td>
<td>( \mathcal{O}(h^3) )</td>
<td>( \mathcal{O}(h^2) )</td>
</tr>
<tr>
<td>Gen. w/ Romberg</td>
<td>( \mathcal{O}(h^5) )</td>
<td>( \mathcal{O}(h^5) )</td>
<td>( \mathcal{O}(h^3) )</td>
</tr>
</tbody>
</table>

Table 1: Local discretization errors

A noteworthy property of our method is the fact that due to the nature of the piecewise linearization it is an implementation of an AVF method (see [2]) on piecewise linear problems. This means on piecewise linear Hamiltonian problems we preserve energy, as was shown in [5] on a simple Hamiltonian system. This system has the form

\[
\begin{pmatrix}
\dot{x} \\
\dot{y}
\end{pmatrix} = \begin{pmatrix}
y \\
-V'(x)
\end{pmatrix}, \quad x(0) = 0, \quad y(0) = 1,
\]
where
\[ -V'(x) = \begin{cases} 
1 - x & \text{für } x \geq 1 \\
-1 - x & \text{für } x \leq -1 \\
0 & \text{für } |x| \leq 1.
\end{cases} \]

In Figure 1 we can see that the piecewise linear integrator unlike the smooth one does not lose energy on the kinks.

As the main result we want to present an error estimator for the piecewise linear trapezoidal rule allowing for the implementation of a step size control. We have to ensure that this error estimator just like the actual error does not artificially increase when the piecewise linear integrator crosses a nondifferentiability. It depends solely on the local Lipschitz constants of the right hand side \( F(x) \) and the approximation qualities of the method on the smooth parts of the system. The error estimators used for Runge-Kutta methods do not recognize this fact if they are applicable at all in this scenario (due to the lack of smoothness).

In order to construct the error estimator we will make use of the fact that the piecewise linear trapezoidal rule gives us a \( C^{1,1} \)-interpolant of the numerical trajectory for free. This is due to the fact that the trapezoidal rule uses a secant approximation of the integral and thus has correct function values \( x \) and tangent values \( F(x) \) at \( \tilde{x} \) and \( \hat{x} \). With this we can construct an interpolation polynomial \( p(t) \) of degree two that approximates the analytical solution trajectory, since \( \|x(t) - p(t)\| = O(h^3) \) whereas only \( \|F(x(t)) - F(\tilde{x} + \frac{t}{h}(\hat{x} - \tilde{x}))\| = O(h^2) \):

\[ L_F \int_0^h \|p(t) - \tilde{x} - \frac{t}{h}(\hat{x} - \tilde{x})\| dt. \]

Here \( L_F \) denotes the Lipschitz constant of \( F(x) \). Together we get an error estimator of the form:

\[ \|x(h) - x_h\| \leq L_F \int_0^h \|p(t) - \tilde{x} - \frac{t}{h}(\hat{x} - \tilde{x})\| dt + \frac{1}{12} \gamma_F h \|\tilde{x} - \hat{x}\|^2. \]
A first result is depicted in Figure 3 for the second example from [5], a stiff, non-smooth LC-circuit system.

This part of the work is still unfinished, as a few implementation questions are yet unsolved. But we expect to be able to present the mentioned step size control soon.

These results allow for a broad variety of applications like nonsmooth dynamical systems or discretized PDEs. Especially interesting is the application to highly stiff systems or even systems that do not satisfy the transversality condition. Furthermore this technique holds the potential for the application to discontinuous systems. By allowing not only the absolute value but also a sign or Heavyside function as elementary functions, one should be able to still achieve at least a first order approximation and thus guaranteed convergence.

References


Pattern Graph for Sparse Hessian Matrix Determination

Shahadat Hossain* and Nasrin Hakim Mithila†

March 30, 2016

1 Introduction

For a once continuously differentiable function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ in some neighborhood of $x \in \mathbb{R}^n$ we can write

$$\frac{\partial F(x + ts)}{\partial t} \bigg|_{t=0} = F'(x)s \equiv As \approx \frac{1}{\epsilon}[F(x + \epsilon s) - F(x)] \equiv b,$$  

(1)

where $s \in \mathbb{R}^n$ is a given direction. By taking $s$ to be unit coordinate vectors $e_i, i = 1, \ldots, n$ the nonzero elements of $F'(x)$ can be approximated with $n$ extra function evaluations (assuming that $F(x)$ has been evaluated already). With Automatic or Algorithmic Differentiation (AD) [1] forward mode one can also compute the product $b = F'(x)s$ for a given direction $s$ in one forward accumulation (approximately at the cost of a small constant multiple of the function evaluation). In this paper we are concerned with direct determination methods for symmetric matrices, i.e., where $F$ is the gradient of a twice continuously differentiable function $f : \mathbb{R}^n \mapsto \mathbb{R}$. In a recent work we have proposed the pattern graph as a unifying framework for methods that exploit sparsity by matrix compression: row compression, column compression, or a combination of the two in sparse Jacobian matrix computation [2]. From an algorithmic viewpoint the structural correspondence between the matrix and its graph leads to a better exposition of the compression heuristics and their efficient computer realization. In this paper we employ pattern graph model to formulate direct determination of sparse Hessian matrices. The new formulation also enables us to generalize direct methods that exploits matrix symmetry. A comprehensive overview of methods for sparse Jacobian and Hessian matrix determination problems can be found in [3]. A natural representation of the structure of a Hessian matrix is to use its adjacency graph. There has been significant work on symmetry-exploiting determination methods where the vertices are partitioned into small number of groups or color classes such that the nonzero unknowns in each group can be determined directly or indirectly, via FD or AD [4, 5, 6, 7, 8, 9]. Gebremedhin et al [9] present several heuristics for star coloring (direct determination) and acyclic coloring (determination via substitution) and analyze the performance of the implemented algorithms on an extensive test suite. In this paper, we generalize symmetry exploiting direct determination of sparse Hessian matrices. The underlying mapping is defined on the nonzero unknowns of the Hessian matrix such that sparsity can be exploited at element level. Further, the mapping induces “multi-coloring” i.e., nonzero unknowns can assume more than one color or belong to more than one group as long as each nonzero can be determined directly. We demonstrate the advantage of the new formulation on an example sparse Hessian matrix that can be determined with strictly fewer Hessian vector products than partition-based direct determination.

The main notational conventions used in the paper are as follows. If an uppercase letter is used to denote a matrix (e.g., $A$), then the $(i,j)$ entry is denoted by $A(i,j)$ or by the corresponding lowercase letter $a_{ij}$. We also use the colon notation [10] to specify a submatrix of a matrix. For $A \in \mathbb{R}^{n \times n}$, $A(:,i)$ and $A(:,j)$ denote respectively, the $i$th row and the $j$th column. For a vector $v$ of column indices, $A(:,v)$ denotes the submatrix consisting of columns whose indices are contained in $v$. Similarly, $A(u,:)$ denotes the submatrix consisting of rows whose indices are contained in vector $u$. A vector is specified using only one dimension. For example, the $i$th element of $v \in \mathbb{R}^n$ is written $v(i)$. The transpose operator is denoted by $(\cdot)^T$. A blank or 0 represents a zero entry; any other symbol in a displayed matrix denotes a nonzero entry.

Let $\mathcal{J} \subset \{1, \ldots, n\}$ be a set of column indices of a Jacobian matrix $A$. For brevity we say “a group of columns $\mathcal{J}$” when we mean “a group of columns whose indices are in $\mathcal{J}$”. A key observation made in [11] for the approximation of the product $b = F'(x)s$ is that if the sparsity pattern of a group of columns $\mathcal{J}$ of the Jacobian matrix $A$ satisfies the property that for each $a_{ij} \neq 0, j \in \mathcal{J}$ there is no index $j' \in \mathcal{J}$ for which $a_{ij'} \neq 0$. An approximation to the $i$th component of the vector $\sum_{j \in \mathcal{J}} F'(i,j)$ is obtained via the difference (1) by setting the direction $s = \sum_{j \in \mathcal{J}} e_j$. The columns in $\mathcal{J}$ are said to be “structurally orthogonal”. Thus, given the zero-nonzero pattern of the Jacobian matrix $A \in \mathbb{R}^{n \times n}$, one obtains vectors $s_i \in \mathbb{R}^n, i = 1, \ldots, p$ and $p$ minimized such that the matrix-vector products

$$B(:, j) \equiv AS(:, j), j = 1, \ldots, p$$

or $B \equiv AS$

(2)

determine $A$ uniquely.

*Corresponding Author, University of Lethbridge, Canada, shahadat.hossain@uleth.ca
†University of Lethbridge, mithila@uleth.ca
The general problem of determination of Jacobian matrices can be viewed as a compression-reconstruction process whereby the number of matrix-vector products \( p \) is minimized and the reconstruction process is numerically stable and efficient.

1. **Seeding or Compression.** Obtain \( S \in \mathbb{R}^{n \times p} \) and compute \( B(= AS) \) using FD or AD.

2. **Harvesting or reconstruction.** For each row \( i \) of \( A \):
   
   (a) define the reduced seed matrix \( \hat{S}_i \in \mathbb{R}^{p_i \times p} \) for the \( i \)th row of \( A \):
   
   \[
   \text{define: } \hat{S}_i \equiv S(v,:) 
   \]

   (b) solve for the \( \rho_i \) unknown elements \( a_{ij} \neq 0 \)
   
   \[
   \text{solve: } A(i,v)\hat{S}_i = B(i,:) 
   \]

   where \( \rho_i \) denotes the number of nonzero elements in row \( i \) of \( A \) and vector \( v \) of length \( \rho_i \) contains the column indices of those nonzero elements.

It is known that (see Chapter 7 of [1]) \( p - \rho_i \) equations of the (overdetermined) linear system that correspond to known zeros in Step 2 of the above procedure can be ignored leading to a \( \rho_i \times \rho_i \) linear system from which the nonzero unknowns can be recovered.

The linear algebraic formulation as outlined above is important. Not only does it allow us to classify and compare sparse matrix determination techniques succinctly, the insight gained from the structure of seed matrix \( S \) is helpful for constructing algorithms and analyzing complexity of the determination techniques [2, 12].

## 2 Symmetric Direct Cover and Hessian Matrix Determination

Given matrix \( A, a_{ij'} \neq 0 \) is a lateral neighbor of \( a_{ij} \neq 0 \) in \( A \) such that the difference \( j' - j \) is the smallest if \( j' > j \) or such that the difference \( j - j' \) is the smallest if \( j > j' \) among all such indices \( j' \) in row \( i \). A lateral neighbor of \( a_{ij} \neq 0 \) in \( A^\top \) is its vertical neighbor in \( A \). The pattern graph associated with \( A \) is \( G_P(A) = (V, E) \), where

\[
V = \{v_{ij} \mid a_{ij} \neq 0, i = 1, 2, \ldots, m, j = 1, 2, \ldots, n\}
\]

and

\[
\{v_{ij}, v_{i'j'}\} \in E \text{ if } a_{ij} \text{ and } a_{i'j'} \text{ are lateral or vertical neighbors}.
\]

We call element \( a_{ij} \neq 0 \) unknown if its value has not been determined; otherwise the element is known. Then, for a symmetric matrix \( A \) if \( a_{ij} \) is known then so is \( a_{ji} \) such that any method for determining a symmetric matrix needs to determine only one of \( a_{ij} \) and \( a_{ji} \). We call a symmetric determination method direct determination if there is a seed matrix \( S \in \{0,1\}^{n \times p} \) such that each unknown \( a_{ij} \) is determined directly i.e., there is an index \( k \) such that \( a_{ij} = b_{ik} \) or \( a_{ji} = b_{jk} \) in the matrix equation \( AS = B \) in which the compressed matrix \( B \) is computed via e.g., AD or FD. A direct determination in which the number of columns \( p \) of \( S \) is minimum is said to be optimal.

Consider the pattern graph \( G_P(A) = (V, E) \) associated with matrix \( A \) where \( A = A^\top \). We denote by \( \mathcal{P}(U) \) the set of all nonempty subsets of set \( U \). Let \( \Phi : V \mapsto \mathcal{P}\{1,2,\ldots,p\} \) be a mapping. The seed matrix \( S \in \{0,1\}^{n \times p} \) defined by the mapping \( \Phi \) is:

\[
S(:,k) = \sum_{\{j \mid k \in \Phi(v_{ij})\}} e_j, \; k = 1, 2, \ldots, p,
\]

where \( e_j \) is the \( j \)th Cartesian basis vector.

Let \( J_i \) and \( L_j \), respectively, be vectors containing the column indices of the nonzero entries in row \( i \) and row \( j \) of matrix \( A \). Also let \( \hat{S}_i \equiv S(J_i,:) \) and \( \hat{S}_j \equiv S(L_j,:) \), respectively, denote the submatrix of matrix \( S \) associated with the nonzero entries in row \( i \) and row \( j \) of matrix \( A \). An unknown \( a_{ij} \) is said to be covered by matrix \( S \) if \( a_{ij} \) or \( a_{ji} \) can be uniquely solved in

\[
A(i,J_j)\hat{S}_i = B(i,:) \text{ or } A(j,L_j)\hat{S}_j = B(j,:), \text{ respectively}.
\]

Matrix \( S \) is said to constitute a cover for \( A \) if each \( a_{ij} = a_{ji} \neq 0 \) is covered. A cover is a direct cover if \( A \) can be determined directly from the cover.

Let \( G_P(A) = (V, E) \) be the pattern graph associated with matrix \( A \).

**Theorem 1.** Mapping \( \Phi : V \mapsto \mathcal{P}\{1,2,\ldots,p\} \) yields a direct cover for the Hessain matrix \( A \) if and only if \( \Phi \) satisfies the following in the pattern graph \( G_P(A) = (V, E) \):

for each \( a_{ij} \neq 0 \)

EITHER
1. \[ \Phi(v_{ij}) \cup \bigcup_{\{v_{ij'} | j' \neq j\}} \Phi(v_{ij'}) \neq \emptyset \]

OR

2. \[ \Phi(v_{ji}) \cup \bigcup_{\{v_{ji'} | i' \neq i\}} \Phi(v_{ji'}) \neq \emptyset \]

Proof. Let the mapping \( \Phi \) yield a direct cover for \( A \) that does not satisfy conditions 1 and 2. Then, for each \( k \in \Phi(v_{ij}) \) there is an index \( j' \neq j \) such that \( \Phi(v_{ij}) = \Phi(v_{ij'}) = k \) and for each \( k' \in \Phi(v_{ji}) \) there is an index \( i' \neq i \) such that \( \Phi(v_{ji}) = \Phi(v_{ji'}) = k' \). Then,

\[
b_{ik} = (e_i^\top A)(S e_k) = A(i,:) \left( \sum_{j'' \mid k \in \Phi(v_{ij''})} e_{j''} \right) = \sum_{j'' \mid k \in \Phi(v_{ij''})} a_{ij''} \neq a_{ij},
\]

since the index set \( j'' \) contains at least two elements. Similarly, it can be shown that \( b_{jk'} \neq a_{ji} \). Then, \( S \) does not constitute a direct cover implying that \( \Phi \) does not yield a direct cover – a contradiction.

To establish the converse consider the seed matrix \( S \) defined by \( \Phi \). We want to show that for each unknown \( a_{ij} \neq 0 \), we have \( a_{ij} = b_{ik} \) or \( a_{ji} = b_{jk'} \) in \( AS = B \). Since \( \Phi \) must satisfy at least one of the two conditions above assume, without loss of generality, that there is an index \( k \in \{1, 2, \ldots, p\} \) such that \( k \in \Phi(v_{ij}) \setminus \bigcup_{\{v_{ij'}, j' \neq j\}} \Phi(v_{ij'}) \). We claim that the \( k \)th column of \( \tilde{S}_i \), \( \tilde{S}_i e_k \), is the \( j \)th coordinate vector \( e_j \) giving \( b_{ik} = (e_i^\top A)e_j = a_{ij} \). This is clearly the case since for \( a_{ij'} \neq 0 \), we have \( k \not\in \Phi(v_{ij'}) \) implying that

\[
\tilde{S}_i(l, k) = \begin{cases} 1 & \text{if } l = j \\ 0 & \text{otherwise} \end{cases}
\]

Therefore, \( b_{ik} = (e_i^\top A)e_j = a_{ij} \).

Figure 1 displays a sparse Hessian matrix and its associated graph. Each pair of distinct vertices in the adjacency graph of the matrix are connected by a path of length 2. It can be verified that any star coloring will require more than 4 colors. Now consider the associated pattern graph. The identically colored vertices can be determined with one matrix-vector product each giving 4 matrix-vector products. The uncoloured vertices are known by symmetry. Thus, the matrix is completely determined with only 4 matrix vector products. The associated seed matrix has the following structure:

\[
\begin{pmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

By Theorem 1 the nonzero entries in a column may be directly determined in more than one Hessian-vector products. The example in Figure 1 demonstrates that using the pattern graph model sparsity and symmetry can be exploited more effectively. Specifically, the definition of cover allows for Hessian matrices with the most general sparsity patterns including instances with sparse diagonal. We remark that the adjacency graph representation of a symmetric matrix sparsity pattern does not define the sparsity of the diagonal elements.

Figure 1: A symmetric matrix \( A \) and its pattern graph \( G_P(A) \)
2.1 Heuristics for Symmetric Direct Cover

The direct cover algorithm (CDC) in [13] has been found to produce better coloring on a suite of practical problems among the three algorithms (ASBC, MNCO, CDC) tested in [14]. We are in the process of implementing a symmetric version of CDC and one based on recursive largest-fast graph coloring heuristic described in [15]. Numerical testings of the implemented algorithms are expected to be completed by summer 2016.

3 Concluding Remarks

In this paper we have presented a generalized model for the symmetry-exploiting direct determination methods for sparse Hessian matrices. Our model uses pattern graphs to represent the sparsity structure of Hessian matrices. The heuristic algorithms currently being investigated are based on the notion of direct cover of nonzero entries and are inspired by our work on combined row and column compression heuristics which have been found to be computationally efficient and yield superior coloring [15, 14].

References

1 Introduction

In [1], Langlois describes the Correction des Erreurs Numériques d’Arrondi (CENA) method. In this method, local roundoff error estimates are combined with derivatives computed using the reverse mode of automatic differentiation in order to provide an estimate of global roundoff errors. This approach is computationally efficient for computations with a single scalar output, but suffers from the potentially large memory requirements of the reverse mode. In this paper, we demonstrate that the same estimate can be computed using a version of the forward mode. With a proper initialization, the computational cost is comparable to that of the reverse mode for the scalar output case and cheaper than the reverse mode in the case of multiple outputs.

2 Background

In the CENA Method, we approximate the error $E_y$ in a result $y$ using the formula:

$$E_y \approx \sum_i \frac{\partial y}{\partial x_i} \delta_i,$$

where $x_i$ is the result of each instruction $i$ used in computing $y$ and $\delta_i$ is the local round-off error in computing $x_i$. In the CENA method, one computes the derivatives using the reverse mode of automatic differentiation using a number of operations proportional to the number of operations in the function evaluation. Unfortunately, employing the reverse mode also incurs a storage cost proportional to the number of operations in the function evaluation.

3 A Forward Mode version of the CENA Method

Instead of the reverse mode of automatic differentiation, we can employ the forward mode, using a seed matrix (vector) $\delta = [\delta_1 \delta_2 \ldots \delta_n]^T$ to directly compute the inner product $\frac{\partial y}{\partial x}^T \delta = \sum_i \frac{\partial y}{\partial x_i} \delta_i$. This is most easily comprehend using the so-called buddy variable approach [?]:

\[
x_{ibuddy} = 0.0 \\
x_i = f_i(x_j, x_k) + x_{ibuddy} \\
\]

which after differentiating and initializing the seed matrix for $x_i$ (via its buddy) yields

\[
x_{ibuddy} = 0.0 \\
ad_{xibuddy} = delta_i \\
x_i = f_i(x_j, x_k) + x_{ibuddy} \\
\]

\[
ad_{xi} = (dfidx_j*ad_{xj} + dfidx_k*ad_{xk}) + ad_{xibuddy} \\
\]

We note that $delta_i$ (the roundoff error in computing $x_i$) may not in general be available until after the computation of $x_i$; however, it is straightforward to simplify the derivative computation to:

\[
x_i = f_i(x_j, x_k) \\
ad_{xi} = (dfidx_j*ad_{xj} + dfidx_k*ad_{xk}) + delta_i \\
\]

*Corresponding Author, Argonne National Laboratory, hovland@mcs.anl.gov
4 Proof of Correctness

Theorem If for each statement \( x_i = \phi_i(x_1, x_2, \ldots, x_{i-1}) \), we compute
\[
E_i = \delta_i + \sum_{j=1}^{i-1} \frac{\partial \phi_i}{\partial x_j} E_j
\]
then \( E_i \) satisfies Equation 1. That is,
\[
E_i = \sum_{j=1}^{i} \frac{\partial y}{\partial x_j} \delta_j,
\]

Proof By induction.

\( E_1 = \delta_1 \)

Assume that
\[
E_i = \delta_i + \sum_{j=1}^{i-1} \frac{\partial \phi_i}{\partial x_j} E_j = \sum_{j=1}^{i} \frac{\partial x_i}{\partial x_j} \delta_j
\]
for all \( i \leq n \). Then,
\[
E_{n+1} = \delta_{n+1} + \sum_{j=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_j} E_j
\]
\[
= \delta_{n+1} + \sum_{j=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_j} \sum_{k=1}^{j} \frac{\partial x_j}{\partial x_k} \delta_k
\]
\[
= \delta_{n+1} + \sum_{j=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_j} \sum_{k=1}^{n} \frac{\partial x_j}{\partial x_k} \delta_k
\]
\[
= \delta_{n+1} + \sum_{j=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_j} \frac{\partial x_j}{\partial x_k} \delta_k
\]
\[
= \delta_{n+1} + \sum_{j=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_j} \delta_k
\]
\[
= \frac{\partial \phi_{n+1}}{\partial x_k} \delta_{n+1} + \sum_{k=1}^{n} \frac{\partial \phi_{n+1}}{\partial x_k} \delta_k
\]
\[
= \sum_{k=1}^{n+1} \frac{\partial \phi_{n+1}}{\partial x_k} \delta_k
\]

5 Implementation

We have implemented the forward mode CENA method using an operator overloading library. An example of some of the overloaded functions is in Figure 1.

6 Conclusions

We have developed and implemented a forward mode version of the CENA method for roundoff error estimation. For simple functions, this implementation provides accurate error estimates at a minimal cost.

Acknowledgements

We thank Vincent Baudouin for introducing us to the CENA method and constructive conversations about roundoff error. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program, under contract number DE-AC02-06CH11357.
double addition_error(double a, double b, double x) {
    if (fabs(a) >= fabs(b)) {
        return (x - a) - b;
    } else {
        return (x - b) - a;
    }
}

double multiplication_error(double a, double b, double x) {
    const double m = pow(2,26) + 1; // depends on the number of bits in the mantissa
    double au, al, bu, bl;
    au = (a - a*m) + a*m;
    al = a - au;
    bu = (b - b*m) + b*m;
    bl = b - bu;
    return x - au*bu - (au*bl + al*bu) - al*bl;
}

double division_error(double a, double b, double x) {
    return (x*b - a - multiplication_error(x, b, x*b)) / b;
}

adouble operator+(const adouble &g1,const adouble &g2){
    double value = g1.value + g2.value;
    double localerr = addition_error(g1.value,g2.value,value);
    return adouble(g1.value+g2.value,g1.error+g2.error + localerr);
}

adouble operator-(const adouble &g1,const adouble &g2){
    double value = g1.value - g2.value;
    double localerr = addition_error(g1.value,-g2.value,value);
    return adouble(g1.value-g2.value,g1.error-g2.error + localerr);
}

adouble operator*(const adouble &g1,const adouble &g2){
    double value = g1.value * g2.value;
    double localerr = multiplication_error(g1.value,g2.value,value);
    return adouble(value,g1.value*g2.error+g1.error*g2.value + localerr);
}

adouble operator/(const adouble &g1,const adouble &g2){
    double recip,newval;
    recip = 1.0 / g2.value;
    newval = g1.value * recip;
    double localerr = division_error(g1.value,g2.value,newval);
    return adouble(newval,recip*g1.error - recip*newval*g2.error + localerr);
}

Figure 1: Overloaded functions for forward mode CENA.
References

Operator Overloading Compatibility for AD - A Case Study of Scientific C++ Codes

Alexander Hück∗ Christian Bischof†

March 2016

1 Introduction

The semantic augmentation of scientific C++ codes in order to compute derivatives can be realized by either source transformation or operator overloading. Source transformations applied to C++ are a non-trivial matter, because of generic programming techniques such as templates or the usage of preprocessor macros, for example. On the other hand, operator overloading can semantically augment the code while leaving most of the code structure unchanged. To that end, the floating-point type $T$ is replaced by a user-defined AD type $\tilde{T}$, providing all necessary overloads, e.g., for arithmetic operations and mathematical functions. However, the introduction of a user-defined type can cause compile time errors due to the way the C++ standard [1] treats built-in and user-defined types. In our previous work [2,3], we have classified several code constructs that can cause errors after such a type change:

**Implicit Conversion** is a conversion from type $T$ to another type $U$ without an explicit conversion statement in the code. For class objects of $\tilde{T}$, these implicit conversions are called user-defined conversions. The C++ standard states ([1] §12.3-4): “At most one user-defined conversion [...] is implicitly applied to a single value.”.

```cpp
1 struct $\tilde{T}$ {
2 $\tilde{T}$(double a);
3);
4 struct X {
5 x($\tilde{T}$ y);
6);
7 void foo(const X& x);
8 foo(1.0); // Error, two user-defined conversions 1.0 -> $\tilde{T}$ -> X: foo(X($\tilde{T}$(1.0)));
```

**Implicit Boolean Conversion** is a conversion from type $T$ to a Boolean type. This conversion is done automatically by the compiler, e.g., for the built-in floating-point types. $\tilde{T}$ can not be implicitly converted to a Boolean type $\tilde{T}$ can not be implicitly converted to a Boolean type and needs to either provide a non-explicit Boolean conversion function ([1], §12.3.2) or introduce an explicit comparison. The latter is preferred since a (non-explicit) conversion function may have unintended side-effects as the compiler is free to transform the user-defined class in undesired contexts [4].

```cpp
1 $\tilde{T}$ a(0.0);
2 if(a) {...} // Error: Needs a conversion function or a comparison, i.e., a ! = 0.0
```

**Explicit Conversion** is a conversion from type $T$ to another type $U$ with an explicit statement in the code (cf. [1], §5.4). For a conversion applied to $\tilde{T}$ to work, the value $T$ needs to be exposed. This can be achieved by using an explicit conversion function using C++11.

```cpp
1 $\tilde{T}$ a(0.0);
2 int b = (int) a; // Error: Cast operation needs the underlying value $T$ exposed
```

**Union** is a type of class whose members share a memory region. The C++03 standard restricts [5], §9.5: $\tilde{T}$ “with a non-trivial constructor, [...] non-trivial copy assignment operator cannot be a member of a union”. This has since been relaxed. Starting with C++11 [1], such a non-trivial object can be a member of a union, but the non-trivial functions are implicitly deleted if there are no user-provided definitions ([1], §9.5-2).

In either case, changing one of the types in a union via operator overloading is likely to change the semantics of the union in an unforeseeable way and should be avoided.

*Corresponding Author, Scientific Computing, TU Darmstadt, Germany, alexander.hueck@sc.tu-darmstadt.de
†Scientific Computing, TU Darmstadt, Germany
Name Lookup pertains to finding the corresponding declaration of a name (e.g., a function call encountered by the compiler) ([5] §3.4). The lookup can be unqualified, or qualified (using the scope resolution operator ::). Two main problems have been identified with qualified name lookups: (1) The usage of friend functions ([5], §11.4) without declarations in the scope of the class $\tilde{T}$ for overloading mathematical functions and qualified lookup (e.g., ::$\sin(a)$) can cause compile errors due to the disabled argument dependent lookup (ADL, [5] §3.4.2). ADL is used by the compiler to find the callee based on the context in these cases, e.g., the type of the argument. (2) If the target software of the type change defines its own definitions of certain functions in its own namespace, the respective (unqualified) lookup becomes potentially ambiguous if $\tilde{T}$ provides these functions as overloads.

```cpp
struct $\tilde{T}$ {
    friend $\tilde{T}$ $\sin$ (const $\tilde{T}$& a) { .. }
};

namespace ns {
    $\tilde{T}$ $\sin$ (const $\tilde{T}$& a); // Error
    $\tilde{T}$ a(2.0);
    ::$\sin$ (a); // OK : $\tilde{T}$ $\sin$ (const $\tilde{T}$& a)
    $\tilde{T}$ $\ns::$ $\sin$ ($\tilde{T}$ v); // OK : $\tilde{T}$ $\ns::$ $\sin$ ($\tilde{T}$ v)
    ::ns::$\sin$ (a); // Error: Ambiguous call
}
```

Listing 1: Case (1)

```cpp
struct $\tilde{T}$ {
    $\tilde{T}$ $\sin$ (const $\tilde{T}$& a) { ... }
};

namespace ns {
    $\tilde{T}$ $\sin$ ($\tilde{T}$ v) { ... }
    $\tilde{T}$ a(2.0);
    ::$\sin$ (a); // OK : $\tilde{T}$ $\sin$ ($\tilde{T}$ v)
    $\tilde{T}$ $\ns::$ (v); // OK : $\tilde{T}$ $\ns::$ (v)
    ::ns::$\sin$ (a); // Error: Ambiguous call
}
```

Listing 2: Case (2)

We have developed a static analyzer based on the Clang compiler framework [6] called OO-Lint. Our analysis targets codes before the semantic augmentation with an AD type is applied. It searches for these problematic code constructs, emits Lint-like messages and can also apply automatic source transformations to the respective code locations to remedy the situation. These transformations are not further discussed here, cf. [3]. To that end, Clang provides the abstract syntax tree (AST) of each respective translation unit (i.e., a .cpp file) including semantic nodes. We use Clang’s AST matcher implementation to apply sophisticated pattern matching on the AST. Figure 1 shows the design of our tool. The compilation database contains all paths to the translation units and each relevant compiler invocation string. Clang requires this to correctly build the AST.

![Diagram](image.png)

Figure 1: Design of the developed Clang-based tool. [2].

2 Results

We have statically analyzed four different software packages with our tool.

**OpenFOAM** (OpenField Operation and Manipulation, [7]) is a CFD solver package and of current research interest at TU Darmstadt. It heavily employs macros, templates and operator overloading throughout its C++ code base of about 640,000 lines of code (LOC).\(^1\) The fundamental data type is defined using the alias scalar, which is a built-in double type in the standard configuration. In previous work [3], we had analyzed an older code base (2.4.x), here we report on a current version 3.0.x.

**ULF Solver** (Universal Laminar Flame Solver, [8]) is a non-public solver for chemically reacting, laminar flows, written in C/C++. The ULF solver is under active development by the chair of Numerical Thermo-Fluid Dynamics (NTFD) at TU Bergakademie Freiberg. It makes no use of any special alias for the underlying floating-point type. The ULF solver code base has about 40,000 LOC.

\(^1\)Measured on the src folder of OpenFOAM-3.0.x
SU2 [9] is a CFD solver package written in C++ with about 80,000 LOC. Notably, the code base inherently supports AD with the CoDiPack AD tool [9, 10]. SU2 uses the **su2double** alias for the computations. It can be set to the AD type at compile time. We have also analyzed SU2_EDU (34,000 LOC) which is a trimmed down version for educational purposes. It does not use the aforementioned alias and is not written for compatibility with AD.

ISSM (Ice Sheet System Model, [11]) is a modeling software for ice flow developed by NASA/JPL. The core makes use of C++, but data (pre- and post-) processing is done in Matlab. Notably, the software was made compatible with operator overloading AD over a time frame of several years and the AD version based on ADOL-C [12] is currently considered to be fully functional. Currently, the C++ code base consists of 95,000 LOC. ISSM distinguishes between active and passive types using the aliases **IssmDouble** and **IssmPDouble**, respectively.

Table 1 shows analysis results of OpenFOAM, ULF Solver, SU2 and SU2_EDU indicating how often a problematic code construct occurred. OpenFOAM exhibits the majority of problematic code constructs. It is the only framework, so far, which has the name lookup complication described in the previous section. In the OpenFOAM namespace several transcendental function are defined. Thus, AD tools overloading them will introduce ambiguous lookups (cf. Section 1 Name Lookup). We counted both calls to the global namespace (e.g., ::sin) and to the local namespace (e.g., foam::sin) to exemplify this problematic situation in the OpenFOAM code base. Hence, the transcendental function inside the OpenFOAM namespace need to be handled carefully to resolve any potential issues for a type change to be successful. In the ULF solver, several conversions would have caused trouble with operator overloading.

SU2 is a special case, as it is inherently compatible with AD. The developers ensure compatibility for AD whenever relevant code parts are added. Unsurprisingly, we could not detect any problematic code constructs. On the other hand, the EDU version uses plain double built-ins only as AD capabilities are omitted. It exhibits a few problems related to explicit conversions and an implicit Boolean conversion in a conditional.

Table 1: Result for OpenFOAM, ULF Solver, SU2 and SU2_EDU.

<table>
<thead>
<tr>
<th>Construct</th>
<th>OpenFOAM</th>
<th>ULF Solver</th>
<th>SU2</th>
<th>SU2_EDU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation Units</td>
<td>1992</td>
<td>125</td>
<td>109</td>
<td>25</td>
</tr>
<tr>
<td>Implicit Conversion</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Implicit Boolean Conversion</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Explicit Conversion</td>
<td>35</td>
<td>25</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Union</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Name Lookup</td>
<td>307</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In Table 2, we present parts of the results of a comparison between a version of ISSM before all AD related transformations were applied to the code base (from 2011) with a more recent one (from 2014), which had all major parts of the AD computations functional. The zero issues of the AD-enabled version compared to the late 2011 version shows that the flagged issues were legitimate problems. The full results are presented in detail in [3].

Table 2: ISSM evolution w.r.t. problematic code constructs [3].

<table>
<thead>
<tr>
<th>Metric</th>
<th>Late 2011</th>
<th>Mid 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>912</td>
<td>857</td>
</tr>
<tr>
<td>LOC</td>
<td>66,573</td>
<td>80,044</td>
</tr>
<tr>
<td>Translation Units</td>
<td>298</td>
<td>254</td>
</tr>
<tr>
<td>#Explicit Conversion match in #files</td>
<td>192 in 46</td>
<td>0</td>
</tr>
<tr>
<td>#Implicit Boolean Conversion match in #files</td>
<td>44 in 10</td>
<td>0</td>
</tr>
</tbody>
</table>

These results show that OO-Lint is useful as a first step towards augmenting a C++ code base with AD through operator overloading. For all these codes, we also applied the OO-Lint transformation capabilities and they were able to at least syntactically resolve the issues discussed, with the exception of the union and name lookup issues in OpenFOAM, which do need human inspection and insight to change the code in a semantics-preserving fashion.

3 Conclusions and Future Work

Our results show that code bases that have not been developed with AD in mind exhibit coding constructs that cause problems when one applies operator overloading of the base floating point type in order to enable automatic differentiation. Hence, our approach of identifying, and in many cases automatically resolving these code constructs greatly reduces the human effort required in augmenting codes with automatic differentiation capabilities. Even for

---

2 Measured on the Common and SU2_ folders.
codes that are being designed with AD in mind, it is a useful tool to ensure that these issues have not been overlooked. In particular, the targeted messages emitted by OO-Lint are much easier to understand and relate to the code base than the voluminous cryptic error messages created by C++ compilers.

In future work, we will investigate improved detection capabilities. Especially abstraction mechanism such as templates still pose a challenge w.r.t. correct code construct classification. As the LLVM/Clang infrastructure also offers potent program analysis capabilities, which at the institute have, for example, been used to augment codes for performance analysis [13], we believe that OO-Lint, in addition to ironing out problematic code constructs, can in the future also develop more towards a general compile-time assist system for AD based on operator overloading.

4 Acknowledgements

We thank Christian Hasse and Danny Messig of the TU Bergakademie Freiberg for kindly providing us access to their ULF Solver code base. We thank Eric Larour of the NASA/JPL for making ISSM available.

References


Exact replication of OpenMP-parallelism in reverse-mode AD for loops with symmetric memory access

Jan Hückelheim∗ and Paul Hovland† and Michelle Strout‡ and Jens-Dominik Müller§ et al.

April 2016

1 Introduction, related work

Reverse-mode algorithmic differentiation (AD) is an efficient method to obtain adjoint gradients in applications including optimisation and uncertainty quantification in weather forecasting[1], computational fluid dynamics[2], machine learning[3] and finance[4]. Many of these applications demand the use of high-performance computers, and the advance of multi-core and many-core architectures[5] makes it increasingly important to exploit the shared-memory parallel capabilities of computers to keep runtimes acceptable. A widely used technique for shared-memory parallelisation is OpenMP[6], due to its portability and non-intrusive pragma-based coding style.

AD for OpenMP is not well established. A major challenge is the fact that communication between threads is not explicitly written in the code, in contrast to e.g. distributed memory parallel applications using message passing, where reverse-differentiation is much better understood[7, 8]. Instead, communication happens whenever more than one thread accesses a particular memory address. In order to reverse the communication for the adjoint code, an AD tool needs to analyse the memory access pattern[9], or resort to very conservative approaches with critical/atomic sections that do not scale well[10]. Applications that occur in practice such as unstructured-mesh computational fluid dynamics solvers have a memory access pattern that is unknown at compile-time. The parallel schedule typically depends on a graph partitioning[11] or colouring[12] which is only realised at run-time, and thus any approach that depends on a compile-time analysis of the parallel schedule is likely to fail.

To address this, we present a novel AD strategy that re-uses the parallelisation of the original (primal) code for the adjoint computation. With our approach, an AD tool does not need to detect the memory access pattern at compile-time. Instead, the adjoint code is generated in a way that ensures that the memory access pattern of the adjoint code is identical to that of the primal code. Hence, the correctness of the parallel schedule of the adjoint code follows from the correctness of the parallel primal code, which is given by the user. As a result, the adjoint and original codes exhibit comparable scaling characteristics. Since we can typically assume that the primal code was carefully hand-optimised to perform well on the available hardware for the problem that the user is interested in, we can be confident in the scalability of the adjoint code as well.

Our method works by identifying code segments in the primal that can be reverse-differentiated without changing their memory access pattern. This is possible for parallel loops that have a Jacobian matrix with symmetric sparsity pattern, and straightforward for parallel loops where each iteration has a Jacobian with symmetric sparsity pattern. An important class of OpenMP-parallel loops where this is the case is commonly found in partial differential equation solvers using the Finite Volume or related methods on both structured and unstructured meshes.

Previous work on differentiation of shared-memory parallel programs requires complete analysis of the memory access pattern or resorts to safeguarding to avoid race-conditions by means of atomic/critical/reduction pragmas in the generated code[13, 9] whenever there is nontrivial read-access to shared memory in the primal code. Preliminary results regarding the application of our method to a CFD code have been presented in a conference talk[14].

Our method does not require safeguarding of write access if the primal code was also free of atomic/critical sections. We make the following contributions in this work: We present a strategy to parallelise the adjoint loop for parallel primal loops that implement a function with a Jacobian matrix with symmetric sparsity pattern, a way to implement the detection and source-transformation necessary to implement our method in an AD tool, and a proof of correctness for the source-transformation shown in our work. We further present a template-based system to perform the source-transformation to generate parallel adjoint code. The work concludes with a case study where the scalability of a primal code and its generated adjoint code are demonstrated on a system with 16 CPU cores, as well as on an Intel XeonPhi accelerator card with 240 threads.

∗Corresponding Author, Queen Mary University of London, j.c.hueckelheim@qmul.ac.uk
†Argonne National Laboratory, hovland@mcs.anl.gov
‡University of Arizona, mstrout@cs.arizona.edu
§Queen Mary University of London, j.mueller@qmul.ac.uk

AD2016 - Programme and Abstracts
2 Replication of primal memory access in adjoint code

Following our method, the adjoint code is parallelised in a way that mimics the parallelisation of the primal code. For this, it is necessary that the Jacobian matrix of the parallel loop has a symmetric sparsity pattern, and hence that the size of the shared input and output array is identical. For code that has several input and output arrays and scalars, our approach can be used for portions of the input and output space that have a symmetric block in the Jacobian matrix corresponding to them.

Let us first consider a special case that is commonly found in practical applications, where each iteration has itself a Jacobian with symmetric sparsity. We then consider the general case where each iteration may have a non-symmetric sparsity pattern, but the overall result has a symmetric sparsity.

2.1 Case 1: Symmetric memory access in every iteration

In this simple case, each iteration uses the same indices to access its input and output arrays. We assume that the primal code is free of data races and hence that no two iterations perform write access to the same memory location at any given time. Since the input and output arrays are indexed in the same way, it follows that there is no concurrent read access from the input array. This is a sufficient condition to guarantee a parallel adjoint code that is free of data races, a property referred to as exclusive read property in [9].

The natural result in this case is an adjoint loop that can be executed in parallel without conflicting write access to \( \tilde{x} \). An example to illustrate this is shown in Algorithm 1 which implements a computation on a coloured graph as a serial outer loop over colours and a parallel inner loop over edges with that colour, with some function that reads from and writes to both vertices connected by each edge.

```
!$OMP for (c in colours)
    for (l, r in edges(c))
        F \times \leftarrow f(x_1, x_r);
        y_{l} \leftarrow y_{l} - F \times x_l;
        y_{r} \leftarrow y_{r} + F \times x_r;

Algorithm 1: Parallel edge flux computation
```

The adjoint code arises naturally as a loop that can be parallelised in the same fashion, as shown in Algorithm 2. In this case, it can be illustrated graphically (see Figure 1) that the adjoint code can be parallelised using the same colouring scheme as the primal. In general, we show in our work that it is enough to ensure that each primal iteration will use the same set of indices for the output and input arrays to guarantee that the adjoint loop can be executed in parallel with no safeguarding for the access to shared arrays. This is regardless of the underlying parallelisation scheme used to avoid data races in the primal code.

```
!$OMP for (c in colours)
    for (l, r in edges(c))
        F \times \leftarrow \tilde{y}_{r} - \tilde{y}_{l};
        \tilde{y}_{l} \leftarrow \tilde{y}_{l} + \tilde{F} \times \tilde{x}_{l};

Algorithm 2: Parallel adjoint flux
```

2.2 Case 2: Asymmetric iterations, symmetric loop

It is possible for a parallel loop to implement a function \( F \) with a Jacobian with symmetric sparsity, yet for each iteration to have a non-symmetric Jacobian sparsity, i.e. the set of input and output indices does not match for at least one (or indeed every) loop iteration. In such a case, the adjoint of two iterations \( i, j \) may cause write access to the same index in \( \tilde{x} \) even if their corresponding primal iterations do not cause conflicting writes in \( y \).
Nevertheless, it is possible to rearrange the adjoint computations such that the memory access of the adjoint code is identical to that of the primal. Consider the following $2 \times 2$ example and its adjoint:

\[
\begin{pmatrix}
  y_1 \\
  y_2
\end{pmatrix} := \begin{pmatrix}
  f_{1,1}(x_1) + f_{1,2}(x_2) \\
  f_{2,1}(x_1) + f_{2,2}(x_2)
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
  x_1 \\
  x_2
\end{pmatrix} + = \begin{pmatrix}
  f'_{1,1}(x_1) \\
  f'_{2,1}(x_1)
\end{pmatrix} \cdot \begin{pmatrix}
  y_1 \\
  y_2
\end{pmatrix}
\]

where the primal is implemented as a parallel outer loop as shown in Algorithm 3. The naive reverse-mode differentiation results in an outer loop that has to be executed in serial as shown in Algorithm 4 due to write conflicts in the output array (both outer iterations try to write to both indices in $x$). In contrast, a modified adjoint code can be created that reads and writes to the same indices as the primal code in every iteration, as demonstrated in Algorithm 5. The results produced by Algorithms 4 and 5 are identical. The concept is further illustrated in Figure 2.

!$OMP for shared(x, y);
for i \leftarrow 1 \ldots 2 do
  | y_i \leftarrow 0;
end

!$OMP for shared(x, y);
for i \leftarrow 1 \ldots 2 do
  | for j \leftarrow 1 \ldots 2 do
    | y_i \leftarrow y_i + f_{i,j}(x_j);
  end
end

!$OMP for shared(y);
for i \leftarrow 2 \ldots 1 do
  | y_i \leftarrow 0;
end

!$OMP for shared(x, y);
for i \leftarrow 2 \ldots 1 do
  | for j \leftarrow 2 \ldots 1 do
    | \bar{x}_j \leftarrow \bar{x}_j + \bar{f}_{i,j}(x_j) \cdot \bar{y}_i;
  end
end

!$OMP for shared(y);
for i \leftarrow 2 \ldots 1 do
  | \bar{y}_i \leftarrow 0;
end

Algorithm 3: Primal parallel loop

!$OMP for shared(x, \bar{x}, \bar{y});
for i \leftarrow 2 \ldots 1 do
  | \bar{x}_i \leftarrow \bar{x}_i + \bar{f}_{i,1}(x_1) \cdot \bar{y}_i;
end

!$OMP for shared(y);
for i \leftarrow 2 \ldots 1 do
  | \bar{y}_i \leftarrow 0;
end

Algorithm 4: Adjoint serial loop

!$OMP for shared(x, \bar{x}, \bar{y});
for i \leftarrow 1 \ldots 2 do
  | \bar{x}_i \leftarrow \bar{x}_i + \bar{f}_{i,1}(x_1) \cdot \bar{y}_i;
end

!$OMP for shared(y);
for i \leftarrow 1 \ldots 2 do
  | \bar{y}_i \leftarrow 0;
end

Algorithm 5: Adjoint parallel loop

3 Detection and source-transformation

To implement our method in an AD tool, it is necessary to detect parallel loops with a symmetric Jacobian matrix. This can be achievable in practice for the case where each iteration itself has a symmetric Jacobian, since in such cases typically the same expression is used in the source code to access indices in the input and output vectors. In more general cases, a pragma given by the user can be used to guarantee this property for a given loop, an approach that has been proposed for other optimisation strategies in the past[15].

3.1 Code generation for symmetric iterations

The source-transformation required to generate parallel adjoint code is relatively straightforward for loops where each iteration has symmetric memory access. It is sufficient to reverse-differentiate the code as usual, and then place an OpenMP parallel loop pragma in front of the adjoint loop whereby all adjoint variables corresponding to a private primal variable have to be private, and all adjoint variables corresponding to a shared primal variable have to be declared as shared.

3.2 Code generation for asymmetric iterations

The strategy required to generate parallel adjoint code with the same parallelisation as the primal can be regarded as a hybrid between a forward-mode and reverse-mode differentiation. The set of indices from which the adjoint input array $\bar{y}$ is read can be derived from the indices from which the primal code reads in $x$, while the adjoint output array $\bar{x}$ in each iteration is written to at the same indices at which the primal code writes to $y$. The derivative values are computed by using a multi-activity reverse differentiation of the loop body, a method that we will describe in more detail in this paper.

4 Test case

To test our method, we apply it to the edge-based compressible CFD flow solver mgopt[16] which simulates flow using the Finite Volume method on unstructured meshes. The solver consists of ca. 30000 lines of Fortran 90 code, of which ca. 10000 are core solver routines that are differentiated using the AD tool Tapenade[17]. Most of the core routines are parallelised using OpenMP for intra-node parallelism. The parallel loops are implemented either as loops over interfaces between fluid control volumes or as loops over control volumes themselves. In both cases, the implementation was found to have symmetric memory access in all iterations of these loops, which makes this code a suitable test case for our method for fully symmetric loops.

In addition to mgopt, we will show the application of our method to an example that has an overall symmetric memory access, but asymmetric access in each iteration. The tests are run on a compute node on the QMUL cluster with Intel(R) Xeon(R) CPU E5-2660 with 8 cores each, clocked at 2.20GHz for a total of 16 cores. In addition, we test the performance of our OpenMP parallel adjoint solver on an Intel XeonPhi Coprocessor (MIC). The MIC is a 5110P Knights Corner clocked at 1.053GHz with 60 physical cores supporting a total of 240 threads.
Figure 3: Scaling of the symmetric flux loop in mgopt and its adjoint generated with our approach on 16 CPU cores

Figure 4: Scaling of the same code on a XeonPhi accelerator card using up to 118 threads

References


1 Introduction

Two dominant methodologies exist for computing the differential of an algorithm without having to explicitly program its derivative: the first and more popular is to replace the floating point type of the algorithm with another type which augments the relational behaviour of the original floating point type with the side effects of recording every operation and associated state to some unique stack. The derivative is evaluated by interpreting the stack after the algorithm has completed its execution. This method is commonly referred to, in context, as operator overloading. The second methodology is to use a tool to read the source code which implements the algorithm, and have the tool generate new source code which contains the implementation of the original algorithm and its derivative.

Both methods have inherent limitations with respect to their application to large industrial codes: the first method destroys compiler optimisation opportunities due to its side effects and its exclusively run-time governed interpretation of the heap-stored stack. The second potentially retains the high-performance characteristics associated to manually implemented differentiated code, but is generally difficult, if not impossible, to apply it to codes written modern system programming languages due to their expansive complexity.

Since the operator overloading approach recasts the fundamental numerical data type, the primal data must always be accessed in tandem with the derivative data. In the case of adjoint evaluation, there is a conflict of use: the primal will only be read, whilst the adjoint will be modifiable. Binding the data prevents any distinction. Consequently, ambiguity of const correctness is introduced, and, in the case of parallel computation, inter-partition exchanges are unnecessarily doubled in size.

Performance is a perennial issue in industrial numerical software, and especially in the field Computational Fluid Dynamics, where enormous computational resources are dedicated to solving simulations. It would not be unreasonable for the user of such software to expect the adjoint of a flow simulation to take about the same time as the flow simulation itself requiring proportional similar memory. To achieve this parity, both the flow algorithm and its adjoint would need to have similar implementation characteristics and compile to highly efficient machine code. A compromise on efficiency from the outset regarding how the adjoint is implemented would hinder the adoption of the feature, especially for industrial users to routinely simulate flow problems close to the maximum capacity afforded by their systems.

In addition to these difficulties, there is the often overlooked but critical issue of program build varieties. It is already common to have two versions of a numerical analysis software, one compiled with mixed precision floating point types and the other with double precision types. With a continuous delivery build system, every variety of the released program must be compiled and tested on a nightly basis. If another version was added, such as build using a differentiable double precision type, the resources required to absorb this extra work load may very well outweigh the perceived benefit in the added functionality being delivered.

With these constraints in mind, hand coding the required differentiated components appears to emerge as the path of least resistance for obtaining the desired adjoint solver. It is under this perspective that the following work pursued, whose aim is of offering some helping-hand to carrying out the task whilst minimising performance compromises.

2 Methodology

Consider the contrived function in Listing 1 which has two input fields and one output field. By templating this function on a differentiation mode and renaming Field and double as types dependent on the provided mode, the proposed methodology enables the function to compute its primal, tangent or adjoint. Evaluating any of the modes is as trivial as calling the function with the correct mode; no recording of operations is performed.

```c++
void evaluate(Region const &region) {
  using double_t = double;
  // becomes: using double_t = Drv<mode, TypeValue<double, 7> >;
  Field<const Pressure, Cell> fp(region);
  Field<const Volume, Cell> fv(region);
}
```
Listing 1: The common task of looping over all elements, filling the output field.

2.1 Destructors

Adjoint evaluation requires the reversing of the sequence of operations. This is triggered by the destructors of the local objects within the loop block scope. Expressions are captured and stored by the local objects during their construction. These captured expressions are retained in an opaque manner (i.e. as an array of `chars`) along with a pointer to its `adjoint` method, which has an assumed signature. Allied with these two pieces of data, the object triggers the execution of the expression adjoint when it goes out of scope.

Since the result is assigned rather than constructed and returned, its expression adjoint is evaluated immediately. This is a reasonable solution provided the state is not later mutated with anything other than accumulation operations. The use of the block scope as a trigger mechanism for adjoint implies that nested blocks cannot be permitted around local objects. Whilst this is unenforceable, this methodology has the prerequisite that code must be functionally pure, ruling out the possibility of block scopes in correctly written code.

2.2 Functional Purity

The dominant context of this methodology is the prerequisite of functional purity of the components of the program to be differentiated. This is a difficult prerequisite to honour, but is possible and is ultimately beneficial regardless of whether or not its pursuit is for the sake of adopting this methodology to compute derivatives. In the context of programming in C++, this can be summarised by the directives of: never passing an argument by non-`const` reference and always initializing class member data in the constructor member initialization list.

2.3 Referenced State

When an expression is evaluated in the adjoint mode, the ultimate data storing the derivative accumulation of the terms need to be written to. To achieve this, the state of all adjoint expression objects contains the primal value, the derivative value, and a reference which, upon construction, holds the address of the derivative value passed in (Listing 2). If no derivative parameter is supplied, as in the case when it is constructed from an expression (via a sub-class), the reference holds the address of its own derivative value.

```c++
1 template<typename double_t>
2 class Drv<
3   DrvMode::Adjoint,
4   double_t ,
5   typename std::enable_if<std::is_same<double, double_t >::value>::type>
6 {
7   public:
8     Drv(double const &primal)
9       : _primal(primal)
10       , _derivative(0)
11       , _adjoint(_derivative)
12     {}
13
14     Drv(double const &primal, double &derivative)
15       : _primal(primal)
16       , _derivative(derivative)
17       , _adjoint(derivative)
18     {}
19 }
```
In the context of evaluating the adjoint of expressions, copies of adjoint objects are made. When the default copy constructor is invoked, all member data is literally copied so that the new object’s adjoint reference holds the address of the copied derivative value, not to its own, enabling adjoint results to be correctly accumulated to its ultimate sources.

2.4 Capturing Expressions

The expression template technique relies on the ability to assign one object to another without the latter having inherent knowledge of the former’s type (i.e. the latter could never own an instance of the former within the constraints of the type system). For the evaluation of the tangent derivative this does not present a problem since the primal and derivative can be evaluated at the same time. For the evaluation of adjoints, somehow the expression received as the argument of the constructor must be stored by the object so as to be later evaluated in an adjoint manner when its destructor is called, i.e. somehow the host must own a copy of the assigned object and be able to make use of it.

Since the type of the expression object is lost once the constructor is evaluated, only raw data can be retained. Alone, nothing useful can be done with the raw data, but a delegate to a member function associated to the raw data enables the host object to evaluate the delegate. This is what is done in the destructor of Listing 3 to trigger the adjoint evaluation of the captured expression.

Two important details are noted here: the first is that despite the standard library offering tools for creating and using delegates (std::bind and std::function), their performance is suboptimal as they are general purpose tools (typically invoking heap allocations) so a method-to-function trait class is used instead. The second point is that in order to capture the expression, a sufficiently large block of raw (stack) storage is required. The required size cannot be known up front, so templating the host class on a suitable integral value to dictate the maximum size is resorted to.

```cpp
template<typename double_v_t>
class Drv{
  DrvMode::Adjoint,
  double_v_t,
  typename std::enable_if<std::is_same<double_v_t::value(), double_v_t>::value>::type>
  : public Drv<DrvMode::Adjoint, double>
{}

// 'adjoint' method signature
using Function = void(*)(void const = const this_. double const &derivative_);

template<typename Expr_t>
Drv(Expr_t expr)
  : Drv<DrvMode::Adjoint, double>(expr.primal())
  , object(this)object(expr))
  , function(this)function<Expr_t>())
{}

'Drv() { this->adjoint(_derivative); }

void adjoint(double const &derivative) const
  { if (_function) { (_function)(object, derivative); }
}

private:

template<typename Expr_t>
void const *object(Expr_t const &expr)
  {
    static assert(sizeof(_stack) >= sizeof(expr), "");
    std::memcpy(_stack.data(), &expr, sizeof(expr));
    return _stack.data();
  }

template<typename Expr_t>
```
static Function function()
{
    using Traits = FunctionTraits<
        decltype(&Expr::t::adjoint)
    >;
    return Traits::template function<&Expr::t::adjoint>;
}

static constexpr std::size_t size()
{
    return sizeof(double) * double_v.t::value();
}

std::array<char, size()> _stack;
void const * const _object;
Function const _function;

Listing 3: Capturing an expression for deferred evaluation

The reason for splitting the adjoint form of the double class into a super-class and and sub-class is to enable the latter to capture constructs built from the former. In other words, the object being captured cannot be some product of other terms of the same type of that which is capturing the expression. The size of that captured must be less than the cache size of the object doing the capturing. This would be impossible if all adjoint double types were identical. Instead, when building up the expression type, the operands must be type-sliced to its super-class (and owned by-value rather than by-reference).

2.5 Optimisation

To obtain the run-time performance achieved in this work, an optimisation must be made concerning what a given operator (such as a binary operator) holds as member data. Ordinarily, these operator classes hold copies of their operands, as opposed to const-qualified references, since any of the received operators may be temporaries, i.e. previous expression results. However, if either of them are not expression types, then they necessarily must be named objects, meaning they can be safely referenced. Doing so improves run-time performance by as much as 20%.

3 Performance

The methodology outlined in this paper incurs a run-time cost of 2.7, relative to the primal, for the adjoint evaluation of Listing 1. The size of the std::array of chars used to capture assigned expressions (Listing 3) is set to the minimum compilable value for the first column in the table, though the effect of doubling, trebling and quadrupling the array size showed no meaningful trend in the effect on execution time. To compare the run-time performance of the adjoint, the Adept library [1] was used to evaluate the adjoint of the same function. Its normalised execution time was 3.3.

Tests were compiled with g++ 5.1.0 (with the -O3 flag) on a machine running Intel Xeon E5-2650 processors. Results presented are obtained by computing the median-average run-time of 50 executions, where the field size is 100,000 elements.

4 Conclusion

A highly performant tool has been developed for aiding the implementon of adjoint code. Its functionality is limited, but what it does support, namely blocks of pure-functional numerical expressions, it supports very well. No change of fundamental numerical type is required at the lowest level, nor an external language parsing tool, nor any post-evaluation stack interpretation.

References

Optimisation of triple-ring-electrodes on piezoceramic transducers using algorithmic differentiation

Benjamin Jurgelucks* and Leander Claes†

March 2016

1 Introduction

For simulating, designing and constructing piezoelectric transducers a precise knowledge of their material properties is indispensable. Recently published material parameter estimation methods try to reconstruct the material parameters of these ceramics by comparing measured and simulated impedance of the piezoelectric transducers in question as part of an inverse problem. The solution method of the inverse problem usually requires non-zero derivatives of the values which are being compared with regard to the variables, at least at the starting point of the solution process. However, in the case of piezoelectric transducers these sensitivities may be very small and close to zero.

The impedance of such a piezoelectric transducer is usually measured by fully covering top and bottom of the device with conducting material, exciting one of these electrodes and measuring the effect on the opposing electrode. In [1] it was shown that the sensitivity of the piezoelectric transducers can be increased by replacing the fully-covering electrodes with triple-ring-electrodes, the sensitivity now depending on the geometry of the rings. The dependency was used to formulate an optimisation problem which, however, could not be solved to fullest satisfaction. The authors now revisit this optimisation problem with the help of the AD tool ADOL-C and present their work regarding the implementation of ADOL-C into the simulation tool CFS++.

2 Simulation Model

The equations of linear piezoelectricity in tensor form are

\[ \boldsymbol{\sigma} = \boldsymbol{c} \boldsymbol{S} - \boldsymbol{e}^T \boldsymbol{E} \]
\[ \boldsymbol{D} = \boldsymbol{e} \boldsymbol{S} - \epsilon \boldsymbol{E} \]

with the electrical flux density vector \( \boldsymbol{D} \), the electrical field vector \( \boldsymbol{E} \), the mechanical stress tensor \( \boldsymbol{\sigma} \) and the mechanical strain tensor \( \boldsymbol{S} \). Furthermore, assuming rotational symmetry and using Voigt notation, the elastic modulus tensor \( \boldsymbol{c} \), the piezoelectric coupling tensor \( \boldsymbol{e} \) and the electrical permittivity tensor \( \epsilon \) can be rewritten as matrices containing the ten for later optimisation relevant material parameters:

\[ \boldsymbol{c} = \begin{pmatrix} c_{11} & c_{13} & 0 & c_{12} \\ c_{13} & c_{33} & 0 & c_{11} \\ 0 & 0 & c_{44} & 0 \\ c_{12} & c_{13} & 0 & c_{11} \end{pmatrix} \]
\[ \boldsymbol{e} = \begin{pmatrix} 0 & 0 & e_{15} \\ 0 & e_{13} & 0 \\ e_{15} & e_{33} & 0 \end{pmatrix} \]
\[ \epsilon = \begin{pmatrix} \epsilon_{11} & 0 \\ 0 & \epsilon_{33} \end{pmatrix} \]

The equations can be complemented to a time and space dependent PDE

\[ \rho \ddot{\boldsymbol{u}} + \alpha \rho \dot{\boldsymbol{u}} - \mathcal{B}^T (\mathcal{C} \mathbf{Bu} + \beta \mathbf{c} \dot{\boldsymbol{u}} + \epsilon \nabla \phi) = 0 \text{ in } \Omega, \ t \in [0, T] \]
\[ \nabla \cdot (e \mathbf{Bu} - \epsilon \nabla \phi) = 0 \text{ in } \Omega \]

with mechanical displacement \( \mathbf{u} \), electric potential \( \phi \), density \( \rho \), Rayleigh damping parameters \( \alpha, \beta \) and the differential operator

\[ \mathcal{B} = \begin{pmatrix} \partial_r & 0 \\ 0 & \partial_z \end{pmatrix} \]

\[ \mathcal{C} = \begin{pmatrix} 0 & \partial_r & \partial_z \\ \partial_r & 0 & \partial_r \\ \partial_z & \partial_r & 0 \end{pmatrix} \]

*Corresponding Author, Institute of Mathematics, Paderborn University, Benjamin.Jurgelucks@math.uni-paderborn.de
†Measurement Engineering Group, Paderborn University
The simulation code CFS++ \[2\] calculates an FEM solution \(u\) and \(\phi\) by applying an electrical charge pulse to one of the electrode rings of the piezoelectric ceramic via an appropriate boundary condition of the PDE. Post-processing three simulation runs with different connection settings of the three electrodes via Matlab delivers the impedance of the device. This is achieved by dividing the electric potential at the rings \(\phi\) by the current flowing at the connected rings \(\tilde{q}\). An additional Fourier transform \(\tilde{f}\) gives

\[
Z_i(\omega) = \frac{\tilde{f}(\phi_i(t))}{\tilde{f}(\delta\tilde{q}_i(t))}, \quad i \in \{1, 2, 3\}.
\]

These three impedances are finally combined to a total impedance by connecting the device to an external circuit and solving Kirchhoff’s laws.

As stated in the introduction, this leaves us with a complex valued impedance \(Z\) depending on frequency \(\omega \in \mathcal{F}\), the material parameters \(p_i \in \mathbf{p} \subseteq \mathbb{R}^{10}\) as well as the ring radii \(\mathbf{r}\) representing the ring geometry

\[
Z : \mathcal{F} \times \mathbf{p} \times \mathbf{r} \to \mathbb{C}.
\]

We define the partial sensitivity of impedance with regard to a specific material parameter \(p_i, 1 \leq i \leq N_p \leq 10\) as the \(L^2\)-norm of the curve

\[
\frac{\partial Z(\cdot; p_i, \mathbf{r})}{\partial p_i} \in \{\mathcal{F} \to \mathbb{C}\}
\]

and denote

\[
\|\nabla_p Z(\cdot; p_i, \mathbf{r})\| : = \left\| \frac{\partial Z(\cdot; p_i, \mathbf{r})}{\partial p_i} \right\|_{L^2(\mathcal{F})} = \left( \int_{\mathcal{F}} \|\nabla_{z_i} Z(\omega; p_i, \mathbf{r})\|^2 d\omega \right)^{\frac{1}{2}}.
\]

We calculate the sensitivity only for the parameters \(p_i, 1 \leq i \leq N_p \leq 10\) relevant for the specific optimisation case and measure the overall sensitivity by considering the (weighted) euclidean norm

\[
\nabla z(\mathbf{r}) := \left\| W \left[\|\nabla_p Z(\cdot; p_i, \mathbf{r})\|\right]_{i = 1, \ldots, N_p} \right\|_2
\]

with \(\left[\|\nabla_p Z(\cdot; p_i, \mathbf{r})\|\right]_{i = 1, \ldots, N_p} \in \mathbb{R}^{N_p}\) the vector containing all partial sensitivities and with the diagonal weight matrix \(W \in \mathbb{R}^{N_p \times N_p}\) which is in the easiest case the unit matrix. For future reference we also denote \(J(\mathbf{r}) := \nabla z(\mathbf{r})\) as the cost function for optimisation with \(r \in \mathcal{R}\) the set of feasible electrode configurations.

## 3 Implementation

For the evaluation of partial sensitivities the derivative of impedance with respect to material parameters, and hence also the derivative of potential \(\phi\) with respect to material parameters is needed. The authors of \[1\] did not have access to the source code of the simulation and thus were limited to finite differences for this calculation. However, the authors now have access to the source code of CFS++ and were able to apply the AD tool ADOL-C. We will first discuss challenges and restrictions imposed by CFS++ concerning the application of ADOL-C and then discuss complexity.

### 3.1 Challenges and Restrictions

Many parts of the simulation code needed for the solution of the PDE are real valued, however, some are complex valued. As there is no out-of-the-box way to handle complex-valued variables with ADOL-C, we chose to use std’s template-based implementation for complex values std::complex<adouble> and adapted many operators, i.e., \texttt{Sin} and \texttt{Cos} for (partial) differentiation of complex values used in CFS++. Hence, alongside the assumption of differentiability we are also assuming that the Cauchy-Riemann differential equations hold.

CFS++ makes extensive use of object oriented programing (especially polymorphism and inheritance), but also templates and template meta-programming. The combination of all three make the integration of ADOL-C’s adouble very troubling: Consider a case where we need to change the dataype of a variable from adouble back to double (this may happen when using the typedef approach #typedef adouble double). Clearly, the signature of all methods which call with this variable as an argument now have a type mismatch which can be resolved by adjusting the signatures. However, this must also be done in all directly or indirectly derived classes. Other methods or classes also may want to call this specific method and now themselves need to be adjusted etc. Conversely, this may need to be done for the base classes and their base classes from whom this variable was inherited as the type mismatch is now inflicted in the class/method of interest itself. Additionally, some template metaprogramming techniques prohibit generic template arguments. This means we may not change the template argument types in arbitrary locations, which adds even more complexity. The number of edits needed for a simple change of datatype grows exponentially with each layer until all layers are processed. Fortunately, these edits are more a complex matter than a complicated one.

A larger challenge poses CFS++’s use of string arithmetic. The two main reasons for CFS++ to use string arithmetic are:
The speed of frequent evaluations of a specific formula with different variable values can be significantly increased by saving and reusing a large portion of bitcode generated for the evaluation to disk thus avoiding overhead. CFS++ uses a specially customised version of the package muparser\textsuperscript{1} hardcoded into CFS++. The Input of muparser is managed via string.

Allowing the user to specify more flexible simulation configurations like boundary conditions given as a closed formula in a string. For greater generality, strings are furthermore used within the internals of CFS++ itself independently of the user’s input.

Once converted to string, a variable looses all derivative information previously stored. Any computations done in string arithmetic cannot easily be traced.

The authors first used ADOL-C’s traced forward mode for derivative calculation. As the tape’s size was over 400Gb, memory management significantly increased the runtime. The authors experimented with reevaluating a previously recorded tape: the solution of the PDE requires a timestepping scheme which would be an ideal location for reevaluation of the tape. Unfortunately, timesteps themselves are handled via strings in our simulation case, they cannot be an input variable for reevaluation of ADOL-C’s tape. This could possibly be overcome by instead including the intermediate variables from a lower hierarchy level not influenced by string arithmetic as input of the tapes evaluation.

3.2 Complexity

The authors successfully applied ADOL-C’s traceless forward mode to CFS++ for the simulation case discussed. In contrast to the traced forward mode, the runtime scales linearly with the input size (see Table 1). Within our expectation, we also note that unmodified CFS++ is much faster: The authors have not optimised any part of the code specifically for AD but plan to do so in the near future. Some profiling of the code indicates that the highest improvement of runtime is to be expected from externalising calls to solving linear systems of equations as described in \textsuperscript{3}.

<table>
<thead>
<tr>
<th>number of timesteps</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime unmodified CFS++</td>
<td>0.711</td>
<td>0.692</td>
<td>0.709</td>
<td>0.750</td>
<td>1.722</td>
<td>1.778</td>
<td>2.296</td>
<td>3.339</td>
<td>5.980</td>
</tr>
</tbody>
</table>

Table 1: Number of timesteps vs. runtime of one CFS++ simulation in seconds. The CFS++ version with ADOL-C calculates the derivative with respect to one parameter.

4 Comparison to Finite Differences

The accuracy of the derivatives is of central importance for the optimisation process: the cost function nearly entirely consists of derivatives that need to be calculated. The authors have validated the derivatives of CFS++ produced by ADOL-C by comparing them directly with derivatives produced by applying a finite difference scheme. This shows good accuracy. The postprocessing steps written in Matlab have been derived symbolically. An independent comparison with finite differences also shows good accuracy.

One major problem in the application of finite differences is choosing a stepwidth. Numerical error theory of finite differences \textsuperscript{4} states that stepwidths too large generally provide a poor approximation. Conversely, an approximation with a stepwidth too small accumulates large round-off errors and also provides a poor approximation. The optimal stepwidth is not known a-priori (without a reference value this is also true a-posteriori). For our application it is also important to note that the optimal stepwidth depends on the concrete configuration used.

In Figure 1(a) (blue line) we compare the relative difference \( \frac{\nabla_{\text{FD}}(r) - \nabla_{\text{AD}}(r)}{\nabla_{\text{AD}}(r)} \) of sensitivity calculated by AD and FD for a single material parameter by varying the stepwidth, in Figure 1(a) (red line) we do the same for a different electrode ring configuration. Note that by choosing a normed stepwidth size of, i.e., \( \gamma = \frac{x + h}{x} - 1 = \frac{h}{x} = 5.995 \cdot 10^{-5} \) in the first case gives a small relative difference of 4.634%. In the second case, however, it does not. The relative difference is now 39.76%.

This is very troubling for the optimisation of sensitivity with the electrode configuration as optimisation variables: The relative differences are of the same magnitude as some of the achieved gains by optimisation in \textsuperscript{1}. Even more: the goal of the optimisation is to increase overall sensitivity. This may also be interpreted as increasing a measure of nonlinearity. In this case we expect finite differences generally to yield increasingly poorer approximations. Finite differences should be used with caution in this context.

\footnote{1}http://beltoforion.de/article.php?an=muparser
\footnote{2}As AD methods usually yield derivatives with working accuracy we may carefully refer to the relative differences as relative errors.
5 Optimisation

In \cite{1} it was shown that the set of feasible electrode configurations $\mathcal{R}$ may be parametrised via the ring radii. This leads to the linearly constrained optimisation problem

$$\max_{r \in \mathcal{R}} J(r) \quad \text{s.t.} \quad Ar \leq b$$

Some preliminary results are shown in Figure 1(b). We expect full optimisation results to be finished for the conference.

6 Conclusion and Future Work

We have proven the feasibility of applying ADOL-C within CFS++ for derivative calculation for the evaluation of sensitivity of impedance with regard to material parameters and outlined difficulties we encountered with the application of ADOL-C. Furthermore, we discussed the influence of an optimal stepwidth on the sensitivity as a cost function, showing that finite differences are very problematic in this context. This was only possible with the use of algorithmic differentiation. The authors also show some optimisation of the electrode configuration.

In future we will explore ways to fix the issue with reusing the tape in the timestepping loop. We also plan to further decrease the runtime of our simulation by optimising the code specifically for AD, thus reducing the computational cost for each optimisation step. Furthermore, we plan to allow more general electrode configurations which are not restricted to ring geometries. Shape optimisation could then be applied to increase the sensitivity even further.

7 Acknowledgements

The authors thank Manfred Kaltenbacher for making CFS++ available to us and Carsten Unverzagt for his preliminary work on calculating the impedances of triple-ring-electrodes on which these simulations are based.

References


Branch-locking AD techniques for composite nonsmooth functions and nonsmooth implicit functions

Kamil A. Khan*

April 2016

In the AD2012 conference, the first computationally tractable methods [1, 2] were presented for the computation of useful generalized derivatives of composite nonsmooth functions, for employment in numerical methods for nonsmooth equation-solving and optimization methods. Each method is a variant of the vector forward AD mode, based on an earlier forward AD mode by Griewank that computes directional derivatives of compositions of absolute-value functions and smooth functions [3]. Since AD2012, these methods have been improved and extended substantially. An improved vector forward AD mode [4] combines the benefits of both approaches presented at AD2012, the computed generalized derivatives were found [5, 4] to be Nesterov’s lexicographic (L-)derivatives [6], and the original theory and methods were extended to inverse and implicit functions [7, 8] and to the solutions of parametric ordinary differential equations [9]. A piecewise linearization method was developed [5] to construct useful piecewise affine approximations that are expressed in a canonical abs-normal form, for use in dedicated numerical methods [10, 11].

This presentation explores several techniques to improve the computational performance of L-derivative evaluation for compositions of known nonsmooth functions, and for local implicit functions defined in terms of a known composite nonsmooth residual function. These techniques use partial forward AD and reverse AD probing sweeps to glean enough information to replace the considered function with a corresponding smooth branch-locked variant that can be differentiated by standard AD to produce the desired L-derivative. Proofs for the presented theoretical claims are straightforward extensions of intermediate results in [4, 7], and are omitted for brevity.

1 Background and notation

This section describes the considered class of functions and briefly outlines relevant concepts concerning generalized derivatives.

Nesterov’s lexicographic (L-)derivatives [6] are generalized derivatives for a broad class of nonsmooth, locally Lipschitz continuous functions that have been shown [6, 9] to provide useful first-order sensitivity information to methods for nonsmooth equation-solving (e.g. [12]) and optimization that are based on Clarke’s generalized Jacobian [13]. L-derivatives are defined for the class of L-smooth functions, which is closed under composition and includes all C^1 functions, all convex functions including the Euclidean norm, and all functions that are piecewise differentiable in the sense of Scholtes [14], including optimal value functions for linear programs: z(b) := min(c^Tx : Ax = b, x ≥ 0). In this abstract, L-smooth functions are referred to as simply nonsmooth.

Lexicographic-directional (LD-)derivatives [4] are variants of L-derivatives that obey a chain rule that resembles the vector forward AD mode. For a C^1 function ϕ : R^n → R, some u ∈ R^n, and some matrix ˆU ∈ R^{n×p}, the LD-derivative of ϕ at u in the directions ˆU is ϕ'(u; ˆU) = ϕ(u) ˆU. The absolute-value function satisfies abs'(u; ˆU) = fsign(u, ˆu_1, ..., ˆu_p) ˆU, which uses the first-sign function [5]:

\[ fsign(z) := \begin{cases} 0 & \text{if } z = 0, \\ \text{sign } z_k^* & \text{otherwise}, \text{where } k^* := \min\{k : z_k \neq 0\}. \]

Other nonsmooth elemental functions may be handled similarly. If ˆU is chosen to be square and nonsingular, then the L-derivative of a nonsmooth function ϕ at u in the directions ˆU is ϕ'(u; ˆU) ˆU^{-1}.

This presentation considers composite functions F : X ⊂ R^n → R^m as described as in Section 2.2 of [15], where y := F(x) is evaluated by a finite nonrecursive procedure:

\[ \begin{align*}
    v_{i−n} &:= x_i \quad \text{for } i = 1, \ldots, n, \\
    v_i &:= \varphi_i(v_{j≺i}) \quad \text{for } i = 1, \ldots, ℓ, \\
    y_{m−i} &:= u_{i−i} \quad \text{for } i = m − 1, \ldots, 0,
\end{align*} \]

with the elemental functions ϕ_i chosen from a library that includes the usual class of differentiable elemental functions, as well as certain nonsmooth elemental functions.

*Mathematics and Computer Science Division, Argonne National Laboratory, kkhan@anl.gov
The vector forward AD mode for nonsmooth functions \[4\] evaluates LD-derivatives \[\dot{Y} := F'(x; \dot{X})\] for \(F\) at \(x \in \mathbf{X}\) in directions \(\dot{X} \in \mathbb{R}^{n \times p}\) using the following procedure:

\[
\begin{align*}
\dot{V}_{i-n} & := \dot{X}_i & \text{for } i = 1, \ldots, n, \\
\dot{V}_i & := \varphi'((v_j)_{j<i}; (\dot{V}_j)_{j<i}) & \text{for } i = 1, \ldots, \ell, \\
\dot{Y}_{m-i} & := \dot{V}_{i-1} & \text{for } i = m - 1, \ldots, 0.
\end{align*}
\]

When each \(\varphi_i\) is differentiable at \((v_j)_{j<i}\), this procedure coincides with the established vector forward AD mode, and \(F'(x; \dot{X}) = F'(x) \dot{X}\). When \(\dot{X}\) has only one column, then \(F'(x; \dot{X})\) is the classical directional derivative.

In some applications, such as evaluation of L-derivatives or LD-derivatives for postcompositions with functions with known generalized derivatives, for implicit functions \([7, 8]\) or for solutions of parametric ordinary differential equations \([9]\), the matrix \(\dot{X}\) in the above procedure is determined by an outer numerical method and cannot be varied. In other applications, such as in equation-solving methods \([16, 12]\) or bundle methods that require an arbitrary generalized derivative at each iteration, \(\dot{X}\) may be chosen to be any square nonsingular matrix of appropriate dimension; and may therefore be chosen to simplify evaluation of \(F'(x; \dot{X})\).

For simplicity, the remainder of this abstract assumes that the only permitted nonsmooth elemental is the absolute-value function; note, however, that the described techniques remain applicable when other nonsmooth elemental such as the Euclidean norm are employed. Under this simplifying assumption, any nonsmoothness in \(F\) is entirely due to certain absolute-value functions \(\varphi_i\). In this case, the abs-normal form of Griewank et al. \([10]\) may be deployed when carrying out the techniques in this presentation; this is avoided in the current abstract for generality, to suggest how other nonsmooth elemental may be treated in a similar manner.

A smooth branch of \(F\) is a variant \(\tilde{F}\) of \(F\) obtained by replacing each “abs” in the computational graph of \(F\) with one of its linear pieces: \(u \mapsto u \lor u \mapsto -u\). Let \(J \subset \{1, \ldots, \ell\}\) denote the set of indices \(i\) for which \(\varphi_i\equiv\text{abs}; \) \(F\) has \(2^{|J|}\) smooth branches. For each \(j \in J\), let \(u_j\) denote the intermediate variable \(v_i\) for which \(i < j\). An absolute-value function \(\varphi_i\) for \(i \in J\) is simply switched if there is no \(j \in J\) for which \(j < i\); the function \(\tilde{F}\) is simply switched if \(\varphi_i\) is simply switched for all \(i \in J\) \([5]\).

Assume that \(F\) is written so that no absolute-value functions are known a priori to have arguments \(u_i\) that are identically zero; these may nevertheless be handled by the input gradient probes introduced in this presentation.

## 2 Branch-locking for composite nonsmooth functions
Griewank \([3]\) shows that there cannot exist a reverse AD mode for nonsmooth functions that is consistent with the identity \(\tilde{y}^T \tilde{y} \equiv x^T \dot{x}\) given by \([15, \text{Equation 3.4}]\). Nevertheless, observe that \(\text{fsign}(0) = 0\), and that \(\text{fsign}(z) \in \{-1, +1\}\) when \(z \neq 0\). It follows that, for each \(x\) and \(\dot{X}\), there is a smooth branch \(F\) of \(F\) for which \(\dot{F}'(x; \dot{X}) = \dot{F}'(x) \dot{X}\). Thus, if \(\tilde{F}\) could be identified, then this LD-derivative could be computed using the standard vector forward AD mode. Moreover, if \(\dot{X}\) is square and nonsingular in this case, then the L-derivative \(\dot{F}'(x; \dot{X}) \dot{X}^{-1} = \dot{F}'(x)\) could be evaluated by applying the standard forward or reverse AD modes to \(\tilde{F}\); the reverse AD mode would be computationally cheaper when \(m \ll n\). This section briefly outlines branch-locking techniques to “lock” \(F\) into an appropriate smooth branch \(\tilde{F}\), which requires locking each absolute-value function in the computational graph of \(F\) into an appropriate linear branch \(u \mapsto u\) or \(u \mapsto -u\). (In the language of \([5]\), branch-locking corresponds to choosing an appropriate smooth signature \(\sigma\).) The techniques describe here will be extended to implicit functions in the next section, where the computational benefits are significantly greater.

The following techniques may be employed to probe for sufficient information to branch-lock:

- **Directional derivative probes**, in which branch-locking information is obtained during evaluation of a directional derivative according to \([3]\), and might be employed in a technique outlined in \([11]\),
- **Interval probes**, which use interval arithmetic to complement directional derivative probes, and are not discussed further for brevity, and
- **Input gradient probes**, involving partial sweeps of classical forward AD or reverse AD, in which the gradients of the inputs of simply switched absolute-value functions are used to branch-lock them.

Each of these techniques replaces \(F\) with a variant \(G\) in which some absolute-value functions have been branch-locked.

The following result is stated loosely, and is the key result underlying this presentation.

**Theorem 1.** The described probes may be applied repeatedly and in any order, branch-locking any possible absolute-value functions at each step, until all absolute-value functions are branch-locked. The resulting branch-locked composite function \(\tilde{F}\) is the appropriate smooth branch of \(F\).

**Proof.** Roughly, each probing operation replaces \(F\) with \(G \equiv F'(x; [\dot{Y} \ -])\) for some appropriate matrix \(\dot{Y}\); established definitions and properties concerning LD-derivatives \([4, 9]\) may then be employed.

Directional derivative probes and input gradient probes are outlined in the remainder of this section. In all cases, \(F\) may be branch-locked by performing one directional derivative probe followed by a series of input gradient probes;
when \( m \ll n \), the resulting reverse AD mode to determine LD-derivatives and L-derivatives of \( F \) will computationally outperform the established nonsmooth vector forward AD mode substantially when certain mild conditions on \( F \) are met. However, the structure of \( F \) may suggest a computationally advantageous alternative sequence of probing steps.

2.1 Directional derivative probes

Suppose that \( \hat{X} \in \mathbb{R}^{n \times p} \), pick a small lexicographic depth tolerance \( k \leq p \), and define \( \hat{Z} \) as the submatrix of \( \hat{X} \) comprising only the leftmost \( k \) columns of \( \hat{X} \). If \( \hat{X} \) may be varied freely, then set \( k := 1 \) and choose the vector \( \hat{z} := \hat{Z} \) randomly. Next, using nonsmooth vector forward AD, evaluate \( F'(x; \hat{Z}) \), for a computational cost that is bounded by \( \gamma \cdot \text{Cost}(F) \) for some small library-dependent constant \( \gamma \). Based on the definition of the “fsign” function, this directional derivative probe branch-locks any absolute-value function \( v := \varphi_j(u) \) in the computational graph of \( F \) for which \( \text{fsign}(u,\hat{u}_1,\ldots,\hat{u}_j) \in \{-1,+1\} \). If the leftmost column of \( \hat{X} \) is chosen randomly, then it is highly probable that the above condition is satisfied for all \( j \in J \), unless \( \nabla u_j(x) = 0 \) for some \( j \in J \). This is the case, for example, if \( u_j \equiv 0 \) in a neighborhood of \( x \). In this case, no directional derivative probe with \( k < p \) can branch-lock \( \varphi_j \); input gradient probes are developed to handle this situation.

2.2 Input gradient probes

Input gradient probes reliably branch-lock absolute-value functions that are simply switched. Since the computational graph of \( F \) is nonrecursive, Theorem 1 implies that the number of input gradient probes required to branch-lock \( F \) is bounded by the switching depth \((1 + \max_{j \in J} \{|j : j < i\}|)\).

If \( \hat{X} \) is fixed a priori, proceed roughly as follows. Choose \( j \in J \) corresponding to some simply switched absolute-value function. Thus, the function \( x \mapsto u_j(x) \in \mathbb{C}^1 \). Using the standard reverse AD mode, evaluate \( \nabla u_j(x); \) either smooth AD mode may also be used instead to determine \( \nabla u_j(x) \) for all simply-switched \( j \in J \) simultaneously. If \( \nabla u_j(x) = 0 \), then branch-lock the absolute-value function \( \varphi_j \) arbitrarily. Otherwise, let \( \xi \) denote the leftmost column of \( \hat{X} \) for which \( \delta_j := \langle \nabla u_j(x), \xi \rangle \neq 0 \) (define \( \delta_j := 0 \) if there is no such column), and branch-lock \( \varphi_j \) by replacing it with the linear branch \( z \mapsto (\text{sign}\, \delta_j) z \). Observe that any known structural information concerning \( \hat{X} \) may help to identify \( \xi \); for example, \( \xi \) is trivial to identify if \( \hat{X} \) is the identity matrix or a permutation matrix.

If \( \hat{X} \) may be varied, then choose \( \xi \) so that \( \langle \nabla u_j(x), \xi \rangle \neq 0 \) for several simply-switched \( j \) with \( \nabla u_j(x) \neq 0 \) and add \( \xi \) as the first column of \( \hat{X} \).

3 Branch-locking for inverse and implicit functions

Here the techniques of the previous section are extended to compute generalized derivatives for a local implicit function \( H \), defined so that

\[
0 \equiv F(x, H(x)),
\]

where \( F \) is a known composite nonsmooth function that satisfies Clarke’s implicit function theorem for nonsmooth functions [13, Theorem 7.1]. Local inverse functions may be treated similarly, with \( F : (x, y) \mapsto x - G(y) \), in which case \( H \equiv G^{-1} \). As shown in previous work [7], the unknown implicit function \( H \) is also L-smooth. Moreover, LD-derivatives for \( H \) may be evaluated; if \( F(x, z) = 0 \), then \( H'(x; \hat{X}) \) is the unique solution \( N \) of the equation system

\[
0 = F'((x, z); (\hat{X}, N)).
\]

This equation system is difficult to solve, since the right-hand side may be discontinuous with respect to \( N \). The most direct way to determine \( N \) is to decompose (2) columnwise, and then solve the systems determining the individual columns of \( N \) from left to right. When (2) is approached this way, the equation determining each column of \( N \) is nonsmooth but continuous. However, this procedure involves invoking a nonsmooth equation-solving method once per column of \( N \), which is computationally costly.

By similar logic to the previous section, there exists a smooth branch \( \hat{F} \) of \( F \) for which

\[
0 = \hat{F}'((x, z); (\hat{X}, N)) = \frac{\partial \hat{F}}{\partial x}(x, z) \hat{X} + \frac{\partial \hat{F}}{\partial z}(x, z) N,
\]

and so

\[
H'(x; \hat{X}) = -\left(\frac{\partial \hat{F}}{\partial x}(x, z)\right)^{-1} \frac{\partial \hat{F}}{\partial z}(x, z) \hat{X},
\]

which may be evaluated using standard AD. Again, all that remains is to branch-lock \( F \) correctly to determine an appropriate smooth branch \( \hat{F} \). Suppose that (1) has been solved to determine \( \hat{z} := H(x) \).

Certain preprocessing steps can save significant computational effort. Partition (1) as much as possible, whenever (1) and any known sparsity pattern for \( F \) suggest that some components of \( H \) can be evaluated independently of the remaining components of \( H \). Treat each partition individually with (2); in each partition, remove unused components of \( x \) and corresponding rows of \( \hat{X} \). If the leftmost column of the resulting matrix \( \hat{X} \) is the zero vector, then the same is true of the rows of \( N \) corresponding to the components of \( \hat{z} \) considered in the current partition, and one may then
proceed to the next columns of $\dot{X}$ and $N$. Repeating this step as necessary, one may assume the leftmost column of $\dot{X}$ is nonzero.

In each partition, as earlier, let $\dot{x}$ denote the leftmost column of $\dot{X}$, and let $\dot{n}$ denote the leftmost column of $N$. If the absolute-value functions in $F$ do not depend on $z$, then $z$ and $\dot{n}$ do not affect the choice of $\tilde{F}$; thus, apply the results of the previous section to branch-lock $F$ exactly as when evaluating $F'((x,z);(X,0))$. Otherwise, using a nonsmooth equation-solving method, solve the equation system $0 = F'((x,z);(\dot{x},\dot{n}))$ for $\dot{n}$; a directional derivative probe may then be applied with dimension $k = 1$. Directional derivative probes for $k \geq 2$ involve solving similar equation systems; hence, at this point, input gradient probes become advantageous.

4 Conclusions and further work

Several probing techniques are presented, which use partial forward AD and reverse AD sweeps to replace a nonsmooth composite function $F$ with an appropriate smooth branch-locked variant $\tilde{F}$, so that standard AD methods may then be used to evaluate a generalized derivative of $F$ as a classical derivative of $\tilde{F}$. The developed techniques were extended to nonsmooth implicit functions, for which existing methods to evaluate generalized derivatives require computationally expensive equation-solving methods. The techniques outlined here have been applied by hand to several small examples; these will be presented in September, along with results of a C++ implementation.

Acknowledgement

This work was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References


Simple implementation and examples for piecewise linearization with abs-normal form

Koichi Kubota*

March 2016

1 Introduction

The aim of this report is to describe simple implementation with C++ for piecewise linearization[1, 2, 3] and to show some examples on piecewise linearization and checking local optimality. From the implementation point of view, most AD tools can treat the derivative of $|x|$ at $x \neq 0$, but can not do the derivative at $x = 0$.

Here we focused on the following three features: (1) computation of piecewise linearization with abs-normal form (ANF) [1, 2], (2) reverse accumulation of derivatives, (3) checking the local optimality with GLPK[4]. More sophisticated implementation techniques are omitted here but we point out several important situations on practical piecewise linearizations. ANF can treat the case $|x|$ at $x = 0$ systematically by means of the directional derivatives.

1.1 Abs-normal form

ANF is introduced by A. Griewank[1, 2] as follows. Given a function $f(x)$ ($f: R^n \rightarrow R^m$) its derivatives $\partial f_i / \partial x_j$ ($1 \leq i \leq m$, $1 \leq j \leq n$) are computed straight forwardly with AD[5, 6].

We assume that there are $s$ execution of abs() operations whose values are equal to zero in the computational process [6] of $y = f(x)$. We can rewrite $y = f(x)$ with $u = g(x, v)$, $v = \text{abs}(u)$, $y = h(x, v)$, where $u \in R^n$ indicates the argument vector of the $s$ execution of abs(), $v \in R^s$ indicates the value vector of the abs($u$), $h: R^{n+s} \rightarrow R^m$ and $g: R^{n+s} \rightarrow R^n$ are appropriately defined functions as well as the vector valued abs(): $R^s \rightarrow R^n$ is the componentwise absolute operation. Now we can compute the directed derivatives with AD[1, 2]:

$$\Delta u = \frac{\partial g}{\partial x} \Delta x + \frac{\partial g}{\partial v} \Delta v, \quad \Delta y = \frac{\partial h}{\partial x} \Delta x + \frac{\partial h}{\partial v} \Delta v$$

and

$$\Delta v = \text{sign}(u + \Delta u) \cdot \Delta u + (\text{sign}(u + \Delta u) - \text{sign}(u)) \cdot u.$$

Note that when we give a value of direction $\Delta x$, we get the values of $\Delta u$, $\Delta v$ and $\Delta y$. Moreover, when we assume the sign of $\Delta u$ firstly, we get the coefficients of $\Delta v$ and $\Delta y$ with respect to linear combinations of $\Delta x = (\Delta x_1, \ldots, \Delta x_n)$.

2 Implementation notes

We check the above concepts with c++ operator overload program 1 for generating computational graph G[5, 6]. We assume that hereafter the function $f$ is a scalar function $R^n \rightarrow R$ (i.e., $m = 1$) for minimization (or maximization).

2.1 Listing up the abs operations with value zero

The nodes corresponding to the values of abs operations whose results are zero are listed when the value of $f$ is computed. We rename the nodes to $v_1, \ldots, v_s$, and their argument nodes to $u_1, \ldots, u_s$, correspondingly. Since the computational graph is a DAG, we can assume that $v_i$ is computed before $v_j$ for $i < j$.

2.2 Computation of derivatives

The partial derivatives $\partial g / \partial x$, $\partial g / \partial v$, $\partial h / \partial x$, $\partial h / \partial v$ are computed as follows.

(i) Make a computational graph of $f$ as well as the sets of renamed nodes, i.e., $V = \{v_1, \ldots, v_s\}$ and $U = \{u_1, \ldots, u_s\}$.

(ii) For each node $v$ in $V$, change the node type of $v$ from the type that is the result of abs() operation to the type that is the new independent variable.

(iii) Compute $\partial g_k / \partial x_j$ ($j = 1, \ldots, n$) and $\partial g_k / \partial v_\ell$ ($\ell = 1, \ldots, s$) by reverse mode for all $u_k \in U$ ($k = 1, \ldots, s$). That is, after the topological sort with the depth first search from $u_k$, the adjoint values are computed for all $w < u_k \in G$.

*Corresponding Author, Chuo University, kubota@ise.chuo-u.ac.jp
1https://github.com/numlib/yaad-pl
(iv) Compute $\partial h/\partial x_j$ $(j = 1, \ldots, n)$ and $\partial h/\partial v_\ell$ $(\ell = 1, \ldots, s)$ by reverse mode from the node corresponding to $y = f(x)$.

Since $\Delta u = (\partial g/\partial x) \Delta x + (\partial g/\partial v) \Delta v$ and $\Delta v = \Sigma \Delta u$, where $\Sigma \equiv \text{diag}(\sigma_1, \ldots, \sigma_s) \equiv \text{diag}(\text{sign}(u_1 + \Delta u_1), \ldots, \text{sign}(u_s + \Delta u_s))$ ($\forall u_i \in U$), we can eliminate $\Delta v$ and get the explicit form $\Delta u$ with respect to $\Delta x$, that is, $\Delta u = (I - (\partial g/\partial v) \Sigma)^{-1} (\partial g/\partial x) \Delta x$. Thus the directed derivative $\Delta y = (\partial f/\partial x) \Delta x$ is computed by

$$\Delta y = (\partial h/\partial x) \Delta x + (\partial h/\partial v) \Sigma (I - (\partial g/\partial v) \Sigma)^{-1} (\partial g/\partial x) \Delta x.$$  \hspace{1cm} (1)

Note that each value of $u \in U$ is zero (this is the reason of the member of $U$) so that $\Sigma \equiv \text{diag}(\text{sign}(\Delta u_1), \ldots, \text{sign}(\Delta u_s))$.

2.3 Local minimization

One of key advantages of the abs-normal form is that the value of a directional derivative $\Delta y$ can be computed without the value of the direction $\Delta x$ itself as follows. (i) Fix a combination of the diagonal values of $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_s) = \text{diag}(\pm1, \ldots, \pm1)$. (ii) Compute $\Delta y$ with eq.(1). Here, we can get explicit form of $\Delta y$ (or the coefficients of the linear combination) with respect to $\Delta x_j$ without the value of $\Delta x_j$ $(j = 1, \ldots, n)$. The coefficients of $\Delta x_1, \ldots, \Delta x_n$ of the above $\Delta y$ can give one of the values of the generalized gradients.

When the value of the $\Delta x$ is needed in order to make $\Delta y$ being really one of the generalized gradients, we should solve the corresponding linear programming: (iii) Check the direction $\Delta x$ that gives the directed derivative $\Delta y$ by feasibility of $A\Delta x \geq 0$ with LP, where the $s \times n$ matrix $A$ whose $\ell$th row is equal to the coefficients of $\Delta u_\ell$ (if $\sigma_\ell = 1$) or $-\Delta u_\ell$ (if $\sigma_\ell = -1$) in the explicit form with respect to $\Delta x$.

Finally, for each matrix $A$ corresponding to all the possible diagonal matrices $\Sigma$, we can check the infeasibility of $\Delta y < 0$ and $A\Delta x \geq 0$. When there are no feasible solutions the given point $x$ is locally minimum.

2.3.1 Branch and bound search

Since the number of combinations of the sign of the diagonal values of the matrix $\Sigma$ is $|\{1, -1\}|^n = 2^n$, the number of solving LP is at most $2^n$. But, there are some cases where the number is drastically reduced with the branch and bound technique or pruning the search tree.

Let $A$ denote the $s \times n$ coefficient matrix of LP and $\tilde{A}$ denote $k \times n$ coefficient submatrix of $A$ $(k \leq s)$. When $A\Delta x \geq 0$ is feasible any subsystem of $A\Delta x \geq 0$ should be feasible, and when $A\Delta x \geq 0$ is infeasible any extended system of $A\Delta x \geq 0$ should be infeasible.

We can make the coefficient matrix $A$ from the first row to $s$th row with step by step as follows.

$\Delta u_1$ is equal to $(\partial g_1/\partial x)$. $\Delta v_1$ is represented as $\Delta v_1 = \sigma_1 \Delta u_1$, where $\sigma_1 = 1$ or $\sigma_1 = -1$. $\Delta u_2$ is equal to $(\partial g_2/\partial x) + (\partial g_2/\partial v_1) \cdot \Delta v_1$. Note that $u_2$ may depend on $v_1$ but it is independent to $v_i$ $(i = 2, \ldots, s)$ since the computational graph is directed acyclic graph. Thus, $\Delta u_2 = (\partial g_2/\partial x) + (\partial g_2/\partial v_1) \cdot \sigma_1 \Delta u_1$. Similarly, we have

$$\Delta u_k = (\partial g_k/\partial x) + \sum_{j=1}^{k-1} (\partial g_k/\partial v_j) \cdot \sigma_j \cdot \Delta u_j \quad (k = 1, \ldots, s).$$

After the computation of $\Delta u_k$ in the explicit form with respect to $\Delta x$ under the selected sign combination of $\sigma_1, \ldots, \sigma_k$, we have $\Delta u_1, \ldots, \Delta u_k$ and can check the feasibility of $A^{(k)} \Delta x \geq 0$, where the coefficient matrix is defined by $A^{(k)} \equiv \begin{pmatrix} \sigma_1 \Delta u_1 \\ \vdots \\ \sigma_k \Delta u_k \end{pmatrix}$. When $A^{(k)} \Delta x \geq 0$ is feasible, we should check $A^{(k+1)} \Delta x \geq 0$ is feasible or infeasible for $\sigma_{k+1} = 1$ and $\sigma_{k+1} = -1$. When it is infeasible, there are no feasible solution with the current $\sigma_1, \ldots, \sigma_k$.

3 Examples

These examples are piecewise linear functions whereas the important point we focused here is the same as that of the general piecewise smooth functions.

3.1 Simple case

For a function $f(x_1, x_2) = \min(|x_1| + |x_2|, |x_1 - x_2|)$, the computational process (Table 1) gives the value of $f(x_1, x_2) = f(0, 0)$, where $\min(a, b)$ is rewritten[1] by $(a + b - |a - b|) \cdot 0.5$.

We rewrite the computational process for $y = f(x_1, x_2)$ into the form: $u_1 = x_1 + x_2$, $v_1 = \text{abs}(u_1)$, $u_2 = x_1 - x_2$, $v_2 = \text{abs}(u_2)$, $u_3 = v_1 - v_2$, $v_3 = \text{abs}(u_3)$, $y = (v_1 + v_2 - v_3) \cdot 0.5$, where $u_1 = w_1$, $v_1 = w_2$, $u_2 = w_3$, $v_2 = w_4$, $u_3 = w_5$, $v_3 = w_6$, $w_7 = w_7$, and $y = w_8$. With the forward AD technique, we have $\Delta u_1 = \Delta x_1 + \Delta x_2$, $\Delta u_2 = \Delta x_1 - \Delta x_2$, $\Delta u_3 = \Delta v_1 - \Delta v_2$, and $\Delta y = (\Delta v_1 + \Delta v_2 - \Delta v_3) \cdot 0.5$.

Now we have $u = g(x, v)$, $y = h(x, v)$, where $x = (x_1, x_2) = (0, 0)$.
Here we consider the case in which $\Delta u_3 = \Delta = \Delta u_1 = \Delta u_2$, we have $\Delta u_1 = \Delta u_2 = \Delta u_3 = \Delta u_3$ and $\sigma_1 = 1$, $\sigma_2 = 1$, $\sigma_3 = 1$. In this case, all the conditions $\Delta u_1 = \Delta x_1 + \Delta x_2 > 0$, $\Delta u_2 = \Delta x_1 - \Delta x_2 > 0$, $\Delta u_3 = \Delta x_1 - \Delta x_2 = \Delta u_1 - \Delta u_2 = 2\Delta x_2 > 0$ hold in the domain defined by $\Delta x_1 > \Delta x_2 > 0$ (the region (1) in Figure 1(b)). Now we have the explicit form of $\Delta y$ with respect to $\Delta x_1$ and $\Delta x_2$:

$$
\Delta y = ((\partial h/\partial x) + (\partial h/\partial v)(I - (\partial g/\partial v)\Sigma)^{-1}(\partial g/\partial x))\Delta x
$$

$$
= \left( (\partial h/\partial x) + (\partial h/\partial v) \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - (\partial g/\partial v) \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix} \right)^{-1} (\partial g/\partial x) \Delta x
$$

$$
= \begin{pmatrix} 0 & 0 & 0.5 & -0.5 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & -1 & 1 & 0 \end{pmatrix} \Delta x = \begin{pmatrix} 1 & -1 \end{pmatrix} \cdot \Delta x = \Delta x_1 - \Delta x_2.
$$

Similarly, we have the domains and the generalized gradients at (0,0) in Table 2.

![Diagram](image.png)

**Table 2: The generalized gradient $\Delta y$**

<table>
<thead>
<tr>
<th>No.</th>
<th>$\Delta u_1$</th>
<th>$\Delta u_2$</th>
<th>$\Delta u_3$</th>
<th>$\sigma_1$</th>
<th>$\sigma_2$</th>
<th>$\sigma_3$</th>
<th>Domain</th>
<th>$\Delta y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\Delta x_1 &gt; \Delta x_2 &gt; 0$</td>
<td>$\Delta x_1 - \Delta x_2$</td>
</tr>
<tr>
<td>(2)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$-1$</td>
<td>1</td>
<td>1</td>
<td>$0 &gt; \Delta x_1 &gt; \Delta x_2$</td>
<td>$\Delta x_1 - \Delta x_2$</td>
</tr>
<tr>
<td>(3)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>1</td>
<td>$-1$</td>
<td>1</td>
<td>$0 &lt; \Delta x_1 &lt; \Delta x_2$</td>
<td>$-\Delta x_1 + \Delta x_2$</td>
</tr>
<tr>
<td>(4)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>1</td>
<td>$\Delta x_1 &lt; \Delta x_2 &lt; 0$</td>
<td>$-\Delta x_1 + \Delta x_2$</td>
</tr>
<tr>
<td>(5)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>1</td>
<td>1</td>
<td>$-1$</td>
<td>$\Delta x_1 &gt; -\Delta x_2 &gt; 0$</td>
<td>$\Delta x_1 + \Delta x_2$</td>
</tr>
<tr>
<td>(6)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$-1$</td>
<td>1</td>
<td>$-1$</td>
<td>$0 &lt; \Delta x_1 &lt; -\Delta x_2$</td>
<td>$-\Delta x_1 - \Delta x_2$</td>
</tr>
<tr>
<td>(7)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>1</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$0 &lt; -\Delta x_1 &lt; \Delta x_2$</td>
<td>$\Delta x_1 + \Delta x_2$</td>
</tr>
<tr>
<td>(8)</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$&gt;0$</td>
<td>$-1$</td>
<td>1</td>
<td>$-1$</td>
<td>$-\Delta x_1 &gt; \Delta x_2 &gt; 0$</td>
<td>$-\Delta x_1 - \Delta x_2$</td>
</tr>
</tbody>
</table>

### 3.2 Max-min of many planes

Here we consider on a two dimensional scalar valued function $f(x_1, x_2)$ is defined with 2n planes as

$$
f(x_1, x_2) \equiv \max_{0 \leq \ell < n} \min(a_{2\ell}x_1 + b_{2\ell}x_2, a_{2\ell+1}x_1 + b_{2\ell+1}x_2).
$$
The $k$th plane $a_kx_1 + b_kx_2$ ($k = 0, \ldots, 2n - 1$) is defined by three points $(p_k, q_k, r_k)$, $(p_{k+1}, q_{k+1}, r_{k+1})$ and $(0, 0, 0)$, where $(p_k, q_k) = (\cos(\frac{\pi}{n}k), \sin(\frac{\pi}{n}k))$ and arbitrary given $r_k ((p_{2n}, q_{2n}, r_{2n}) \equiv (p_0, q_0, r_0))$.

### 3.2.1 Local minimal point $n = 32$ (64 planes, $s = 63$)

When $n = 32$, $r_{2k} = 0.3$, $r_{2k+1} = 1.0$ ($k = 0, \ldots, 31$), the function has locally minimal point at $(0, 0)$. In this case, there are 63 absolute operations the values (of the arguments) of which are all zero. So, the required number of solving LP is $2^{63}$ with naive implementation. But, the total number of solving LP is only 10340 with the above branch and bound search and the computational time is about 7 seconds (Ubuntu14.04LTS, VMware Fusion 7.1.3, Macbook pro core i7). Note that $L^5 \neq 0$ but $L^6 = 0$ where $L \equiv (\partial g/\partial u)$ in this example.

### 3.2.2 Not local minimal point $n = 16$ (32 planes, $s = 31$)

When $n = 16$, $r_{26} = -0.1$, $r_{2k} = 0.3$ ($k = 0, \ldots, 12, 14, 15$), and $r_{2k+1} = 1.0$ ($k = 0, \ldots, 15$), the function is depicted in figure 2. The origin $(0, 0)$ is not locally minimal so that there are directions along with which the value of $f$ is decreased.

We found the directions for decreasing the value of $f$ by solving LP 2222 times with the above branch and bound. There are two optimal solutions and the other are infeasible solutions in 2222 solutions whereas the number of all the combinations is $2^{31}$.

Figure 2: Origin is not local minimal point

### 4 Conclusions and further work

We gave the simple implementation of piecewise linearization with “abs-normal-form” in C++ and an efficient way to check the local minimal point with branch and bound technique, where two dimensional examples that may have $2n$ planes were shown. The size of the corresponding strictly lower matrix $L \equiv \partial g/\partial u$ is $(2n - 1) \times (2n - 1)$ (§3.2).

More practical experiments and the investigation of the optimal topological order for the branch and bound are needed in future work as well as the investigation of the higher derivatives with absolute operations and effects of numerical computational errors.

Acknowledgements: The author thanks Prof. Andreas Griewank for personal conversation after his talk presented in Tokyo October 2015, and Mr. Shogo Kamei who implemented a forward AD prototype tool for piecewise linear function as his bachelor thesis.

References


1 Introduction

Scientific applications are usually written in a single language such as C, C++, or a flavor of Fortran. Various algorithmic differentiation (AD) tools exist to differentiate these applications by using source transformation [1, 2, 3, 4] or operator-overloading [5, 6]. The language that an application is written in often dictates the approach and tool to be used for differentiation. Additionally, performance considerations, tool philosophy (recomputation vs. storage), and specific tool capabilities such as support for sparse Jacobians or fixed-point iterations may play a role in the tool that is ultimately used to differentiate an application.

Operator overloading and source transformation have their own strengths and weaknesses. The most important feature of operator-overloading-based AD tools is that they can be used in software regardless of its design complexity. Many large simulation tools (e.g., ISSM [7], SU2 [8, 9]) have been successfully differentiated in this manner. One drawback, however, is the large amount of memory required to store the tool-specific internal representation of the computation in order to run it in the reverse mode. The execution speed also suffers because of low compiler optimization potential. Both problems do not occur in source transformation tools. However, such tools cannot handle runtime features of C++ such as object inheritance, polymorphism, and templating.

Some scientific applications are coded in C++ but contain portions that are in C or are C like. Often, the global structure of the application is complex and requires the use of advanced C++ features, but the computationally intensive portions are C functions. In our previous work [10] we successfully demonstrated the interfacing of an application differentiated mainly by using operator overloading with a library that has been differentiated using source transformation. The implementation relies on the externally differentiated function feature of ADOL-C, where such functions have actually been differentiated by using ADIC. The interface works also well with Tapenade. We demonstrated that this mixed approach is able to amortize the memory requirement for the calculation of adjoints and Jacobians on two applications. By using both approaches in the same application wherever they are applicable and well suited, we were able to use the strengths of both approaches.

In this work we demonstrate the inverse interface. Here, the global application structure is a straightforward pure C (or Fortran) implementation; however, certain library calls may internally use complicated C++ features. We have created an entirely new interface keeping in mind the features of both ADOL-C and ADIC to support this setup.

2 Interfacing

When the C++ library is differentiated by using ADOL-C, an ADOL-C–specific internal representation of the computation, called a trace, will be generated. The interpretation of such a trace with the drivers provided in the ADOL-C library results in the computation of the derivatives for this function. To interface a C++ function with an external application that is differentiated by ADIC, we annotate the C++ function in the library and use a preprocessor to generate the helper codes listed below.

- ADIC stub containing the same activity patterns as the C++ function
- Interfacing function that replaces the differentiated-stub
- Trace creation source and header

The ADIC stub is used in the outer code to allow ADIC to have access to a complete code base. By keeping the activity patterns consistent, ADIC creates an appropriate differentiated stub. However, this differentiated stub must be replaced in ADIC’s output code with an interfacing function that contains calls to appropriate ADOL-C drivers.

Additionally, the trace creation source/header files are used to create the trace for the C++ function before computing the derivatives. This process occurs when the interfacing function is executed for the first time. This is true even if the outer code contains multiple calls to the C++ function, since ADOL-C traces are reusable at different evaluation points. An exception to this rule is when an ADOL-C branch switch warning occurs that requires setting up the trace again at an appropriate evaluation point.

*Kshitij Kulshreshtha* and *Sri Hari Krishna Narayanan*†

March 2016
As is the norm in differentiating a library with ADOL-C, the source code of the C++ function must be manually instrumented for using operator overloading with ADOL-C. The ADIC-generated files and preprocessor-generated files are compiled and appropriately linked to the ADOL-C differentiated library. In this setup, the interfacing function copies primal data from ADIC’s buffers, as well as pointers to tangent or adjoint data from ADIC’s buffers, and passes them to the ADOL-C driver function for the appropriate mode of differentiation. The primal data is then copied back into ADIC’s variables. The output adjoint/tangent data is implicitly written into ADIC’s buffers because of the use of pointers.

The workflow for the C++ library developer involves instrumentating the library as usual for differentiation by ADOL-C. Additionally the developer must write annotations for each API function and run the preprocessor to generate files for each of them. The developer of the C application must use the stub function to the differentiate application using ADIC. The differentiated stub must then be discarded. The build process for the application must be modified to additionally compile and link the preprocessor-generated files.

### 3 Annotations and Preprocessing

The annotation used in the C++ library is itself a description of the arguments of the C++ function, in that it specifies the active input and output variables, as well as the inactive indices, sizes, counters, and so forth. The input and output variables can be scalars, vectors, matrices, or slices of vectors or matrices, or even lower, or upper triangular matrices, or slices thereof. The annotation describes the dimensions associated with each variable name as well as its position in the formal argument list. This annotation is written by using Python syntax inside a specialized C++ comment. It is therefore ignored by a C/C++ compiler, and the Python-based preprocessor does not bother with the actual C/C++ code. One such annotation for a simple case can be seen in Fig. 1.

```plaintext
/*@
stad_export interface
name = 'k_eval'
iarr = [ ("n", 2), ("m", 4), ("j", 6) ]
input = [ ("y", [ "iArr[0]" ], 1), ("u", [ "iArr[1]" ], 3) ]
output = [ ("k", [ "s", "iArr[2]", "iArr[2]+1", 0, "iArr[0]" ], 5 ) ]
@*/

void k_eval(double *y, int n, double *u, int m, double **k, int j)
```

Figure 1: Annotations describing input and output variables with their dimensions

Writing the annotation is intuitive for anyone familiar with the semantics of the API function being annotated. The annotation is written as a list of tuples for each of input, output, and integer data (iArr). The tuple consists of the formal argument name, its dimension if it is an active input or output, and its position in the formal parameter list. For inactive integers the dimension is not required because they are all scalars. The numbers in the list of inactive integers may be referenced to provide the dimensions of the input or output variables by indexing them starting at 0. For example, in Fig. 1 iArr[0] is a reference to the parameter named n mentioned as the first element of the iArr list. Dimensions may also be given in terms of global variables or expressions. The dimension itself is a list of length up to 6. Length 0 implies scalars, length 1 vectors, and length 2 matrices. Additionally, one may specify slices of vectors and matrices by putting the character “s” as the first element of the dimension list, followed by the first index and past-end-of-slice index for each dimension respectively. Upper or lower triangular matrices may be specified by putting the character “u” or “l” at the end of the dimension list.

In order to generate C code for the interfacing function described in the preceding section, as well as the code to set up the trace properly and a stub to be processed by ADIC, the annotation is extracted and processed by a Python-based preprocessor script. Processing the information about the formal argument names, their dimensions, and the position in the parameter list allows one to use a generic loop structure to copy required data from one data structure to another and call either the ADOL-C–instrumented API function or the driver functions of ADOL-C in an application-agnostic way. The preprocessor relies mainly on regular expressions and string concatenation in Python. Only the portion between /*@ and @*/ is parsed in Python. The skeleton of the interface code generated from the annotation in Fig. 1 is shown in Fig. 2.

### 4 Validation

We have tested the mixed approach on a simple scalable test case of moderate code size modeling a periodic adsorption process used in an optimal control setting. This code is used to compute the derivatives required by the optimization algorithm, namely, gradients and Jacobians.

The periodic adsorption process was studied from an optimization point of view in [11, 12]. A system of PDAEs in time and space with periodic boundary conditions models the cyclic steady state of a process, where a fluid is preferentially absorbed on the surface of a sorbent bed. This leads to dense Jacobians that dominate the computation time (see [11]). Therefore, previous works have used inexact Jacobians (for example, [12]). Using AD, however, we
compute the equality and inequality constraint Jacobians as well as the objective gradient exactly up to machine precision. The PDAE system is discretized in space by using a finite-volume approach, and the resulting system of ODEs is then integrated in time by using a Runge-Kutta method.

The code of the original application was written completely in C. This application has been separately differentiated by using both ADIC and ADOL-C in the past. Therefore, to test the approach, we arbitrarily divided the call graph of the application into two portions: one to be differentiated by ADIC and one to be differentiated by ADOL-C. We added annotations to the ADOL-C portion and used the preprocessor to generate the interfacing function, stub, and tape creation header and source. We then used ADIC to generate derivative code and modified the ADOL-C portion by transforming all the double variables to adoubles. We then modified the build process of the original application and generated an executable. The resulting values for the objective gradient and the equality and inequality constraint Jacobians matched the values computed by a purely ADOL-C application.

5 Conclusion and Future Work

We have developed a method to interface a Fortran or C application differentiated by source transformation AD with a C++ library differentiated by operator overloading using ADOL-C. The method requires the use of hand written annotations to automatically generate additional files. We have shown that the method works on a medium sized application. In the future, we would like to study the approach on a large scale application differentiated by ADIC or Tapenade that must be interfaced with a large library differentiated by ADOL-C. We would also like to study the exploitation of structure such as sparsity within the combined application in such a framework.

Acknowledgments. This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References


1 Introduction

R is a language and environment for statistical computing and graphics [1]. It currently is widely used in statistics and data mining. To obtain derivatives in R, one can use several non-native approaches, including the TMB system [2] and Ryacas [3]. However, none of these options support the differentiation of functions expressed as R programs, as would an algorithmic differentiation (AD) tool for R. Attempts to develop such a tool include radx [4]. This tool is capable of computing first- and second-order forward-mode derivatives of univariate functions. But it is no longer actively developed. Natively, inside R, the numderiv package provides methods for calculating (usually) accurate numerical first and second order derivatives [5]. Accurate calculations are done by using Richardson’s extrapolation, or, when applicable, a complex step derivative is available. A simple difference method is also provided. The deriv function from the stats package computes derivatives of simple expressions, symbolically [6]. Because numerical differences cannot be reliably accurate and cannot compute adjoints, there is a need to provide derivatives within R using AD tools.

One method to obtaining derivatives is ADOL-C [7]. It is a mature and widely applied tool for algorithmic differentiation using operator overloading in the C++ language. Because of the language dependency it can natively be used only with applications that were originally written in C or C++. Previously, a Python-wrapper [8, 9] was written for the most widely used functionality in ADOL-C. It can be used to compute first and higher order derivatives in both forward and reverse mode for applications written in Python. This wrapper was written manually, however, and must be maintained and updated manually to keep in sync with the changes and new features of the C++ library. Also only the most commonly used ADOL-C API calls were available.

Given the success of manually interfacing Python with ADOL-C we have investigated an automated interfacing mechanism for ADOL-C with R and Python. We used the SWIG interface generator for this purpose. Using the interfaces that were generated, we are able to use ADOL-C from within R and Python to obtain derivatives. The rest of the document shows how SWIG was used and provides examples of ADOL-C usage.

2 SWIG interface generator

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is typically used to parse C/C++ interfaces and generate the “glue code” required for the target languages to call into the C/C++ code [10, 11, 12, 13]. It can generate interfaces for many different languages including R, Python, TCL, and Octave. Important for this work, by using SWIG, an interface for ADOL-C can be generated automatically during the build process of the ADOL-C library. Once the interface generation with SWIG has been set up correctly for the intended target languages, the generated interface will automatically contain all the new features and updates from ADOL-C.

SWIG generates interfaces based on an input file (usually somemodule.i). This input file consists of SWIG macros. A simple module may be defined, by using the input file in Figure 1. This will create a module with the name somemodule

![Image](image-url)

Figure 1: (a) SWIG input file for a simple example module; (b) skeleton ADOL-C SWIG input file containing a wrapped interface in the scripting language of choice for the C/C++ API declared in the file that is given

*Paderborn University, Germany, kshitij@math.upb.de
†Argonne National Laboratory, USA, snarayan@mcs.anl.gov
in the `%include` macro. In this case it is `<myheader.h>`. Actual C/C++ code is given between the macro delimiters `%{`and `%}`. This code is required to compile and link the generated interface with the original C/C++ library. Other macros of importance are `%ignore` and `%rename`. These will cause SWIG to ignore a certain C/C++ API name or rename it to something else for the generated interface, respectively. This feature is useful if these names contain certain characters that are unsupported by the target language or include keywords or if wrapping these in the target language is not desirable at all.

One caveat of the `%include` macro is that it will read only the named file and will not recurse into any files that are `%included` inside it, unlike the C/C++ preprocessor. This is a challenge for processing ADOL-C via SWIG, since the outer header file `<adolc/adolc.h>` contains a large number of `%include` directives for subsidiary headers, as well as system headers. Running the C++ preprocessor directly results in a file containing all the APIs from all the system headers as well as all the subsidiary headers. We do not need to wrap the system APIs for the target language, only the ADOL-C API. We therefore wrote a Python script that first excludes all the system headers from the ADOL-C headers and then runs the C++ preprocessor on it to produce a flat single header containing all ADOL-C APIs, but no system APIs. This file is then `%included` and processed with SWIG, and then the generated sources are compiled.

**R interface** The expectation that SWIG would generate a working interface from the input file automatically was not met. We encountered several difficulties when R was the chosen target language. First, the generated interface for R contained inplace modification of arrays given as arguments. Generally, R programmers prefer to use the returned values from a function as the output instead of modifying the input arguments. However, this is the standard practice in C/C++ when multiple values need to be output. SWIG version 3.0.8 did not have the necessary mechanism for modifying the input arguments. We therefore needed to modify the SWIG sources themselves and introduced `%typemap(argout)` instructions as detailed in Section 11.5 of the SWIG Documentation for considering 1D and 2D arrays as inplace modifiable arguments in R. These changes in the SWIG sources are not yet, at the time of writing this, included in any official SWIG release or source repository.

Another difficulty is imposed by the structure of the R language itself. It does not allow for operator overloading in the same sense as C++. The C++ compiler is responsible for choosing the correct operator based on context in any expression. In R, the programmer is responsible for checking the arguments to any overloaded function or operator and dispatching the correct version. As a safeguard against inadvertent overloading of common mathematical operators, the SWIG-generated interface contains named functions for such operators (e.g. “Plus” for operator `+`). To utilize operator overloading correctly, we needed to modify the generated R source code as shown in Figure 2.

**Python and NumPy interface** Using the experience in creating an interface between R and ADOL-C, we were able to create an interface for ADOL-C and Python. SWIG has been used extensively to generate Python interfaces to C++ software, for example, in the FEniCS project [14, 15, 16]. There are differences, however, in the way Python deals with intermediate results to those in C++, as well as how array data structures are handled in NumPy, the numerical mathematics module in Python. In C++, the assignment operator can be overloaded to account for the temporary intermediate _adub_ objects that are allocated on the stack with short lifetimes. In Python, the assignment operator cannot be overloaded, and all objects must be allocated on the heap. This difficulty is straightforward to
handle; we can simply ignore the operators defined in C++ and write simple one-line wrappers that will return a heap allocated adub* instead of a stack-allocated adub using a special typecast operator defined in ADOL-C. Python's own garbage collection mechanism deals with the resulting memory.

Arrays in Python are handled as numpy.array or numpy.ndarray objects. The NumPy authors have provided a SWIG input file numpy.i containing the specific typemaps for converting a C/C++ array argument given as a pointer and its size in a separate function argument. However, these work only if each such array has its own size right next to it. In ADOL-C, most drivers take several array arguments with the size; either the number of dependents or number of independents, and these sizes are known from the trace. For all such functions to be able to interpret and return NumPy arrays properly, some simple wrappers are again required, with modified C++ signatures. A few such signatures are shown in Figure 3. These wrappers are written purely in C/C++, and the maintainer does not need to write any Python code or use any Python or NumPy API for C.

Figure 3: (a) ADOL-C drivers with original signatures; (b) their NumPy array-aware wrapper signatures; (c) their usage in Python

3 Using the generated interfaces

Figure 4(a) shows the example usage of ADOL-C from within R. After the initial loading of the ADOL-C dynamic library, the code mimics familiar ADOL-C drivers written in C++. The main differences are the use of special SWIG-created interface functions for identifying the independent and dependent variables and the use of the adbouble function to initialize the independent variables. The tracing portion of the code is used to create a trace of the computation. Following that, the different functions that are invoked use the trace to compute forward or reverse derivatives. The same example in Python is shown is Figure 4(b).

4 Conclusion

We have used SWIG to automatically create an interface between ADOL-C and R as well as ADOL-C and Python. In the future, we will add support for computing the derivatives of sparse derivatives in R. Within R, we will study the usage of derivatives obtained through ADOL-C in the context of solving optimization problems and machine learning.

Acknowledgments. This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. We thank Prasanna Balaprakash, Joseph Wang, and Richard Beare for their suggestions.

References

dyn.load(paste("adolc", .Platform$dynlib.ext, sep=""))
cacheMetaData(1)
trace_on(1)
a <- adouble(2.0)
b <- adouble(1.0)
badouble_declareIndependent(a)
badouble_declareIndependent(b)
x <- a*a + b*b + 2*a*b
badouble_declareDependent(x)
trace_off()
c1 <- c(1.0,2.0)
c2 <- c(0.0)
zos_forward(1,1,2,1,c1)
c4 <- c(1.0, 1.0)
c5 <- c(0.0)
fos_forward(1,1,2,1,c1,c4)
c6 <- c(1.0, 2.0)
c7 <- c(0.0, 0.0)
gradient(1, 2, c6, c7)
c8 <- c(1.0)
c9 <- c(0.0,0.0,0)
fos_reverse(1,1,2,c8,c9)
c10 <- c(1.0, 2.0)
c11 <- matrix(0.0, ncol = 2, nrow = 1)
jacobian(1,2,c11)
c12 <- c(1.0, 2.0)
c13 <- matrix(0.0, ncol = 2, nrow = 2)
hessian(1,2,c13)

from adolc import *
import numpy as np
trace_on(1)
a = adouble()
b = adouble()
a <<= 2.0
b <<= 1.0
x = a*a + b*b + 2*a*b
X.declareDependent()
trace_off()
c1 = [1.0,2.0]
c2 = zos_forward(1,1,2,1,c1)
# c2 is np.array with .shape = (1,)
c4 = [1.0, 1.0]
(c2,c5) = fos_forward(1,1,2,1,c1,c4)
# c2 is np.array with .shape = (1,)
# c5 is np.array with .shape = (1,)
c7 = gradient(1,c1)
# c7 is np.array with .shape = (2,)
c8 = [1.0]
c9 = fos_reverse(1,1,2,c8)
# c9 is np.array with .shape = (2,)
c10 = jacobian(1,c1)
# c10 is np.array with .shape = (1,)
c11 = hessian(1,c1)
# c11 is np.array with .shape = (2,2)

Figure 4: Example usage of ADOL-C from within (a) R and (b) python


Semi-automatic transition from simulation to one-shot optimization with equality constraints

Lisa Kusch∗, Tim Albring†, Andrea Walther‡ and Nicolas R. Gauger§

1 Introduction

The transition from existing simulation models and tools to optimization can be laborious as it often demands for a redesign or re-implementation of the models to interface with a classical non-linear programming package. Especially in the context of partial differential equation solvers the idea might be abandoned because of the need for Jacobian sparsity patterns or partial derivatives. The use of AD-based discrete adjoint methods is therefore a very promising tool to allow for the development of mathematical methods and algorithmic techniques for the semi-automatic transition from simulation to optimization.

The one-shot approach is applicable to all areas of scientific computing, where large scale PDEs are treated by fixed point solvers and is advantageous for applications in which the fixed-point type solver has a rather slow linear convergence. In these applications one-shot strategies are applied to avoid the effort of recovering primal and dual feasibility in each optimization step as done in hierarchical design optimization. In the one-shot approach feasibility and optimality are obtained simultaneously involving a carefully selected design space preconditioner to achieve bounded retardation of the convergence rate. A review on one-shot approaches can be found in [1]. The coordination of primal, dual and design iteration is well established when only the state equation is considered as equality constraint. Recently, the one-shot approach was extended in [2] to include additional equality constraints. The extension is based on the one-shot approach by Hamdi and Griewank described in [3]. The optimization problem with partial differential equation constraint is solved by minimizing an exact penalty function that involves weighted primal and dual residuals added to the Lagrangian to form an augmented Lagrangian. This augmented Lagrangian was modified in [2] to extend the setting to additional equality constraints.

In the present work we apply the extended one-shot approach in the framework of the open-source multi-physics package SU2 [4]. In preliminary work SU2 was extended with an AD-based discrete adjoint [5], [6] using the AD tool CoDiPack based on the operator overloading approach. It uses static polymorphism and expression templates, resulting in an extremely fast evaluation of adjoints or forward derivatives and is specifically designed for HPC applications. Additional constraints can be found in various applications. We intend to solve multi-objective optimization problems in SU2. The idea of the equality constraint method [7] for multi-objective optimization is to transform the multi-objective optimization problem into several constrained single-objective optimization problems. The solution of these optimization problems provides points fulfilling the necessary conditions for Pareto optimality. The application presented in this work is multi-objective aerodynamic shape optimization. In [8] the scalarization approach for multi-objective optimization is applied to the shape optimization of an exhaust after-treatment system.

2 Extension of the one-shot approach

We consider design problems of the form

\[ \min_{y,u} f(y, u) \quad \text{s.t.} \quad c(y, u) = 0, \quad h(y, u) = 0, \]

where \( f \) is the objective function, \( y \in Y \) the state vector, \( u \in U_{ad} \subset U \) the design vector. The constraint \( c : Y \times U_{ad} \to Y \) describes the state equation and \( h : Y \times U_{ad} \to V \) additional equality constraints. In the following we assume that \( Y, U \) and \( V \) are finite dimensional Hilbert spaces. Furthermore, we assume that a solution of the state equation is calculated by a fixed point solver. Hence, the state equation \( c(y, u) = 0 \) can be transformed without loss of generality into a contractive fixed point equation of the form \( G(y, u) = y \).

The Lagrangian for the optimization problem (1) reads

\[ L(y, u, \lambda) = f(y, u) + \lambda^\top \lambda = N(y, u, \lambda) - y^\top \lambda, \]

*Corresponding Author, Chair for Scientific Computing, Technische Universität Kaiserslautern, lisa.kusch@scicomp.uni-kl.de
†Chair for Scientific Computing, Technische Universität Kaiserslautern, tim.albring@scicomp.uni-kl.de
‡Institute of Mathematics, Universität Paderborn, andrea.walther@uni-paderborn.de
§Chair for Scientific Computing, Technische Universität Kaiserslautern, nicolas.gauger@scicomp.uni-kl.de
where \( N(y, u, \hat{\lambda}) \equiv f(y, u) + G(y, u)^\top \lambda + h(y, u)^\top \mu \) is the shifted Lagrangian and the Lagrangian multiplier \( \hat{\lambda} \) is \( \hat{\lambda} = (\lambda, \mu)^T \). A stationary point \((y^*, u^*, \hat{\lambda}^*)\) of the problem must satisfy the first-order necessary optimality conditions:
\[
\begin{align*}
\nabla_y L(y^*, u^*, \hat{\lambda}^*) &= N_y(y^*, u^*, \hat{\lambda}^*)^\top - \lambda^* = 0, \\
\nabla_u L(y^*, u^*, \hat{\lambda}^*) &= N_u(y^*, u^*, \hat{\lambda}^*)^\top = 0, \\
\n\hat{\lambda}^* &= \mathcal{G}(y^*, u^*) = \left( \frac{G(y^*, u^*) - y^*}{h(y^*, u^*)} \right) = 0.
\end{align*}
\]

Based on the design equation (3) the one-shot strategy of Hamdi and Griewank [3] can be augmented by an iteration for the additional Lagrange multiplier \( \mu \) in the following way:
\[
\begin{align*}
y_{k+1} &= G(y_k, u_k) \quad \text{(primal iteration)} \\
u_{k+1} &= u_k - B_k^{-1} N_u(y_k, u_k, \hat{\lambda}_k)^\top \\
\hat{\lambda}_{k+1} &= N_y(y_k, u_k, \hat{\lambda}_k)^\top \\
\mu_{k+1} &= \mu_k - B_k^{-1} h(y_k, u_k) \quad \text{(augmented iteration)}.
\end{align*}
\]

where \( B_k \) and \( \hat{B}_k \) are suitable chosen invertible preconditioners.

The corresponding doubly augmented Lagrangian reads
\[
L^a(y, u, \lambda) = \frac{\alpha}{2} \left\| \hat{G}(y, u) \right\|^2 + \frac{\beta}{2} \left\| N_y(y, u, \lambda)^\top - \lambda \right\|^2 + N(y, u, \lambda) - y^\top \lambda
\]
with positive penalty parameters \( \alpha, \beta \in \mathbb{R} \). For the particular choice of
\[
\alpha \geq \max \left\{ \frac{2||N_{uu}||}{(1 - \rho)^2} \left[ \left( 1 - \rho \right)^{-p} \left\| \hat{G}_u \right\|_{1-\rho}^{1-p} - \left\| N_{uu} \right\|_{1-\rho}^{1-p} \right] \right\} \quad \text{and} \quad \beta = \frac{2}{||N_{yy}||}
\]

it can be shown, that the augmented Lagrangian is a suitable penalty function and that the update in (4) is a descent direction for \( L^a \) for large enough symmetric positive definite preconditioners \( B \) and \( \hat{B} \). As a result, the augmented one-shot iteration can be used together with an appropriate line search in a descent algorithm to find a stationary point of the augmented Lagrangian. Further details and the corresponding proofs can be found in [2]. It is also shown, that the preconditioner \( B \) is strongly related to the Hessian \( \nabla_{uu} L^a \). Further investigations on the choice of the preconditioner \( B \) of the augmented iteration will be done for the conference.

3 Application and first results

As an application example we consider the multi-objective optimization of a two-dimensional airfoil with a NACA0012 profile as initial design. The optimization task is to minimize the drag coefficient \( c_d \) and maximize the lift coefficient \( c_l \). The flow is transonic and inviscid with a Mach number of 0.8 and an angle of attack of 1.25. The airfoil is parametrized with the help of 38 Hicks-Henne functions. SU2 provides the numerical framework for the underlying PDE constraints, the steady Euler equations, using a Jameson-Schmidt-Turkel scheme.

The concept of the equality constraint method is to optimize one objective function \( f \) while imposing inequality constraints on the remaining competing objective functions. The constraints as well as the objective function to be optimized can be varied in each step of the algorithm to find different Pareto optimal solutions that are evenly distributed. The constraints as well as the objective function to be optimized can be varied in each step of the algorithm to find different Pareto optimal solutions that are evenly distributed. The resulting minimization problem for the \( j \)-th step of the algorithm applied to a general multi-objective PDE-constrained optimization problem with \( k \) objective functions is
\[
\min_{y, u} f_{s_j}(y, u) \quad \text{s.t.} \quad c(y, u) = 0, \quad f_i(y, u) - f_i^{(j)} = 0 \quad \forall i \in \{1, ..., k\} : \quad i \neq s_j,
\]
which is a problem of the form (1). The constraints for the different steps are distributed equidistantly. The outlines of the front can be found by minimizing the objective functions individually without imposing additional constraints using the one-shot approach of [3].

For all constrained optimization problems we make use of the extended one-shot approach. The preconditioner \( B \) is not computed exactly but its inverse is approximated by means of a BFGS update, as we have
\[
B \Delta u \approx \nabla_u L^a(y, u + \Delta u, \hat{\lambda}) - \nabla_u L^a(y, u, \hat{\lambda}).
\]

Thus, if one identifies \( H \) with the approximated inverse of \( B \), the secant equation is given as \( H_{k+1} r_k = \Delta u_k \) with \( r_k := \nabla_u L^a(y_k, u_k + \Delta u_k, \hat{\lambda}_k) - \nabla_u L^a(y_k, u_k, \hat{\lambda}_k) \).

It is important to apply this update only if the positive definiteness of \( H \) is maintained, which is guaranteed when using a linesearch to satisfy the second Wolfe condition. For reasons of efficiency we use a backtracking line search in the following and set \( B = I \) when the curvature condition \( r_k^T \Delta u_k > 0 \) is not fulfilled, which is a common practice.
For the BFGS update step we obtain
\[
\nabla_u L^* = \alpha G^T_u (G - y) + \beta N^T_{yu} (N^T_y - \lambda) + N^T_u + \alpha h^T \nabla
\]
with the reverse mode of AD. The second order derivative term \(\beta N^T_{yu} (N^T_y - \lambda)\) is approximated with finite differences for the direction \(y\). The multipliers are chosen as \(\alpha = 20\) and \(\beta = 2\). The preconditioner \(\tilde{B} = 20\) is constant throughout the optimization. The objective function as well as the constraint function are scaled with a factor of \(10^{-2}\).

The solutions in Figure 1 in the objective space are possible Pareto optimal solutions and were found by minimizing the drag coefficient while varying the target lift coefficient between the bound of \(c_l = 0.098\) determined by the minimum drag coefficient and \(c_l = 0.6\). The designs in the right of the figure correspond to the solutions from top \((c_l = 0.6)\) to bottom \((c_l = 0.98)\). The dashed design is the original design.

Exemplary, the optimization history for the minimization of the drag coefficient with a target lift of 0.4 is shown in Figure 2. The primal solution of the steady Euler equations would need around 300 flow iterations to reach a pressure residual of \(10^{-7}\). The drag coefficient of 0.0012 (12 drag counts) is reached after around 1400 iterations. The dashed line in the figure shows the lift coefficient that approaches the desired value of 0.40 during the optimization.

The common logarithm of the primal and the dual pressure residuals corresponding to the optimization history are shown in Figure 3. Both residuals reach a desired level of accuracy during the optimization.

Each iteration of the optimization needs the taping of two primal iterations and three tape evaluations for the adjoint. The additional constraint does not increase the costs of one iteration when compared to the classical one-shot approach, but the solution of the constrained optimization problem requires more optimization steps for the specific test case.
4 Conclusions and further work

For a semi-automatic transition from simulation to optimization we have introduced the single-step one-shot strategy extended by equality constraints in the framework of the multi-physics package SU2. The highly modular code structure and the use of the AD-based discrete adjoint therefore allows the exchange of the implemented PDE solvers. We have tested the introduced framework for multi-objective aerodynamic shape optimization, where the multi-objective optimization problem was transformed into appropriate PDE-constrained optimization problems with an additional equality constraint.

At the conference we will provide another application based on more than one equality constraint. Moreover, we will further investigate the preconditioner $\tilde{B}$ especially to enable applications with more than one equality constraint. So far, the approximation of second order derivatives with finite differences is common practice in one-shot optimization framework. We will further elaborate on the application of second order adjoints in this context.

References


On Lower Bounds for the Optimal Jacobian Accumulation Problem on Linearized DAGs

Viktor Mosenkis* and Uwe Naumann†

April 11, 2016

1 Introduction

The aim of this paper is to provide a new approach for computing lower bounds for a highly relevant graph-theoretical problem arising in Algorithmic Differentiation (AD) [1].

We consider the definition and notation of the linearized directed acyclic graph (l-DAG) for a given implementation of a multivariate vector function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ as in [1]. For example, the function $F : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ with the mapping

$$f(x_1, x_2) = (y_1, y_2)^T = \left((\sin(x_1 \cdot x_2) + x_1 \cdot x_2)^2, \exp(\sin(x_1 \cdot x_2) + x_1 \cdot x_2)\right)^T$$

can be implemented by the single assignment code (SAC)

$$v_1 = v_0 \cdot v_2; \quad v_2 = \sin(v_1); \quad v_3 = v_1 + v_2; \quad v_4 = v_2^2; \quad v_5 = \exp(v_3),$$

and yields the l-DAG shown in Figure 1(a) (Note that $x_1 = v_0$, $x_2 = v_2$, $y_1 = v_1$ and $y_2 = v_5$). The local partial derivatives

$$c_{(-1,1)} = v_0; \quad c_{(0,1)} = v_2; \quad c_{(1,2)} = \cos(v_1); \quad c_{(1,3)} = c_{(2,3)} = 1; \quad c_{(3,4)} = 2 \cdot v_5; \quad c_{(3,5)} = \exp(v_3)$$

are attached to the respective edges. A graph-based interpretation of the chain rule of differential calculus first presented in [2] yields the following method for accumulating all entries of the Jacobian matrix $A = F'(x) \in \mathbb{R}^{m \times n}$ of $F$ at some point $x \in \mathbb{R}^n$. Let $(i, \ldots, j)$ denote a path connecting a vertex $i$ with a vertex $j$ in $G$ and let $A = (a_{j,i})_{i=0,\ldots,-n-1}^{j=p+1,\ldots,p+m}$. Then

$$a_{j,i} = \sum_{(i,\ldots,j)} \prod_{(k,l) \in \{(i,\ldots,j)\}} c_{(k,l)}$$

for all sources $i \in \{0, \ldots, -n + 1\}$ and sinks $j \in \{p + 1, \ldots, p + m\}$ of edges in $G$.

The task of finding a way to compute $F'$ by using the minimal number of multiplications is generally referred to as the optimal Jacobian accumulation (OJA) problem. The general version of this problem was proven to be NP-hard in [3]. The proof exploits the algebraical dependencies of edges. The formal proof for the NP-hardness of the OJA problem under the assumption that all edge labels are mutually algebraically independent is still outstanding. This problem is referred to as the structural OJA (SOJA) problem. Often a special case of the SOJA problem is examined, where the accumulation of the Jacobian is considered as a transformation of the l-DAG $G$ into $G'$, such that $G'$ is a subgraph of the complete directed bipartite graph $K_{n,m}$. Then the labels of the edges of $G'$ correspond to the non-zero entries of $F'$. The transformation of the l-DAG is performed by elimination methods. These methods exploit the associativity of the chain rule of differential calculus and are based on the local applications of Equation (1) to parts of the graph [4]. The most commonly used elimination methods are the elimination of a vertex and elimination of an

---

*Corresponding Author, LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, mosenkis@stce.rwth-aachen.de
†naumann@stce.rwth-aachen.de
edge. The edge elimination consists of two subtypes: front and back elimination of an edge. The rigorous definitions of both elimination methods can be found in [4] or [1]. The problem of transforming an l-DAG into a bipartite graph by vertex eliminations with minimal cumulative costs is generally referred as the optimal vertex elimination (VE) problem. Similarly we refer to the problem of transforming the l-DAG into a bipartite graph by front and/or back eliminations of its edges with minimal costs as optimal edge elimination (EE) problem. Although in [4] it was shown that the solution of EE or VE problem does not solve the structural OJA problem in general, edge and vertex eliminations are often used in applications to accumulate Jacobians. Both VE and EE problems are claimed to be NP-hard, although a formal proof is still outstanding. Supporting evidence is provided in [5, 6]. Thus branch and bound based algorithms in addition to greedy heuristics are used to solve these problems as in [7]. For efficient application of such algorithms, good lower bounds are essential. Lower bounds can also help to understand the quality of the current solution. The most common application of the OJA problem is during the compiler optimization step (e.g. OpenAD/F [8]). The graphs for which the OJA problem is solved represent the basic blocks of a program. The resulting highly optimized derivative code for basic blocks is then used to speed up the derivative computation of the entire program.

The main contribution of this article is a new approach for computing lower bounds for EE and VE problems on absorption-free l-DAGs. Absorption-free l-DAGs are l-DAGs, where two vertices are connected by at most one directed path. Using the results from [9] we show how lower bounds for absorption-free l-DAGs can be used to compute lower bounds of arbitrary l-DAGs.

This paper is structured as follows. In Section 2 we provide notations and definitions that are used in this article. In Section 3 we discuss the lower bounds for the EE problem on absorption-free graphs. The results for arbitrary l-DAGs are presented in Section 4.

2 Problem description and notation

In this section we gather the definitions and notations used in this paper.

**Definition 2.1.** Consider an l-DAG $G$. The predecessor set of a vertex $v$ is denoted as $P_G(v)$, while $S_G(v)$ denotes the set of successors of $v$. We set $X(G) := \{ v \in V(G) \mid P_G(v) = \emptyset \}$, $Y(G) := \{ v \in V(G) \mid S_G(v) = \emptyset \}$ and $Z(G) = V(G) \setminus (X(G) \cup Y(G))$. The elements of $X(G)$ are referred to as independent, the elements of $Y(G)$ as dependent and the elements of $Z(G)$ as intermediate vertices. Edges that do not connect independent and dependent vertices are called eliminatable edges. The set of eliminatable edges of $G$ is denoted with $E^*(G)$. Eliminatable edges that are adjacent only to intermediate vertices, are called intermediate edges. The eliminatable edges that are adjacent to a dependent or independent vertex are called non-intermediate edges.

Next we define the notation for edge and vertex eliminations.

**Definition 2.2.** The front elimination of the edge $(i, j)$ is denoted by $G - (i, j)^+$. Similarly $G - (i, j)^-$ denotes the back elimination of the edge $(i, j)$. The elimination of the vertex $v$ from l-DAG $G$ is denoted by $G - v$.

Finally we need the definition of the optimal edge elimination sequence and the corresponding optimal costs.

**Definition 2.3.** Subsequent elimination of eliminatable vertices (edges) is called vertex (edge) elimination sequence.

We refer to a vertex elimination sequence that solves the VE problem as optimal vertex elimination sequence. $VE(G)$ denotes the costs of an optimal vertex elimination sequence.

Similarly we refer to a sequence of front and/or back eliminations of an edge that solves the EE problem as optimal edge elimination sequence, while $EE(G)$ denotes the costs an optimal edge elimination sequence.

3 Absorption-free l-DAGs

In this section we present a new approach to compute lower bounds for absorption-free l-DAGs. Its basis is the concept of critical degree originally presented in [9].

**Definition 3.1.** Consider an l-DAG $G$. An edge $(i, j) \in E^*(G)$ is called $n$-critical for vertex [edge] elimination if and only if $VE(G) - VE(G') \geq n$ \( EE(G) - EE(G') \geq n \), where $G' := (V(G), E(G) \setminus \{(i, j)\})$. For $n = 1$ we say that $(i, j)$ is critical.

We set $\Delta n^G \{(i, j)\} := EE(G) - EE(G')$ and $\Delta n^G \{(i, j)\} := VE(G) - VE(G')$, and refer to them as critical degree of $(i, j)$ in $G$ for edge or vertex elimination respectively.

The critical degree of an edge can be understood as the “contribution” of this edge to an optimal edge (vertex) elimination sequence. In general the critical degree of an edge for the EE problem can be greater or smaller than its critical degree for the VE problem. Here we will use this notation only in the context of computing an underestimator for the critical degree of an edge. It is ensured that the computed underestimator is a valid lower bound for the critical degree for both vertex and edge eliminations. Therefore throughout this article we will not distinguish between the critical degree for edge and vertex eliminations.

The lower bound for the critical degree of an intermediate edge can be computed according to the following theorem.
Theorem 3.1. Consider an absorption-free l-DAG $G$, $(a, b) \in E^*(G)$. If $(a, b)$ is an intermediate edge then
\[
\Delta_G((a, b)) \geq \min\{|X(a)|, |Y(b)|\} + |X(a)| \cdot |Y(b)|.
\]
If $(a, b)$ is a non-intermediate edge then
\[
\Delta_G((a, b)) \geq |X(a)| \cdot |Y(b)|.
\]
Where $X(a)$ is the set of independent vertices connected to $a$ and $Y(b)$ is the set of dependent vertices connected to $b$.

Proof. The proof is too long to be presented in the abstract.

For example, according to Theorem 3.1 the critical degree of the intermediate edge $(1, 3)$ in l-DAG from Figure 1(b) is at least $2 + 2 \cdot 2 = 6$ and the critical degree of the non-intermediate edge $(3, 4)$ in the same l-DAG is 2.

To compute the lower bound of the EE problem on absorption-free l-DAG we estimate the critical degree for one of the edges of the l-DAG according to Theorem 3.1. Then we remove this edge from the l-DAG and estimate the critical degree of the next edge on the resulting graph. We continue this procedure until the resulting l-DAG is completely disconnected. The following example demonstrates this approach.

Example 3.1. Consider the absorption-free l-DAG $T$ in Figure 2(b). The best lower bound for absorption-free graphs, known so far (based on sum of Markowitz degrees of the intermediate vertices [10]) gives us a lower bound of 12 for the costs of the optimal edge elimination sequence on $T$.

Let us try the new approach. According to Theorem 3.1 $\Delta_T(1, 3) \geq 2 + 2 \cdot 4 = 10$. The graph $T_1$ resulting from the removal of $(1, 3)$ is shown in Figure 2(c). Applying Theorem 3.1 to $(2, 3)$ on $T_1$ we get $\Delta_{T_1}(2, 3) \geq 10$. Hence we get the following lower bound for the costs of an optimal edge elimination sequence on $T$:
\[
\text{EE}(T) = \text{EE}(T_1) + \Delta_T(1, 3) \geq \text{EE}(T_1) + 10 = \Delta_{T_1}(2, 3) + 10 \geq 10 + 10 = 20.
\]
This lower bound is also the solution of the EE problem on $T$.

![Figure 2: Computing lower bound for the EE problem on absorption-free subgraph](image)

The quality of the lower bounds based on this approach strongly depends on the choice of the edges. Inappropriate choice of edges can worsen the lower bounds, as demonstrated in the following example.

Example 3.2. Consider the absorption-free l-DAG $T$ as in Figure 2(b). If we try to compute the lower bound for the EE problem on $T$ by applying the Theorem 3.1 to the non-intermediate edges $(-3, 1), (-2, 1), (-1, 2), (0, 2)$, we get the following lower bound for the EE problem:
\[
\text{EE}(T) \geq 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 4 = 12.
\]
As we can see the lower bound we get this way is worse than that computed in Example 3.1.

The proper choice of the edges is the subject of ongoing research. So far we can only show that one should first estimate the critical degree for the intermediate edges before starting to remove non-intermediate edges.

4 Lower bounds on arbitrary l-DAGs

In this section we show how to use the lower bounds developed in the previous section to compute lower bounds for EE and VE problems for arbitrary l-DAGs. Before we can proceed we need to define subgraphs for l-DAGs.

Definition 4.1. Consider l-DAGs $G$ and $G'$. $G'$ is a subgraph of $G$ ($G' \subseteq G$) if and only if $V(G') \subseteq V(G)$, $E^*(G') \subseteq E^*(G)$ and $Z(G') \subseteq Z(G)$.

In [9] the following result was proven.
Theorem 4.1. Consider l-DAG $G$. The following holds for every l-DAG $G'$ with $G' \subseteq G$

$$ EE(G') \leq EE(G) \quad \text{and} \quad VE(G') \leq VE(G) $$

The result of Theorem 4.1 allows us to use the lower bound of an absorption-free subgraph of $G$ as a lower bound for the l-DAG $G$ itself. For example to compute the lower bound for the EE problem of the bat graph (in Figure 2(a)), we can first build an absorption-free subgraph $T$ of the bat graph by removing the edges $(1, 4)$ and $(3, 7)$ (see Figure 2(b)). Then we can use the lower bound for this subgraph (computed in Example 3.1) as the lower bound for the EE problem on the bat graph. Hence we get $EE(G) \geq 20$. According to [4] we know that the $EE(G) = 23$. The best lower bound known so far, based on sum of minimal Markowitz degrees [7], provides a lower bound of 16.

The proper choice of the absorption-free subgraph is crucial for the quality of the lower bounds. For example, using the absorption-free subgraph resulting from the removal of the edges $(1, 3)$ and $(2, 3)$ yields a lower bound of 4 for the bat graph. Development of the algorithm for the choice of the proper absorption-free subgraph is the subject of the ongoing research.

In [4] another elimination method, face elimination (FE), was introduced. The proof that lower bounds presented here are also valid lower bounds for the FE problem on arbitrary l-DAGs is outstanding. Supporting evidence are built on the conjecture that $FE = EE$ for absorption-free l-DAGs.

5 Conclusion and outlook

In this abstract a new approach for computing lower bounds for the EE problem on absorption-free graphs is presented. In our tests this method has proven to be in most cases superior to previously known lower bounds on absorption-free graphs. This approach is then extended to arbitrary l-DAGs.

In the talk we will present the proof idea for the underestimators of the critical degree of an edge and discuss strategies for an appropriate choice of the edges during the computation of the lower bounds in more detail. We will then show under which circumstances the underestimator in Theorem 3.1 is equal to the critical degree of an edge. Based on this we will develop optimality preserving eliminations for the EE problem on absorption-free l-DAGs and formulate an algorithm for solving the EE problem on absorption-free l-DAGs with maximal path length 3 in linear time. The second part of the talk will deal with computing the lower bounds for the EE and VE problems on arbitrary l-DAGs. We will discuss strategies for choosing absorption-free subgraphs to improve the lower bounds, and talk about ways of improving the lower bounds by using the information from the parts of the l-DAG that were removed during the construction of an absorption-free subgraph.

References


AD-Suite - A Test Suite for Algorithmic Differentiation

M. Narayananurthi∗, T. Bosse†, S. Narayanan† and P. Hovland†

March 29, 2016

1 Introduction

Algorithmic differentiation (AD) is a widely accepted methodology to obtain derivatives of scientific code for use in optimization, integration methods, and sensitivity analysis. Currently about 60 AD tools are listed on autodiff.org to compute sensitivity information. Several years have been invested in the research and development of popular implementations such as ADOL-C, CppAD, OpenAD, and Tapenade. In most cases, however, the test code that comes packaged along with the tools comprises either toy or academic examples that are far removed from real-life applications. In general testing scientific software is difficult. A common reason is the lack of separation between theory and code in a scientist’s mind that limits testing to a mere verification of theory [7].

The lack of good test codes for AD in particular can be attributed to a couple of factors. One is that tools are written in and for different programming languages — C/C++, Python, Java and Fortran. The second is that AD is usually implemented by using operator overloading or source transformation and requires special formatting of the input code. These two factors contribute to the need for changes in the original simulation code or even recoding of the whole program in order to apply the developed AD software and get appropriate results. Drawing inspiration from projects that attempt to solve the problem of testing scientific software in other fields such as [1], [3], [4], [5], and [6], we propose to create AD-Suite: a test suite for AD and a classification that describes the applications included in that suite. To the best of our knowledge, there exists neither a classification of applications nor a test suite for AD. Although, projects such as SifDEC provide the ability to generate input for AD tools, they deal mostly with nonlinear optimization and hence narrow the scope for an AD classification suite; see, for example, [2], where most or all of the problems within the CUTEr test set have sparse derivatives. An additional barrier is that one needs to create a Sif-to-any-programming-language source transformation tool in order to generate test cases in different languages and the required special format (and this of course has to be tested independently). Also, most users are understandably unwilling to learn a new format and recode their application in order to apply AD.

Instead, we aim to crowd-source codes in multiple languages, spanning several application areas and having code structure that is typical in practice. In particular, we hope that AD users will actively contribute to the test suite with minimal additional effort, because further advancement of open-source AD tools is highly dependent on the variety and richness of the provided applications and examples. The examples also yield instructive templates for other interested users who want to apply AD. For AD developers, AD-Suite will be useful in a wide range of scenarios. For example, available applications will enable AD developers to better understand the needs of the users, anticipate future research areas; and design appropriate drivers. Moreover, it can be used to validate the correctness of a new implementation and compare the results. It also allows for performance comparison with other tools and different approaches on a “suitable” set of problems.

We expect AD-Suite to be publicly available on autodiff.org and contain for each problem all necessary information. As part of this effort, we envision a webform where we will collect the information along with each submission in a standard format. Through this endeavor, we hope to strengthen the AD community by bringing together AD developers, industrial partners, and users of AD.

2 AD-Suite

From a mathematical point of view, most scientific codes can be thought of as a sufficiently smooth function $F : \mathbb{R}^n \times \mathbb{R}^q \rightarrow \mathbb{R}^m$ with output $y = F(x, p) \in \mathbb{R}^m$ that depends on some parameter $p \in \mathbb{R}^q$ at a given base-point $x \in \mathbb{R}^n$. Depending on the application, these functions usually have special characteristics and/or a certain structure, which is reflected in their implementation. For example, in regression analysis the function $F$ represents a code evaluating the objective function of the optimization problem

$$\min_{x \in \mathbb{R}^n} F(x, p) = \sum_{i=1}^{r} \|f(x, d_i) - m_i\|.$$  

(1)

∗Corresponding Author, Virginia Tech, US, maheshnm@vt.edu
†Argonne National Laboratory, IL, US
The objective function minimizes the sum of \( r \in \mathbb{N} \) residuals, in a given norm, between the observed measurements \( m_i \in \mathbb{R}^k \), and a model function \( f : \mathbb{R}^n \times \mathbb{R}^s \rightarrow \mathbb{R}^k \) for some corresponding input \( d_i \in \mathbb{R}^l \). In other words, \( x \in \mathbb{R}^n \) has to be chosen such that \( f \) fits the data \( p = (d_i, m_i)_{i=1}^r \) in an optimal way. Obviously, parts of this function/code can be evaluated in parallel, and the same holds true for its derivatives. A different class of examples, which typically arise in time integration methods, comprises functions that represent a repeated composition

\[
F(x, p) = F_1(F_{r-1}(\ldots F_2(F_1(x, p), p), \ldots), p)
\]

of functions \( F_i : \mathbb{R}^n \times \mathbb{R}^q \rightarrow \mathbb{R}^{m_i} \) with \( m_{i-1} = n_i, \ i = 2, \ldots, r \), such that each successive stage of the computation is dependent on the previous stage.

These structures need to be taken into account for an efficient evaluation of the sensitivity information for \( F \) by some AD tool. Other features include the smoothness of \( F \), the sparsity of the derivatives, and the number of (in-)dependent variables (see Table 1). In fact, most AD tools have specialized drivers that allow an efficient treatment for the different classes of problems. For example, ADOL-C provides a checkpointing algorithm to efficiently compute adjoints of functions \( F \) that are given by a repeated composition [S]. Naturally, a successful implementation of such drivers requires suitable test examples in order to validate the computational results and get a measure of the progress from the latest developments.

## 2.1 AD Classification

In addition to collecting code samples from the AD community, we would like to define an extensible standard that can be used to categorize each application. This standard will help the developers find appropriate test examples and allow AD users to choose the correct AD software or driver. Therefore, we follow the CUTEr example and assign each problem an intuitive label that consists of \( 5 + 3 + (1) \) alpha-numeric entries that are separated by hyphens using the ordered categories and code features given in Table 1. In detail, we propose to use the following (ordered) syntax.

### Example

For example, the label C2-L-N-S-S-100-10-4 can be used to denote a twice differentiable problem with a loop structure that is nonlinear and has sparse derivatives similar to that given in [2]. Its evaluation is serial, with 100 independent and 10 dependent variables and 4 parameters. The (optional) special feature can be used to provide user-specified information about a problem in more detail depending on its underlying structure or implementation using the given classification codes. For example, if the problem has a loop consisting of 99 iterations, one could label the previous example by C2-L-N-S-S-100-10-4-L99. In particular, every code feature with the exception of the dimensions, which is not yet listed in Table 1, should be a capital letter to allow for multiple features. It can be followed by one or more lower-case letters and numerical values, as was done in the given example to indicate its smoothness (C2), or the special feature to describe the number of iterations for the loop (L99); that is, each entry can consist of one or more entries of the form [A-Z][a-zA-Z0-9] using Sed syntax. This allows the convention to be more extensible and stay consistent. For example, the first example [1] gives rise to Lipschitz continuous functions (Lc0) with a nested structure (N) that require the computation of derivatives and the solution of linear problems (S) within the code itself. Hence, an appropriate label would be Lc0-NS-N-D-P-5-1-99999 if the nonlinear function with dense derivatives has 5 independents, 1 dependent variable, and a large number of parameters.

Separately, AD developers may also use the label to provide information about the tool capability. For example, C2-L-N++-S-500-500-++ may represent an AD tool with checkpointing capability (L) that can efficiently handle at least twice differentiable (C2) nonlinear functions (N) with a serial loop structure (S), which could have sparse or dense derivatives (**), up to 500 (in)dependent variables, and an arbitrary amount of parameters (**).

### Table 1: Classification codes for different AD applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Code features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoothness of ( F )</td>
<td>Lipschitz continuous (C0), ( C^1 ) (C1), ( C^2 ) (C2), ..., ( C^\infty ) (C( \infty ))</td>
</tr>
<tr>
<td>Source Code Features</td>
<td>Loop (L), (non-)linear Solver (S), Cross-derivative (C), Nested Derivative (N),...</td>
</tr>
<tr>
<td>Nonlinearity of ( F )</td>
<td>Linear (L), Quadratic (Q), Rational (R), Nonlinear (N)</td>
</tr>
<tr>
<td>Sparsity</td>
<td>Sparse (S), Dense (D), Block-structure (B)</td>
</tr>
<tr>
<td>Parallel</td>
<td>Serial (S), Parallel (P),</td>
</tr>
<tr>
<td>Dimensions</td>
<td>m-n-p: Independents (n), Dependents (m), Parameters (p)</td>
</tr>
<tr>
<td>(Special Feature)</td>
<td>(user-specified)</td>
</tr>
</tbody>
</table>

## 2.2 Standard Test Problems for AD

We expect the AD-Suite to contain applications that have been successfully differentiated, along with supporting files (described below) and the correct classification code. Submissions to the suite will use the directory structure illustrated in Figure 1 and will contain the following information.
The differentiated code must compile and run without errors or abortion, for example, after calling 
respectively. The standard signature of the methods follows the input/output convention

driver files for the differentiated code are in the root directory and contain only one method, for example,
the version (public release) or revision number (code repository), have to be specified in the
must be stored in the separate folder
corresponding
calling
In particular, the code must run the
methods follows the input/output convention
for example,
should be described in the
packages a path variable, which needs to be adapted such that the code (compiles and) executes. The necessary changes
elsewhere. The driver files are in the root directory with a corresponding
data for the variable
do expect, however, that cited literature in some standard format (typically
the developers will provide detailed documentation of the code solely for the purpose of inclusion in the test suite. We
comments within the code of the form “this method implements formula XX in YY”. Naturally, we do not expect that
not meant to be used outside the research group that develops them. Usually, the documentation consists of only minor
structure; for example, one must store the intermediate values of each iteration
resulting numerical values for derivatives such as the Jacobian-vector product
The files for the Jacobian/first derivative, Hessian/second derivative, and so on are simply called deriv_1.csv, deriv_2.csv
,... The values for directions $\bar{\eta}$ and adjoint $\bar{y}_j$ are stored in the files dir_1.csv and adj_j.csv, respectively. The
resulting numerical values for derivatives such as the Jacobian-vector product $F'(x)\bar{x}_1$ are given in correspondingly named files (e.g., deriv_1_dir1.csv). Also, this folder must contain additional files for applications with a special structure; for example, one must store the intermediate values of each iteration $i$ in the files iter_i.csv if the example
involves a loop and allows for checkpointing.

Documentation and literature for the test Often, the documentation of scientific codes is limited because they are not meant to be used outside the research group that develops them. Usually, the documentation consists of only minor comments within the code of the form “this method implements formula XX in YY”. Naturally, we do expect that the developers will provide detailed documentation of the code solely for the purpose of inclusion in the test suite. We do expect, however, that cited literature in some standard format (typically pdf) will be provided in the folder doc to help other researchers and AD developers grasp the most important steps of the code.

Source code of the example The code must be completely contained within the directory src_code. Driver files, the data for the variable $x$, the parameter $p$, the directions $\bar{x}_i$, and the adjoints $\bar{y}_j$ with the corresponding output exist elsewhere. The driver files are in the root directory with a corresponding Makefile that includes for all third-party packages a path variable, which needs to be adapted such that the code (compiles and) executes. The necessary changes should be described in the README file. Each of the driver files must contain only one method that either reads one of the dimensions for the scenario $t$, the initial values for $x$ or $p$, or evaluates $y = F(x)$ with corresponding filenames, for example, get_dim_n.X, get_x.X, and eval_F.X for the language-specific extension X. The standard signature of the methods follows the input/output convention

```
get_dim_n( n, $l$ ), get_x( \frac{n}{\bar{n}}, x, \frac{l}{\bar{l}} ), save_x( \frac{n}{\bar{n}}, \frac{\bar{x}}{\bar{\bar{x}}}, \frac{l}{\bar{l}}, \bar{x}, y, \frac{p}{\bar{p}} ), flag ).
```

In particular, the code must run the 4th test scenario without errors or abortion after modifying the Makefile and calling make && make test t. Furthermore, the call make test t should also create the output file y.csv in the corresponding data subdirectory.

Differentiated code or corresponding driver files Similar to the original source, we expect that the differentiated code must be stored in the separate folder src_deriv. The AD tools used to generate the derivative code(s), along with the version (public release) or revision number (code repository), have to be specified in the README file. Again, the driver files for the differentiated code are in the root directory and contain only one method, for example, deriv_1.X, deriv_1_dir.X, and adj_deriv_1.X for the computation of the Jacobian, Jacobian-vector, and vector-Jacobian products, respectively. The standard signature of the methods follows the input/output convention

```
deriv_1( \frac{n}{\bar{n}}, \frac{\bar{m}}{\bar{\bar{m}}}, \frac{\bar{q}}{\bar{\bar{q}}}, \frac{\bar{x}}{\bar{\bar{x}}}, y, F'(x), \bar{p}, flag ), deriv_1_dir( \frac{n}{\bar{n}}, \frac{\bar{m}}{\bar{\bar{m}}}, \frac{\bar{q}}{\bar{\bar{q}}}, \frac{\bar{x}}{\bar{\bar{x}}}, y, F'(x)\bar{x}_1, \bar{p}, flag ),
```

The differentiated code must compile and run without errors or abortion, for example, after calling make && make deriv_1 t to evaluate the Jacobian of test scenario t. Additionally, the output file deriv_1.csv must be generated in the data subdirectory.
**Testing information** Besides the data for the validation of the code and its derivatives, other information may be of interest to a user, such as the sparsity ratio for each computed derivative. Of special importance are performance metrics for comparing different AD tools applied to a considered test example. Of course, these performance metrics should be measured by normalized values because simple run-time measurements are overly influenced by other factors such as the programming language and the system architecture. More independent estimates for the performance of an AD tool seem to be the averaged run-time ratios for each task w.r.t. the time required for evaluating the original function itself:

\[
\frac{\text{Time trace } [\mathcal{F}(x)]}{\text{Time eval } [\mathcal{F}(x)]}, \quad \frac{\text{Time eval } [\mathcal{F}(x) \vec{x}_1]}{\text{Time eval } [\mathcal{F}(x)]}, \quad \frac{\text{Time eval } [\vec{y}_1^T \mathcal{F}'(x) \vec{x}_1]}{\text{Time eval } [\mathcal{F}(x)]}, \ldots
\]

Similarly, one can define a normalized measure for the ratio of peak/average memory consumption with respect to the peak/average memory consumption for only the function evaluation. For completeness, the software and hardware platform used for obtaining the metrics should be reported. All this information will be collected in an editable csv-file on [autodiff.org](http://autodiff.org) with additional information that could, for example, include the number of used parallel processes or the checkpointing pattern, density, or ratio if applicable. The file can be used for state-of-the-art performance plots without the necessity of redoing all the work and testing.

### 3 Conclusions and further work

We have started collecting a small number of “realistic” problems, which we will provide to the AD community for testing. The examples are given in the proposed format and are used to describe the “standard” format in more detail. Moreover, we will try to formulate a suitable extension of the standard to provide additional information such as the underlying discrete graph structure for the test-set problems that could be used by other AD developers to improve their algorithms. We envision having AD-Suite become an interactive part of the AD website [autodiff.org](http://autodiff.org) along with a small set of minitools, which could be used, for example, to create performance plots.

**Acknowledgments**

This material was based upon work supported the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

**References**


1 Introduction

Development of a capable algorithmic differentiation (AD) tool requires large developer effort to provide the various flavors of derivatives, to experiment with the many AD model variants, and to apply them to the candidate application languages. Considering the relatively small size of the academic teams that develop AD tools, collaboration between them is a natural idea. This collaboration can exist at the level of research ideas as well as tool development.

This work describes the interoperability of two source-to-source transformation AD tools OpenAD [1, 2] and Tapenade [3, 4]. The interoperable pipeline uses the parsing and source analysis capabilities of Tapenade with the transformation algorithms of OpenAD.

The aim of such interoperability is to ensure the robustness and stability of the AD tools. The redundancy between some components of either tool offers more flexibility to the end user. A weakness in one component may be compensated by choosing another route in the components graph. A long-term objective is “à la carte” AD, where one may combine powerful capabilities from either tool, for instance, the preaccumulation capacities of OpenAD with the accurate data-flow model of Tapenade for activity, adjoint liveness, and TBR analysis. Additionally, not relying on any one component, such as the front-end compiler developed externally, allows the AD tool to persist beyond the lifetime of that front end compiler. Even further, we can analyze the strengths and weaknesses of each tool’s AD model. These models are similar (source transformation, with a store-on-kill adjoint model), yet some choices differ, for instance, association-by-name vs. association-by-address [5].

2 Architecture of the interoperable tool

Interoperation between the OpenAD and Tapenade is possible because they share the same global architecture, namely, a front-end that parses and builds an internal representation, followed by a static data-flow analysis stage, then actual building of the differentiated program still in internal form, and finally a back-end that outputs this differentiated internal form into new source files. Figure 1 details this architecture for OpenAD and for Tapenade.

OpenAD pipeline OpenAD’s pipeline starts with a custom Python preprocessor. The preprocessed code is parsed by using an Open64 translator called whirl2xaif that generates whirl intermediate representation, invokes OpenAnalysis for program analysis, and transforms the whirl representation and analysis results into the XML Abstract Interface Form (XAIF) [6]. The XAIF is given to XAIFBooster [7] as input, which produces an augmented (differentiated) XAIF. A stored version of the original whirl representation and the differentiated XAIF are used by xaif2whirl

*Corresponding Author, Argonne National Laboratory, snarayan@mcs.anl.gov
†INRIA Sophia-Antipolis, laurent.hascoet@inria.fr
to produce a differentiated whirl representation, which is then used to generate Fortran code. The Fortran code is postprocessed using a Python postprocessor. The pipeline for C programs is similar, using a different parser/unparser.

**Tapenade pipeline** The Tapenade pipeline starts with a parser for the application language. Parsers produce an abstract syntax tree with predefined tree operators encompassing the syntactic structures of both Fortran and C, extensible to other imperative languages, hence the name “IL.” The input IL tree is immediately transformed into Tapenade’s internal representation, a call graph (CG of FG) whose nodes represent procedures, and each procedure node is itself a flow graph whose nodes are basic blocks of sequential code. The data-flow analysis stage operates on the CG of FG, annotating it with the analysis results; then actual differentiation produces a new CG of FG, which is then translated back into an IL tree. The last stage unparse the differentiated IL tree into the application language.

This architecture, adopted by both OpenAD and Tapenade, is standard and is used by many other AD tools. Having the same architecture, however, does not ensure interoperability. At a deeper level, the internal representation of programs must also use the same concepts so that it can be easily transferred between tools. OpenAD and Tapenade explicitly use a CG of FG structure. Symbol tables are also used in the same manner.

Another requirement is that the tools not be monolithic: successive components of their workflow must be identified and clearly separated. For OpenAD, each component represented by the blue rectangular boxes in Fig. 1 is programmed independently and implemented in an arbitrary language. Each component respects the imposed format of its input/output. At two places, this format is XAIF. The XAIF holds the call graph, flow graph, and symbol tables; it also contains the results of alias and analysis as well as DefUse chains and DefOverwrite chains. Some components, however, require information from tools earlier in the workflow. These (dotted lines in Fig. 1) are propagated through the chain in an ad hoc manner, as pointers kept in XAIF. This approach restricts modularity, as some components downstream impose a specific component upstream. Tapenade was not designed with openness as a primary objective. The workflow components are clearly separated Java packages, but they operate on an in-memory internal representation of the code. In most places, this in-memory object is the CG of FG discussed above, except at both ends of the chain, where it is an abstract syntax tree in the IL formalism.

Based on the architecture of the two tools, one can identify the connections where program information between them can be exchanged (red arrows in Fig. 1). The connections represented by solid arrows are the ones implemented and tested in this work. Starting from either tool’s front-end, one may transfer to the other tool to take advantage of its additional analyses, then run one tool’s differentiation component, and finish with either tool’s back-end. At present, because of the dependencies of OpenAD’s back-ends on their respective front-ends, selecting an OpenAD back-end (rightmost bridge arrow in Fig. 1) implies that the corresponding OpenAD front-end (leftmost bridge arrow) must have been used. The format used for transfers between separate processes must be XAIF, and it is Tapenade’s task to transform its in-memory internal representation to XAIF and back. Figure 1 is simplified and does not show possible connections upstream of the OpenAnalysis tool(s). In the future we may be able to split up components of the actual differentiation stages, allowing for fine blending of the differentiation models.

![Figure 2: Overview of XAIF callgraph schema](http://www.mcs.anl.gov/XAIF xaif.xsd)
3 Translating to and from XAIF

Figure 2 shows an overview of the CallGraph element of the XAIF schema that represents a program. Within it, analysis results for DefOverwrite chains, DefUse chains, Alias analysis results, and Scope hierarchy exist along with the control flow graphs for each procedure in the program. Activity analysis results are stored within the symbol tables inside the scope hierarchy and within the statements contained inside the control flow graphs.

Translating between XAIF and Tapenade’s internal representation is straightforward. A natural match exists between almost all structures, which is not surprising. Obviously, graph structures require some easy serialization in order to translate to XAIF. One difference between the tools is that XAIF systematically uses a general graph structure even for abstract syntax trees, which makes it a little harder to find the root of these trees. However, this allows XAIF to naturally represent common subexpressions. Translation to Tapenade loses this and results in duplicated common subexpressions.

Tree operators for the abstract syntax of general imperative languages are almost the same in XAIF and Tapenade. However, Tapenade tries to handle the union of operators used in Fortran and in C, OpenAD tries to reduce this set of operators, relying on a preliminary canonicalization stage. The advantage is a smaller number of cases to manage in the analysis and differentiation stages. The drawback is that the final differentiated code is also canonicalized and thus harder to read because it discards some syntactic choices of the source. Consequently, the Tap2Xaif translator must apply canonicalization, too; and code coming back to Tapenade may have a slightly poorer, though equivalent, form.

Tapenade’s internal representation takes in Fortran (90) modules as first-class concepts, whereas they are simulated in XAIF through special markers. This distinction requires some technical manipulations in the translators on the bridges, unless the XAIF side evolves to incorporate XML elements for modules as it has for procedures.

Figure 3 presents the steps involved in using our pipeline. First, Tap2Xaif calls Tapenade to parse and analyze an input source code and to output the result in XAIF. Then XAIFBooster is used to generate differentiated XAIF.

4 The OpenAD template mechanism

Figure 4 presents a simplified view of the back-end. While Tapenade fully creates the structure of the differentiated code during its differentiation stage, the XAIFBooster produces a collection of differentiated pieces (e.g., tangent, forward adjoint, backward adjoint . . . ) in the differentiated XAIF.

The differentiated XAIF is translated by xaif2whirl into a “glued” source code with special markers to separate out the different portions. This code is then post-processed according to templates to produce the final differentiated code. This approach allows OpenAD to separate AD at the basic block level (performed by XAIF-Booster) from AD at the higher flow graph level (done by the postprocessor). The basic-block level deals with differentiation of individual assignments, including, for instance, preaccumulation. Additionally, XAIFBooster determines what data to be checkpointed in adjoint mode, while the postprocessor uses the template to generate checkpointing code. The postprocessor also includes refined strategies for special constructs such as fixed-point loops. In contrast, the Tapenade differentiator handles all these issues at the same time through two different Java classes, called FlowGraphDifferentiator and BlockDifferentiator, with methods from the former calling methods from the latter. This is a significant difference between the OpenAD and Tapenade.

Interoperation requires (see Fig. 4) that Tapenade, after receiving an OpenAD-differentiated XAIF file, generate the same “glued” output with the appropriate markers in it, so that it can be sent back to XAIF’s postprocessor at the end of the pipeline. We have tested this mechanism with success on our first examples. An alternative
could be to let Tapenade apply its own flow graph reconstruction strategy to the collection of differentiated pieces returned in XAIF. Technically, this means using Tapenade’s `FlowGraphDifferentiator` while shunting the calls to its `BlockDifferentiator`. We have not implemented this alternative.

5 Conclusion and future work

We have described the implementation of a tool architecture that uses the front end, analysis, and backend of Tapenade with the transformation algorithms of OpenAD’s XAIFBooster component. The new tool architecture has been tested on several small test codes from OpenAD’s regression test suite. We plan to apply the pipeline to larger codes.

OpenAD and Tapenade share runtime libraries as well. Adjoinable MPI has been created to handle MPI calls in reverse mode AD. ADMM is a library to handle dynamic allocation of memory in the reverse mode of AD. It is expected the new architecture will continue to be able to use these libraries in a transparent manner.

Both OpenAD and Tapenade support nonstandard differentiation techniques. The Christianson method for fixed point iterations [8] is supported by both OpenAD and Tapenade. OpenAD employs a special template and its postprocessor for this purpose. Tapenade’s `FlowGraphDifferentiator` handles it internally. We will support the use of either approach in the interoperable pipeline.

Acknowledgments. This material was based on work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357; and the INRIA-Illinois-ANL-BSC-JSC-RIKEN/AICS Joint Laboratory on Extreme Scale Computing, and the Partner University Fund.

References


Adjoint Code Design Patterns

Uwe Naumann*

The Problem  According to common Algorithmic Differentiation (AD) wisdom [1, 2] an algorithmic adjoint of a given implementation of a multivariate vector function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m : y \equiv z^q = F(x)$ as a computer program (the primal code) evaluates

$$
\begin{align*}
\bar{X} & \in \mathbb{R}^{n \times 1} = \nabla F^T, \\
\bar{Y} & \in \mathbb{R}^{m \times 1} = \nabla F^{1T} \cdot (\ldots \nabla F^{kT} (z^{k-1}) \cdot (\nabla F^{k+1T} \cdot \ldots ) (\nabla F^{qT} \cdot \bar{Y}) \ldots ) \ldots
\end{align*}
$$

for $x \equiv z^0$ and elementals $z^i = F^i(z^{i-1})$ with Jacobians $\nabla F^i(z^{i-1})$, $i = 1, \ldots, q$. The art of differentiating computer programs amounts to provision of correct, efficient, scalable, and extensible (in the following abbreviated as “adequate”) Adjoint Code Modules $\bar{Z}^{i-1} = \nabla F^i \cdot \bar{Z}^i$ for all elementals. It is widely accepted that sustainable adjoint simulation software engineering should expose fundamental mathematical and structural / data dependence properties of individual modules. Relevant scenarios include checkpointing of evolutions [3] and of general call trees, path-wise adjoints of ensembles [4], reverse accumulation [5], integration of symbolic [6] or hand-written adjoints and handling of non-differentiable elementals [7] as well as treatment of black-boxes, coupling of overloading and source transformation AD tools, integration of accelerators such as GPUs, preaccumulation while exploiting sparsity / scarcity, and libraries of adjoint versions of numerical methods or user-defined / domain-specific adjoint elementals.

For motivation we consider the solution of a stochastic differential equation (SDE) for pricing a European call option as described in [8]. The numerical method amounts to a Euler-Maruyama scheme [9] performing $\mu$ explicit Euler integration steps along $\nu$ independent Monte Carlo sample paths. A basic implementation of this method evaluates a function of the following structure:

$$
y = F(x) = G \left( \begin{array}{c}
P_{\mu-1}^0 (P_{\mu-2}^0 (\ldots P_0^0 (S^0(x)) \ldots )) \\
\vdots \\
P_{\mu-1}^\nu (P_{\mu-1}^\nu (\ldots P_0^\nu (S^\nu-1(x)) \ldots )) \\
\end{array} \right),
$$

where the arguments $x$ (strike price, time to maturity, interest rate(s), price of underlying and market observed volatilities) are scattered over the individual paths by the $S^i$, $i = 0, \ldots, \nu - 1$, each path performs integration steps $P_j^i$, $j = 0, \ldots, \mu - 1$, and the results are gathered by $G$ to yield $y$ (option price). A similar structure is discussed in [10] and can be found in numerous other practically relevant applications.

An adequate adjoint $\bar{X} = \nabla F^T(x) \cdot \bar{Y}$ is likely to use binomial checkpointing [3] at the level of individual paths to ensure optimal tradeoff between memory requirement and operations count. Concurrency among the paths should be exploited in order to limit the memory requirement to that of a single path (or a set of paths) [4]. A lower memory footprint may enable evaluation of the adjoint in faster memory (e.g. cache). Parallelization of path-wise adjoints is likely to scale almost ideally.

These thoughts should be reflected in the design of the adjoint code. The sketched structure is common to various methods for solving SDE. Implementations of adequate adjoints should

---

*LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, D-52056 Aachen, Germany. Email: naumann@stce.rwth-aachen.de
provide the building blocks \((S^i, P_k^j, G)\) without having to deal with specifics of the data flow reversal mechanism within \(F\). More generally, modularization of the primal and exposure of both data dependencies among the modules and of their mathematical properties promises correct, efficient, scalable, and extensible adjoint as well as primal code.

**The Proposed Solution** We propose *Adjoint Code Design Patterns* as a fundamental approach to software design in the context of AD. Tangent code is naturally covered by similar ideas, as it turns out to be substantially less challenging from a software development perspective.

An excerpt from the ACDesignPatterns library is visualized in Fig. 1 showing a corresponding UML Class Diagram generated with Visual Paradigm.\(^1\) The relevant Adjoint Code Module hierarchy is wrapped into the namespace *ACDesignPatterns*. At its core stands the generic abstract class *ACModule* containing references to the module’s inputs and outputs and to the corresponding adjoints. Support for checkpointing all input arguments is provided in order to enable both split and joint reversals \([1]\). The logic of binomial checkpointing of evolutions over a given *step* function through recursive bisection and using recompute-all reversal of the *tail*\(^2\) is encapsulated in correspondingly named interface classes. Path-wise adjoints of ensembles with mutually independent *path* functions are wrapped in a similar fashion. Default implementations for the evaluation of the primal, the split (augmented) primal and the split adjoint are provided at all levels of specialization. The user is forced to implement this functionality if deriving directly from the abstract base class *ACModule*. When specializing other, non-abstract interface classes the user has the option to overwrite the given implementations. Note the recursive structure of the *ACDesignPatterns* module hierarchy. For example, a single evolution step could involve the iterative solution of a parameterized system of nonlinear equations. Again, checkpointing may be required. Alternatively, a symbolic adjoint of the implicitly defined solution might be preferred \([6]\). An appropriately specialized Adjoint Code Design Pattern can be provided.

As a proof of concept we consider the design of an algorithmic adjoint for Eqn. (1). In our case study, merely the integration *step* function needs to be implemented. In particular, custom versions of a primal, a split primal and a split adjoint evolution step are required. Passive user data can be collected in *MyACUserData* and shared among custom Adjoint Code Modules as required, e.g. random numbers are used within each evolution step in an SDE scenario. Specializations of the remaining interface classes ensure instantiation of *path*, *step*, and *tail* with the appropriate custom type. The CLIENT (e.g. the main routine) allocates the top-level Adjoint Code Module and the user data followed by evaluations of the split primal and adjoint. A corresponding UML Object Diagram is shown in Fig. 2. Refer to [www.stce.rwth-aachen.de/files/naumann/AD16/html/](http://www.stce.rwth-aachen.de/files/naumann/AD16/html/) for the documented source code with a toy problem solved for illustration of the user perspective. Type genericity of Adjoint Code Modules enables instantiation with custom data types and thus application of operator overloading techniques. Seamless integration of, for example, (higher-order) AD by overloading, interval or arbitrary precision arithmetic or convex relaxations are supported provided that actual user implementations of the interface feature the same level of type genericity as the ACDesignPatterns library itself.

Adjoint Code Modules can serve as abstract entry points for adjoint callback interfaces provided by certain overloading AD tools to deal with external functions. The same tools can be used to provide implementations for the evaluation of adjoints for individual modules representing the building blocks of the given problem. The proposed solution is meant to be fully recursive.

The development of the ACDesignPatterns library is work in progress. A first version of the Community Edition (GPL) is expected to be released prior to the AD2016 conference. The talk discusses Adjoint Code Design Patterns in general and the given case study in particular including details on our recursive bisection approach to the implementation of binomial checkpointing in the context of call tree reversal methods. It aims to promote a sustainable approach to adjoint code development through proven software engineering methodology.

---

\(^1\) [www.visual-paradigm.com](http://www.visual-paradigm.com)

\(^2\) steps following the last available checkpoint
Figure 2: ACDesignPatterns: User Perspective

References


Comparing High-Order Multivariate AD Methods

Richard D. Neidinger* and Ben Altman†

March 30, 2016

1 Introduction

To compute all multivariate derivatives up to an arbitrarily high order, interpolation methods will be shown to be significantly less reliable for accuracy than a direct forward method. Since the accurate method had comparable efficiency, we explain how the implementation handles the indirect referencing into the large non-rectangular data structures used in the direct forward method.

Forward AD methods using Taylor series are known to be an effective tool for computing arbitrarily high order derivatives. Multivariate Taylor series coefficients (or unique partial derivative values) up to some arbitrary but fixed high order form a large non-rectangular data structure, that will be called a corner. There are direct methods that propagate these corners through a program, as in [1]. Alternatively, interpolation methods propagate univariate directional series and reconstruct the corner values at the end, which has been shown to be theoretically more efficient. The original method, that we call GUW-interpolation, from [2] (and presented in [3]) uses a direction corresponding to each multi-index of highest order and combines all of these univariate series into the desired corner of values, by multiplying by a large pre-computed matrix of constants. Another method, we call nested interpolation, from [4] uses different directions which can be combined (theoretically) even more efficiently. Nested refers to the fact that directions for order $d$ contain directions for order $d-1$, and only that subset is used in the order $d-1$ computation (unlike GUW that uses all directional values). These two interpolation methods and a direct method were implemented in MATLAB on the same machine for the purpose of comparing speed and accuracy.

Our goal is to understand and evaluate the algorithms necessary to compute a full corner of values, so sparsity will not be considered in this context. In application, sparsity and contextual structure are very important considerations. We will assume that a seed matrix ([3], p. 311) has already restricted to a smaller subspace of dimension $n$. We also assume that order $d$ is fairly large, say 5 to 20, so that interpolation has a theoretical advantage and forward can compete with reverse strategies [2].

2 Comparing accuracy and efficiency

Our implementations of the interpolation methods strove to simply code the methods as described in the corresponding papers. Coding was done on an older Windows laptop in MATLAB, so consistency, but not overall speed, was the goal. Direct multivariate methods have been implemented by many over the years. The idea is a natural generalization of univariate series manipulation, although managing the corner data structure is a sticking point, so our method for this is described in section 3.

We will consider several examples for testing but focus on one typical example for now. The tennis serve function

$$t(x, y, z) = \frac{y^2 \cos^2 x}{32} \left( \tan x + \sqrt{\tan^2 x + \frac{64z}{y^2 \cos^2 x}} \right)$$

models the horizontal range of a tennis serve as a function of angle $x$, initial velocity $y$, and initial height $z$, in feet and seconds. We seek to compute derivatives up to order $d = 20$ for this function of $n = 3$ variables. To easily compute relative error, we embed this function (in a meaningless way) into a normalized function that has derivative 1 for every possible higher-order derivative at (1,1,1). Series coefficients for $t$ are computed but then divided by themselves in

$$g(x, y, z) = \exp \left( \frac{t(x, y, z)}{t(x, y, z)} (x - 1) + (y - 1) + (z - 1) \right).$$

The resulting performances of the three implementations are shown in Table 1.

While the timings reveal lessons in where the naive implementation should be improved, the more serious concern is the relative error. This is the maximum error given by the infinity norm, although many terms include significant error, which does grow with the order of the derivative. The direct method effectively achieves machine precision accuracy on this example. The much larger GUW-interpolation error around $2e-04$ was also experienced by ADOL-C.

*Corresponding Author, Davidson College, rineidinger@davidson.edu
†Davidson College undergraduate research, bealtman@gmail.com
The nested interpolation method completely failed to give a useful value for this 20th derivative. However, there are easier examples where all methods perform with excellent accuracy, confirming the implementations. There are also harder examples where all methods struggle to find accurate answers, and symbolic computation is even worse. We will consider how condition numbers of matrices associated with the methods help explain the error for typical examples. We also reflect on efficiency lessons, but now realize that the interpolation methods are unreliable in the current form. Possibly another different choice of directions might salvage the idea.

Concerning efficiency, the GUW-interpolation does show the run time savings. This run time includes both the propagation of univariate series (theoretically \( q = \frac{23}{2} \) times the cost of direct corner propagation) and cost of interpolation at the end (which matrix multiplies by nonzero elements in a large global array). Clearly, one-time computation of the array was not done efficiently but is a naive implementation of the formula for coefficients as on p. 315 of [3]. It does point out the need to find an efficient way to pre-compute this large array where each entry is a large sum of weighted multinomial coefficients. Note that once this array is computed for \( n = 3 \) and \( d = 20 \), then it may be reused on any example function or program that seeks 20 derivatives in three chosen subspace variables. In contrast, the nested interpolation went to the other extreme so that run time does not show the expected savings. In theory, nested interpolation is doing a smaller number of floating point operations, but the lesson here is that the algorithm unwisely chose to forgo pre-computation of arrays to perform the interpolation. Instead, the method computes a literal interpolating polynomial in Newton form on the fly and collects terms computationally to get the final values. It appears that this is too computationally expensive, though pre-computing inverse matrices for multiplication should alleviate this problem. Still the error problem persists, so that such improvements are probably not worth pursuing.

### Table 1: Timing and Accuracy for tennis serve function to order 20

<table>
<thead>
<tr>
<th>Method</th>
<th>Run time (sec)</th>
<th>One-timeGlobals (sec)</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct forward</td>
<td>12.4</td>
<td>4.8</td>
<td>5.6e-16</td>
</tr>
<tr>
<td>GUW-interpolation</td>
<td>5.6</td>
<td>13617.5</td>
<td>1.7e-04</td>
</tr>
<tr>
<td>nested interpolation</td>
<td>18.2</td>
<td>0.0</td>
<td>9.8e+00</td>
</tr>
</tbody>
</table>

(which uses the GUW-interpolation method) where the maximum error was around 6e-05 for this example. The nested interpolation method completely failed to give a useful value for this 20th derivative. However, there are easier examples where all methods perform with excellent accuracy, confirming the implementations. There are also harder examples where all methods struggle to find accurate answers, and symbolic computation is even worse. We will consider how condition numbers of matrices associated with the methods help explain the error for typical examples. We also reflect on efficiency lessons, but now realize that the interpolation methods are unreliable in the current form. Possibly another different choice of directions might salvage the idea.

### 3. Direct method index reference scheme

A direct method relies on multiplication of corners of Taylor series coefficients, one for each multi-index \( \mathbf{i} = (i_1, \ldots, i_n) \) with order \( |\mathbf{i}| = i_1 + \ldots + i_n \leq d \). All standard library functions \( h = f(u(x)) \) are then given by one or two differential equations of the form \( h_{x_1} = v \ast u_{x_1} \), where the subscript is a partial derivative, so that a similar multiplication of corners performs a recurrence relation for the function. Details of every multivariate Taylor recurrence relation for the standard functions and operations are found in [5]. A conceptual geometric understanding of the \( n \)-dimensional relationships shows what must be accomplished. For a simple multiplication of two corners, the product value at an index \( \mathbf{i} \) is given by pulling out the box of all coefficients with index \( \mathbf{j} \leq \mathbf{i} \) component-wise from each corner and doing a dot product of all entries but with one box in reverse order. The analogous operation for the differential equation version strips off one (opposite) face of each box and dots parallel slices of each, also times the index of that slice. The key idea is that we can store one 2-dimensional reference array for each multi-index in \( n \) variables, with each row containing addresses for the elements in each slice. Let’s clarify with a specific example.

For practicality, all corners of Taylor coefficients are stored in a linear array corresponding to increasing \( |\mathbf{i}| \), and in reverse lexical ordering of multi-indices within the same order. Consider the specific example of multi-index \( \mathbf{i} = (2, 0, 3) \) which corresponds to linear index (or address) 45 in a corner for \( n = 3 \) variables to any order \( \geq 5 \). The reference array would store linear indices corresponding to every multi-index \( \leq (2, 0, 3) \) component-wise, as in Table 2. The actual reference array will store only the linear indices but the corresponding multi-index is included in the table for our understanding. For \( h = u \ast v \), where \( u \) and \( v \) have previously computed corners of Taylor coefficients, the coefficient of \( h \) at linear index 45 is computed by multiplying coefficients of \( u \) at the linear indices in Table 2 (reading across and down) times the coefficients of \( v \) at linear indices (reading backwards and up). For \( h = \exp(u) \), the relationship \( h_{x_1} = u_{x_1} \ast h \) results in the following value for the coefficient of \( h \) at linear index 45:

\[
h[45] = \frac{1}{2} \left( 1 \ast u[2, 7, 16, 30] \ast h[30, 16, 7, 2]^T + 2 \ast u[5, 13, 26, 45] \ast h[20, 10, 4, 1]^T \right)
\]
Observe that the first row of indices in Table 2 is not used for u and the last row is not used for h, the factors of 1 and 2 are simply the row numbers (origin 0), and the outside factor of 1/2 is just one over the last row index. This combination using the 2-dimensional reference array is called a ddot operation in [5] and is used to implement every Taylor coefficient recurrence formula for all standard functions. Even for any higher number of variables n and any multi-index, we use a 2-dimensional reference array. (In the degenerate case of only one non-zero entry in the multi-index, the array is one column). For an index such as (2,1,4,5,0), we could remove any of the faces with a non-zero entry, but choose the smallest non-zero (in this case 1 in the second entry) to minimize the number of multiplications in a ddot. The reference array for (2,1,4,5,0) would have two rows, the first for addresses corresponding to second index 0, and the second row corresponding to second index 1 (in increasing linear order). Then the number of multiplications between the two arguments in ddot would be half of the number of entries in the reference array.

Efficient generation of the reference array is done iteratively by combining previous arrays. This was found to be much, much faster than directly searching through a listing of all multi-indices and their addresses for the needed \( j \leq i \). (Such inefficient looping through multi-indices for every multi-index is at the heart of the very long one-time globals generation for GUW-interpolation in Table 1.) When the program generating reference arrays reaches linear index 45, a computational formula returns the addresses 30 and 26 (above and to the left in Table 2) corresponding to reducing the multi-index by one separately in each respective non-zero entry. The reference arrays for 30 and 26 have already been computed and consist of the first two rows and first three columns of Table 2, respectively. It is then simple to combine these into the reference array for 45. Actually, more non-zero dimensions require combining multiple arrays but it is not difficult.

4 Conclusion and further work

The larger error from the interpolation methods appears to be the most serious problem and indicates that direct methods are preferable until the problem is fixed. More test examples will refine what features of the function or program to be differentiated, cause the trouble. Testing will also directly compare problems with known derivatives, or where the true value can be determined with extended precision, as opposed to the artificial normalization of the tennis problem. It would be good to compare symbolic accuracy and speed, where previous research has shown that evaluation of huge symbolic results can be less accurate and slower than AD. Implementation of the interpolation methods could be made more efficient but it is doubtful that would improve the accuracy, so it may not be worthwhile.

4.1 Acknowledgements

We thank Andrea Walther for performing and sharing the computation error for the normalized tennis serve function using ADOL-C.

References


Mixed-language Automatic Differentiation

Valérie Pascual* and Laurent Hascoët†

March 29, 2016

As AD usage is spreading to larger and more sophisticated applications, problems arise for codes that use several programming languages. Many AD tools have been designed with one application language in mind. Only a few use an internal representation that promotes language-independence, at least conceptually. When faced with the problem of building (with AD) the derivative code of a mixed-language application, end-users may consider using several AD tools, one per language. However, this leads to several problems:

- Different AD tools may implemented very different AD models such as overloading-based versus source-transformation based or association-by-address versus association-by-name. These models are often not compatible.
- When selecting the source-transformation model (for efficiency of the differentiated code), performance of the differentiated code strongly depends on the quality of data-flow analysis, which must be global on the code. A global analysis with separate AD tools would require inter-tool communication at the level of data-flow analysis, which does not exist at present.

In any case, interoperable data-flow analysis between tools imply that the tools share their analysis strategy, which is almost never the case. Consequently, we think the only viable approach is to use a single tool, with a single internal representation and data-flow analysis strategy, therefore converting each source file to this unique representation regardless of its original language. It turns out that Tapenade [1] provides such an internal representation, accessible at present from C or Fortran sources.

Other AD tools provide a language-independent internal representation. OpenAD provides such a representation based on the XAIF formalism. However, this gives birth to two separate tools, OpenAD/F for Fortran, and ADIC2 for C. Still, it seems that their is no deep reason to prevent OpenAD application to mixed-language codes. We are lacking information about common architecture between TAF and TAC++ that would allow such mixed-language AD.

Rapsodia [2] was the first AD tool to support algorithmic differentiation in tangent mode of mixed-language components, specifically C++ and Fortran. As Rapsodia uses operator overloading, it performs no global analysis of the code. To our knowledge the extension of mixed-language differentiation with Rapsodia to adjoint mode is not yet provided.

1 Language standards and interoperability

Language standards often say very little about interoperability with other languages, leaving much freedom to compilers. Still, usage has progressively brought de facto standards, in particular between C and Fortran. The Fortran 2003 standard has specified its interaction with C in more detail. As far as AD is concerned, AD tools should in any case not commit to any specific interoperability strategy, and in particular to parameter-passing behaviors. Those might change with new languages and versions of languages. Instead, an AD tool must be able to handle a set of behaviors (hopefully small) from which one can describe all reasonable ways of parameter-passing.

The main issue raised by analysis and transformation of mixed-language codes is parameter-passing. Other issues are related to matching elements across languages, mainly types and procedures. These matching issues seem less complex than parameter-passing, but they should be discussed first.

Interoperability between types (and between variables of these types) across languages relies on identical memory representations built by compilers. Obviously interoperable types must match in the sense that they have the same structure, number of fields, and these fields must recursively be of interoperable types. Compilers often grant a natural interoperability between structured types. However Fortran 2003 provides the bind attribute to tell at compile time that a Fortran type has a C equivalent. It is essential to identify interoperable types in both languages, in particular because it may help distinguish candidate interoperable procedures according to the types of their arguments.

Similarly, interoperability between procedures relies on the compiler identifying the called procedure from another language, usually comparing procedure names, arguments number, and interoperable arguments types. Interoperable procedures in Fortran 2003 are declared with an explicit interface. The bind attribute allows one to associate the Fortran procedure name to the C procedure name. With Fortran 77 and Fortran 90, different conventions exist to associate both names: either by adding an underscore character at the end of the Fortran name, either with the same name, or with the name in uppercase, depending on the compilers.

*Corresponding Author, ECUADOR team, INRIA Sophia-Antipolis, France, Valerie.Pascual@inria.fr
†ECUADOR team, INRIA Sophia-Antipolis, France, Laurent.Hascoet@inria.fr
The parameter-passing strategy is already a property of each given, single language. Mixed-language calls may be at the interface between two different parameter-passing strategies, which adds extra complexity. Still all resulting parameter-passing behaviors should be eventually expressed using a small number of elementary tactics.

Inside a given language, parameter-passing may use one of a few classical strategies: call by value, call by reference, call by value-result, call by sharing, call by name . . . . Call by value is the most common strategy. In call by value, the argument expression is evaluated, and the resulting value is copied to the corresponding variable in the function. If the function assigns values to its parameters, only its local copy is assigned and the argument passed into the function call is unchanged when the function returns. In call by reference, a procedure receives a reference to a variable and can modify the variable used as argument. Call by value-result, also named call by copy-restore, is a special case of call by reference. It differs from call by reference when two arguments alias one another. Under call by reference, writing to one will affect the other. Call by value-result gives the function distinct copies, but result in the callers environment depends on which of the aliased arguments is copied back first. Call by sharing is a terminology used by languages such as Python, Java and other object oriented languages. It is analogous to call by value where the passed value is either the argument when it is of a primitive type or its address when is is an object. Whereas in call by value, the arguments are computed before calling the function, in call by name, each occurrence of the formal argument is replaced with the actual argument in the style of macro-expansion.

Consider now the classical mix of Fortran and C. Parameter-passing mechanisms differ in the two languages. In Fortran, call by value-result and call by reference are used. Fortran 2003 introduces the VALUE attribute to specify call by value. In C, call by value is the only parameter-passing mechanism. All parameters are passed by value, except for arrays, which are translated into the address of the first element. In C, one simulates a call by reference by passing a pointer to this parameter. In mixed language calls, the caller and the called procedure must agree on how parameters are passed, e.g. passing a pointer from C to a Fortran procedure if the parameter has no VALUE attribute.

We believe that every mixed-language parameter-passing strategy can boil down to a few simple behaviors at the time of entry into and return from the called procedure. At call time, we define what we call the passed argument, which may be the same as the actual argument, or the memory pointed to by the actual argument, or conversely the address of the actual argument, depending on the mixed-language strategy that must be captured. Then the internal memory corresponding to this passed argument is copied into the internal memory corresponding to the called procedure’s formal argument. At return time, the internal memory corresponding to the formal argument may be either copied back to the actual parameter, or not copied in which case it will vanish when the called procedure is popped from the call stack. When there is a back copy, it follows the link from the passed argument back to the actual argument: if they are the same, the copy is written into the actual argument, and if the passed argument is the destination of the actual, then the copy is written at the address designated by the actual argument.

Fig. 1 illustrates these behaviors for a few example multi-language calls, and also for pure Fortran calls differentiating the scalar case from the array case and for a C call using pointers. For each situation, we give the choice of the passed argument and the choice about back copy that implement the desired behavior. Anticipating on the next section, Fig. 1 also shows for each situation, the Translator object used by Tapenade to specify these choices to the data-flow analyses. We will now investigate how to extend our AD tool to analyze mixed-language codes, and how differentiation must be adapted.

2 Extension of Tapenade algorithms for interoperability

Tapenade was originally designed to support different imperative languages. The motivation was to share the model of the tool and its implementation between these languages (at present, Fortran and C). We believe that this architecture also lets us deal with mixed-language source for a minimal implementation effort, affecting only a few components of the tool.

Tapenade represents a code as a call graph whose nodes represent procedures and arrows represent calls. Each call graph node contains a flow graph, in which nodes are blocks of elementary instructions (in particular calls), and arrows represent control jumps. At present, Tapenade called on C source or on Fortran source builds an internal representation of the same nature, using the same components for procedures, instructions, variables, types . . . We exploit this unicity of representation to deal with mixed-language codes, in particular mixing Fortran and C.

Let’s first take a look at matching of types and procedures. Two interoperable entities are represented with the same internal representation in the AD tool. This representation distinguishes each component of structured types, and distinguishes pointer variables from their pointee destination variables. Representation of arrays on the other hand do not distinguish array elements, and therefore we need not worry about their possibly different memory layout in Fortran and C. Sometimes it is not possible to preserve this nice structural matching, for instance for the complex Fortran type which should match a 2-fields structure in C. Then the correspondence must be enforced by implementation, and possibly incurs some loss of information.

The question of procedure matching amounts to finding, for a given procedure call, the node of the call graph that will be effectively called. This depends on the mixed-language conventions on procedure names and on type matching. Procedure name conventions may vary, and Tapenade offer parameterization to define one convention, using command-line arguments or directives at the call site. It also interprets attributes of Fortran 2003. For instance, these parameters let us specify that inside a Fortran code all calls to a procedure named FOO will connect to a C
Procedure named `foo`.

<table>
<thead>
<tr>
<th>C calls Fortran</th>
<th>C calls Fortran, by value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>float *y;</code></td>
<td><code>float y;</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>bar(y):</code></td>
<td><code>bar(y):</code></td>
</tr>
<tr>
<td>- Passed argument is <code>*y</code></td>
<td>- Passed argument is <code>y</code></td>
</tr>
<tr>
<td>- Upon return, value of <code>V</code> is copied back into <code>*y</code></td>
<td>- Upon return, no copy takes place into <code>y</code></td>
</tr>
<tr>
<td><strong>Translator:</strong> <code>V -&gt; *y</code> (Back copy)</td>
<td><strong>Translator:</strong> <code>V -&gt; y</code> (No back copy)</td>
</tr>
</tbody>
</table>

Fortran calls C

<table>
<thead>
<tr>
<th>REAL <code>X</code></th>
<th><code>void foo(float *a)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>CALL FOO(X)</code></td>
<td><code>CALL FOO(X)</code></td>
</tr>
<tr>
<td>- Passed argument is address of <code>X</code></td>
<td>- Passed argument is <code>y</code></td>
</tr>
<tr>
<td>- Upon return, no copy takes place into <code>&amp;X</code></td>
<td>- Upon return, no copy takes place into <code>y</code></td>
</tr>
<tr>
<td><strong>Translator:</strong> <code>a -&gt; &amp;X</code> (No back copy)</td>
<td><strong>Translator:</strong> <code>a -&gt; y</code> (No back copy)</td>
</tr>
<tr>
<td><code>*a -&gt; X</code> (Back copy)</td>
<td><code>*a -&gt; *y</code> (Back copy)</td>
</tr>
</tbody>
</table>

Fortran calls C

<table>
<thead>
<tr>
<th>REAL <code>X</code></th>
<th><code>void foo(float *a)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>CALL GEE(X)</code></td>
<td><code>CALL GEE(Y(10))</code></td>
</tr>
<tr>
<td>- Passed argument is <code>X</code></td>
<td>- Passed argument is <code>&amp;(Y(10))</code></td>
</tr>
<tr>
<td>- Upon return, value of <code>V</code> is copied back into <code>X</code></td>
<td>- Upon return, no copy takes place into <code>Y</code></td>
</tr>
<tr>
<td><strong>Translator:</strong> <code>V -&gt; X</code> (Back copy)</td>
<td><strong>Translator:</strong> <code>B -&gt; &amp;Y(10)</code> (No back copy)</td>
</tr>
</tbody>
</table>

Fortran calls Fortran, scalars

<table>
<thead>
<tr>
<th>REAL <code>X</code></th>
<th><code>SUBROUTINE GEE(V)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>CALL GEE(X)</code></td>
<td><code>CALL GEE(Y(10))</code></td>
</tr>
<tr>
<td>- Passed argument is <code>X</code></td>
<td>- Passed argument is <code>&amp;(Y(10))</code></td>
</tr>
<tr>
<td>- Upon return, value of <code>V</code> is copied back into <code>X</code></td>
<td>- Upon return, no copy takes place into <code>Y</code></td>
</tr>
<tr>
<td><strong>Translator:</strong> <code>V -&gt; X</code> (Back copy)</td>
<td><strong>Translator:</strong> <code>B -&gt; &amp;Y(10)</code> (No back copy)</td>
</tr>
</tbody>
</table>

Fortran calls Fortran, arrays

Parameter passing comes into play during data-flow analysis. All data-flow analyses (e.g. in-out, activity, liveness) inherit from a base analysis class that provides primitives to transfer data-flow information between a caller procedure and a callee. Most adaptations to mixed-language code must be done in this base class. This information transfer is driven by an object we call a Translator, which describes how actual arguments are matched with formal arguments. Variables, and in particular those in arguments, may have a finer structure. In addition to the classical scalar and arrays that existed for Fortran 77, languages have introduced variables of structured type and pointers. Since the data-flow properties that we analyze may be different for each component of these structured objects, we distinguish all of their elementary components. For example, a formal argument of type `mystruct *arg` where `mystruct` is a record:

```c
struct mystruct {
  int numElems;
  float *elems;
}
```

is represented by 4 elementary formal arguments, for `*(arg->elems)`, `arg->elems`, `arg->numElems`, and the top-level `arg` respectively. To each elementary formal argument of the called procedure, the Translator associates the corresponding elementary actual argument at the call site, which is either an expression or an elementary variable known to the calling procedure. In addition, the Translator associates to each elementary formal argument a boolean that specifies whether it must be copied upon return. For each example situation in Fig. 1, the Translator that implements the desired behavior is shown below the textual description of the behavior, as a set of arrows from formal elementary argument to actual elementary argument and back copy boolean. The rule of thumb is that the Translator associates the root formal argument with the passed argument.

The “Fortran calls C” situation deserves further comment: since the C formal argument is a pointer to a float, there are in fact two elementary formal arguments, one for `a` and one for `*a`. The same happens for “C calls C”. As `a` is associated with the passed argument `&X`, `*a` is naturally associated with `*(&X)` in other words with `X`. It is in fact thought this second elementary argument that the actual value of (or information regarding) `X` is propagated to the...
callee. Consequently, even if no back copy is done upon return into $X$ itself, every write into $\star a$ in $\text{foo}$ is automatically reflected into $X$.

The way each data-flow analysis, which computes a given data-flow value, uses the specification from the $\text{Translator}$ can be sketched as follows: immediately before the call, and for each elementary formal argument (left column of $\text{Translator}$), we retrieve the corresponding elementary actual argument (right column), and we retrieve the current data-flow value for it. The initial data-flow value of the elementary formal argument is set to this retrieved value. Analysis can then run on the called procedure. Upon return from the call, the top-level elementary arguments that bear the “No back copy” retain the data-flow value they had before the call. For all other elementary arguments the data-flow value of the elementary formal argument upon return is back copied into the data-flow value of the elementary actual argument. This description applies to forward data-flow analyses. Adaption to backward analyses require only minor technical changes.

```c
void bar (float a, float *b);
void foo (float *x, float *y) {
    bar(*x, y);
}

void foo_b (float *x, float *xb,
            float *y, float *yb) {
    bar_b(*x, xb, y, yb);
}
```

Figure 2: Mixed-language differentiation with Tapenade, C calling Fortran case

The main impact of dealing with mixed-language code on actual differentiation is related to arguments being copied to the formal arguments and not being copied back. For adjoint differentiation, the consequence is that the copied formal argument must not be identified with the actual argument of the call. In particular, the adjoint formal argument must be a different variable from the adjoint actual argument, and the adjoint code must follow the pattern illustrated in Fig. 2: variable $u$ is the original formal argument, which is not copied back because of its “value” attribute. The differentiated actual argument is (conceptually) $ub0$, whereas the differentiated formal argument is $ub$. Only at the end of $\text{BAR}_B$, the differentiated formal argument $ub$ is added into the differentiated actual argument $ub0$. Consequently, $ub0$ must be passed a reference $xb$, even if the corresponding non-differentiated argument was passed a dereferenced $*x$. In other words the adjoint of a passed by value argument must sometimes be passed by reference.

When differentiating a procedure that calls a function $F$ that returns a value, the last argument of the differentiated procedure contains the result of $F$. This argument and its corresponding passed parameter must be declared and used according to the parameter-passing mechanism, e.g. by passing the address of the argument when calling a Fortran function from C, instead of just passing the argument for a C function.

3 Conclusion and further work

Tapenade is now able to differentiate codes that mix C and Fortran, with calls in either direction. Apart from short non-regression tests, we are validating this extension on the “CalculiX” finite element library. The architecture of Tapenade allowed us to implement this functionality quite easily, with modifications in only a few component of the tool. We still need to study interoperability of global variables, e.g. using the $\text{bind}$ statement of Fortran 2003. As the implementation is still at an early stage, we need to experiment on a few representative examples.

References


Operator Overloading-based Automatic Differentiation of C++ Codes on Emerging Manycore Architectures

Eric Phipps† and H. Carter Edwards‡

March 2016

1 Introduction

A challenge for applying automatic differentiation (AD) techniques to computations relevant to high-performance scientific computing is the efficient differentiation of parallel computations. Historically computations have been parallelized through the use of distributed message passing libraries such as MPI. Accordingly, AD software and techniques have been developed to differentiate these programs [1, 2, 3]. In the recent past, multicore architectures have become common within scientific computing, and techniques for differentiating programs parallelized through shared-memory parallel environments such as OpenMP have been developed [4, 5, 6, 7]. Even more recently however, computer architectures have increased in complexity and diversity with the addition of manycore architectures such as general-purpose graphics processing units (GPUs) and Intel Xeon Phi. A significant challenge for simulation code developers wishing to harness the power of these parallel architectures is how to develop a single code base that is both portable and performant across this diversity of architectures. Compounding this challenge is how to effectively differentiate code for these architectures in such a manner that the resulting derivative code is portable and performant.

In this work, we consider the forward-mode differentiation of C++ codes written for these emerging manycore architectures using operator overloading techniques. We first review the Sacado operator overloading-based AD library for C++ applications [8, 9] which provides the foundation of our work. We then describe the key abstractions that are required for achieving portable and performant code on manycore architectures, and how these abstractions are implemented in a library called Kokkos [10, 11]. We next describe how AD must be integrated into these abstractions to achieve portable and performant derivative code, and how this integration is implemented in Sacado and Kokkos. Next we describe several computational experiments that demonstrate the performance of our approach on several common manycore architectures. Finally we summarize our conclusions and future work. Due to space constraints for the extended abstract, we only sketch our ideas and results here, leaving an in-depth discussion to the full paper.

2 Operator Overloading-based AD with Sacado

Sacado is an operator overloading-based automatic differentiation library designed for C++ applications. It provides the forward, reverse, and Taylor polynomial modes of AD, and allows these modes to be applied recursively for the computation of higher derivatives. It incorporates the Rad tool [12] to provide reverse mode capabilities while the forward mode tools are based on the original Fad library [13]. The forward mode tools use expression templates [14] to eliminate most of the overhead associated with operator overloading, and employ techniques such as expression-level reverse mode to further improve efficiency [9]. In this work, we are only concerned with the forward mode tools, which have been used extensively for Jacobian and parameter sensitivity evaluation within the context of discretized systems of partial differential equations (PDEs). The Sacado tools themselves are only designed for the differentiation of small, dense operations and don’t provide any directly capability for managing sparsity or parallelism. In the context of PDEs discretized over a mesh, sparsity is managed by only applying AD at the “element” or mesh cell level of the discrete residual evaluation [15, 8], which are typically small, dense (or nearly dense) computations. The resulting derivatives are then manually assembled into parallel, distributed linear algebra data structures. To facilitate the use of Sacado in large-scale, complex, simulation codes, a template-based generic programming approach has been developed [16, 17], whereby the relevant computations are templated on the scalar type and then instantiated on a variety of Sacado AD data types. These techniques have been applied successfully in numerous large-scale, parallel scientific computations [18, 19, 20, 21].
3 Manycore Performance Portability with Kokkos

Kokkos is a programming model and C++ library that enables applications to implement thread scalable computations that are performance portable across diverse manycore architectures such as a multicore CPU, Intel Xeon Phi, and NVIDIA GPU. Kokkos works by defining common parallel patterns such as for-each, reduction, scan, and task-dag. Users supply code bodies to be invoked within these patterns and multidimensional arrays of data upon which these patterns and code bodies operate. Performance portability is realized through the integrated mapping of the following three components on to the underlying manycore architecture:

1. Code is mapped onto the target architecture’s best performing threading mechanism, e.g., pthreads or OpenMP on CPUs, OpenMP on Xeon Phi, and CUDA on NVIDIA GPUs.

2. Parallel execution is mapped with architecture-appropriate scheduling, e.g., each pthread/OpenMP thread is given a contiguous range of the parallel iteration space while each GPU thread is given a thread-block-strided range of the parallel iteration space.

3. Multidimensional arrays are given an architecture-appropriate layout, e.g., on CPUs/Phi’s arrays have a row-major layout and on GPUs arrays have a column-major layout.

These mappings are applied without modification of application source code. The target architecture is specified when the source code is compiled and the appropriate mappings are inserted through C++ template instantiation mechanisms (i.e., through template meta-programming).

To achieve performance, it is critically important to match the multidimensional array layout to the chosen parallel scheduling. This will be demonstrated in the full paper by examining a computational kernel implemented in Kokkos for computing a dense-matrix vector product:

$$c = Ab, \ b, c \in \mathbb{R}^m, \ A \in \mathbb{R}^{m \times n}. \quad (1)$$

The performance of this kernel will be measured on three contemporary architectures: Intel Sandy Bridge CPU, NVIDIA K20X GPU, and Intel Xeon Phi accelerator. By varying the layout of the multidimensional array $A$ on each architecture, substantial variations in performance can be observed.

4 AD on Manycore Architectures

Matching the multidimensional array layout to the architecture’s best-performing parallel scheduling has dramatic implications for automatic differentiation on these architectures. For an operator overloading-based approach like Sacado, the AD operations will be applied in a thread-local context where each active variable will be replaced by a corresponding Sacado forward-mode AD object (consisting of a value field and derivative array). When these active variables are contained within a multidimensional array, a (possibly) large stride is created between, e.g., a given component of the derivative array, for consecutive elements of the outer multidimensional array. For architectures with a row-major ordering (CPU/Phi), this is not harmful since each thread is given a contiguous range of the multidimensional array to process. For each element of this range, the thread will evaluate the AD overloaded operators just as in the thread-serial case. Generally good AD performance is achieved because (1) the derivative loops implemented inside the overloaded AD operators are typically simple for the compiler to auto-vectorize and (2) the elements of the derivative array are stored consecutively enabling the use of packed load/store operations.

Recall for GPU architectures however, multidimensional arrays are stored in column-major order and each thread-block is given a contiguous range of the array to process, where each thread iterates over that range with a stride determined by the size of the thread block. This ensures consecutive GPU threads within a thread-block access consecutive entries of the multidimensional array (coalesced memory access). However in the AD context, each GPU thread will now be evaluating the overloaded AD operators applied to elements of the multidimensional array, which in turn involve iterating over the components of the derivative array. Due to the large stride between a given component of the derivative array for consecutive multidimensional array elements, consecutive GPU threads will no longer access consecutive values in memory, destroying coalesced access and substantially reducing performance.

Overcoming this problem requires deeper integration of the AD library with the parallel programming environment to manipulate parallel scheduling and data layout in a manner appropriate for AD on the given architecture. There are two possible approaches: (1) modify the data layout of multidimensional arrays of AD objects so that coalesced accesses are maintained for column-major orderings by striding the derivative array or (2) modify the parallel execution so that the GPU threads of a given thread-block iterate over the derivative components of a single element of the multidimensional array. Both approaches have been implemented in Sacado and Kokkos, giving the user the freedom to choose the best approach for the given derivative computation.

Sacado can specialize the multidimensional array data structure provided by Kokkos. For a rank-$R$ multidimensional array of Sacado forward AD objects with $p$ derivative components, internally this specialization creates a rank-$(R + 1)$ multidimensional array of floating-point values. The additional dimension is the last dimension of the array, has size $p + 1$, and stores the value field and derivative components for each AD object (thus internally a rank-$R$ array of AD objects is not created). The multidimensional array element access operators are specialized to create
temporary AD proxy objects containing value and derivative array pointers that reference the corresponding data within the inner rank-\((R+1)\) array. The layout of the multidimensional array is applied to the inner rank-\((R+1)\) array, and for a column-major ordering, the proxy objects are supplied a non-unit stride for accessing elements of the derivative array. Thus consecutive GPU threads still access consecutive values in memory as each thread iterates over its assigned derivative arrays, maintaining coalesced accesses. This approach requires no modification of the parallel kernel body or execution policy defining the parallel scheduling for Kokkos. Furthermore all derivative data is stored in one contiguous allocation improving memory access patterns and eliminating possibly numerous small dynamic memory allocations during array construction.

Alternatively, Sacado can specialize the multidimensional array data structures in Kokkos but always store the derivative components consecutively. Internally a rank-\((R+1)\) array is still created to provide a contiguous allocation of derivative data, and temporary proxy objects are created by the array access operators, however the layout is manipulated so these proxy objects always have unit stride. However for good performance, the execution policy controlling parallel scheduling must be manipulated when launching the parallel kernel. Sacado provides a mean for doing this whereby consecutive GPU threads iterate over the elements of the derivative array for a given element of the multidimensional array. Such an approach can work well when the derivative array length \(p\) is large, but requires manual modification of the kernel invocation.

In summary, the deep integration of Sacado with Kokkos enables easily exploration and management of the mapping of AD data and computations onto diverse manycore architectures, and thus tune those mappings for performance with few, if any, changes to the application source code.

5 Computational Results

Due to space constraints, computational results cannot be included in the extended abstract. In the full version of the paper, we will describe and demonstrate the use of both approaches described above for the kernel implementing Eq. 1 for all three architectures. We will demonstrate how the use of the natural AD layout causes performance degradation on the GPU in comparison to a hand-coded derivative, and how both approaches mitigate this.

6 Conclusions

In this work, we described the performance problems that arise when applying automatic differentiation techniques on emerging manycore architectures such as GPUs. These difficulties are due to the loss of coalesced memory accesses induced by the natural layout of multidimensional arrays storing derivative data. Furthermore we described approaches for eliminating these performance losses, described how these approaches are implemented within the Sacado AD and Kokkos performance portability libraries, and demonstrated their effectiveness. It is worth noting that the observed performance issues, as well as their resolution are not specific to Sacado, Kokkos, or operator overloading-based AD in general. For example, one could imagine manipulating multidimensional array data layouts and parallel kernel execution policies to achieve the same effects described here in the context of source transformation-based AD techniques. The work described here only considered first derivatives, and in the future we are interested to explore the extension of these ideas to higher derivatives.

7 Acknowledgements

This work was supported by the Advanced Technology Development and Mitigation program of the U. S. Department of Energy’s National Nuclear Security Administration and the Laboratory Directed Research and Development program of Sandia National Laboratories.

References


How AD Can Help Solve Differential-Algebraic Equations

John D Pryce*          Nedialko S Nedialkov†

April 2016

1 Introduction

DAE formulation and basic ideas. Industrial engineering is highly interdisciplinary and will become more so. Modeling a car’s engine, for instance, blurs the boundaries between mechanical engineering, electrical engineering, chemical engineering, and thermodynamics. To support this, software tools for analysing and simulating engineering artefacts (e.g. Dymola, gPROMS, MapleSim or Simulink), and special languages such as Modelica, increasingly use equation-based modeling, which describes system components by the basic physical laws they obey (Force = Mass × Acceleration, etc.)

Such a model is inherently interdisciplinary, as mathematically it is just a set of n equations in the same (usually) number n of state-variables and their time derivatives—a differential-algebraic equation (DAE) system, describing:

• the behaviour of individual system components, independent of whether this involves chemistry, mechanics, etc.;
• the relations between components;

and hence the time evolution of the system.

It is always possible by introducing extra variables to reduce the DAE to first order, i.e. to one in which only first derivatives occur, as one does for an ordinary differential equation (ODE) system. Then it can be formulated mathematically as \( F(t, \mathbf{x}, \dot{\mathbf{x}}) = 0 \) where \( t \) is time, \( \mathbf{x} \) is the vector of \( n \) state variables \( x_j = x_j(t) \), and \( \dot{\mathbf{x}} \) is \( \frac{d\mathbf{x}}{dt} \), which is the form accepted by many codes for the DAE initial value problem, such as relatives of DASSL [1, 2].

However we prefer a more flexible form allowing arbitrary higher derivatives:

\[
  f_i(t, \text{the } x_j \text{ and derivatives of them }) = 0, \quad i = 1, \ldots, n. 
\]

This lets one formulate many problems more concisely, and solve by our DAETS initial-value code more efficiently.

Solving a DAE can be very different from solving an ODE. The system

\[
  x_1 - u(t) = 0, \quad x_1 - \dot{x}_2 = 0
\]

is a DAE (where \( u(t) \) is a given driving function). Solving it requires integrating the driving function: \( x_2 = \int u(t) dt + C \) where \( C \) is a constant, and it is really an ODE, with one degree of freedom. But also a DAE is the system

\[
  x_1 - u(t) = 0, \quad \dot{x}_1 - x_2 = 0,
\]

whose unique solution has \( x_2 = \dot{u}(t) \). Solving it requires differentiating the driving function, so it is not like an ODE at all; for \( t \) where \( u \) is not differentiable, no solution exists. Such behaviour is common, particularly in control problems.

Specifying initial values is not obvious for DAEs. Take the notorious simple pendulum in cartesian coordinates, where \( x(t), y(t), \lambda(t) \) are the state variables and \( G, L \) are positive constants:

\[
  0 = A = \ddot{x} + x\lambda \\
  0 = B = \ddot{y} + y\lambda - G \\
  0 = C = x^2 + y^2 - L^2
\]

One cannot specify any initial position \((x, y)\): it must satisfy the “overt” constraint \( C = 0 \) of lying on the circle. Similarly one cannot specify an arbitrary initial velocity \((\dot{x}, \dot{y})\): it must satisfy the “hidden” constraint \( C = 0 \), that is \( 2(\dot{x}\ddot{x} + \dot{y}\ddot{y}) = 0 \), which says motion must be tangential to the circle. Thus one can describe the configuration at a given time \( t \) by a point in \((x, y, \dot{x}, \dot{y})\)-space that must obey the equations \( C = 0, \dot{C} = 0 \) defining the manifold \( \mathcal{M} \) of consistent points at a given \( t \). Its dimension, here 2, is the number of degrees of freedom—how many independent initial values one can prescribe.

Various kinds of non-negative integer index have been devised to measure how far the DAE is from being an ODE. The differentiation index \( \nu_i \) is the number of times it is necessary to differentiate one or more of the equations (w.r.t. \( t \)) to obtain an ODE in all the components, such that every solution of the DAE is a solution of the ODE. The first example above has index 1 (the ODE could be \( \dot{x}_1 = \dot{u}(t), \dot{x}_2 = u(t) \)). The second has index 2 (the ODE could be \( \dot{x}_1 = \ddot{u}(t), \dot{x}_2 = \dddot{u}(t) \), obtained by differentiating the first equation twice).

*Corresponding Author, Cardiff University, prycejd1@cardiff.ac.uk
†McMaster University, nedialk@mcmaster.ca
Index reduction and structural analysis. Many numerical methods for higher-index DAEs start with index reduction: augmenting the DAE by time-derivatives of some of its equations to produce a DAE of larger size and smaller index. The index $\nu_d$ does not help much in finding how to do this. A minor reason—a remediable glitch in its definition—is that it tends to overestimate the amount of differentiation needed: e.g., one can solve the first example above without differentiating, and the second by doing so once. A more serious reason is that to compute $\nu_d$ from its formal definition (really, estimate it, since matrix condition is involved) entails heavy linear algebra.

Thus index reduction—in particular the dummy derivative method below—is usually done by discrete structural analysis (SA) algorithms based on the DAE’s sparsity structure, namely what variable-derivatives occur in what equations. The essential idea is to find a number $c_i$ of times to differentiate the $i$th equation that gives a structurally nonsingular system for the resulting highest, $d_j$ th, derivatives of the $x_j$. That is, the $n \times n$ incidence matrix $S$ with

$$S_{ij} = \begin{cases} 1 & \text{if } x_j^{(d_j)} \text{ occurs in } f_i^{(c_i)}, \\ 0 & \text{otherwise}, \end{cases}$$

is structurally nonsingular (can be permuted to put 1’s on the diagonal). Such offset vectors $c = (c_1, \ldots, c_n)$, $d = (d_1, \ldots, d_n)$ can always be found, unless the DAE is “structurally ill-posed”, which usually indicates a modeling error. There exist unique elementwise smallest non-negative $c, d$, the canonical offsets, which we assume chosen henceforth.

For numerical solution, the key further requirement is that the $n \times n$ system Jacobian

$$J = \left( \frac{\partial f_i^{(c_i)}}{\partial x_j^{(d_j)}} \right)_{i,j=1:n} \quad (5)$$

be nonsingular at some consistent point. Then we call the DAE SA-friendly. Assuming suitable smoothness of the $f_i$, solutions then exist locally through this point and through any nearby consistent points. Solvable DAEs that are not SA-friendly occur in applications, though rarely; handling them is an active research area [3, 4].

There is evidence that in the 19th century Jacobi knew the essence of DAE structural analysis, even an early form of the dual linear assignment problem. In the computer era, key SA developments are:

- Pantelides (1988) [5] on computing consistent points. He finds the numbers $c_i$ by a bipartite graph algorithm.
- Mattsson & Söderlind’s dummy derivatives (DDs) method (1993) [6], using the $c_i$ found by Pantelides’ method.
- Pryce (1998, 2001) [7, 8] finds $c_i$, $d_j$ by the signature matrix method ($\Sigma$-method).

The Pantelides and Pryce methods apply to DAEs of arbitrary index and give essentially equivalent results except that the former was described only for DAEs of first order. Hence DDs can equally well be based on the $\Sigma$-method.

The $\Sigma$-method involves solving a linear assignment problem (a kind of linear programming problem). Its original motivation was an algorithm to solve DAEs by Taylor series expansion, which led to the Nedialkov–Pryce code DAETS (2006 onward) [9].

Each of these three applications requires non-singularity of the system Jacobian (5), so one can say

The DAEs for which Pantelides’ method can find consistent points, for which the dummy derivative method is applicable, and which DAETS can solve, are precisely the SA-friendly DAEs as defined above.

2 Dummy derivatives

Various index reduction methods have been used that convert the DAE to an ODE with more degrees of freedom than the DAE. Then the solution paths of the DAE are a proper subset of those of the ODE. This tends to be bad numerically, because errors cause drift from the consistent manifold that is often exponential once it starts.

DDs by contrast are a systematic way to form an equivalent ODE with exactly as many DOF as the DAE. If one regards the DAE as a flow on the consistent manifold $\mathcal{M}$, DDs describes the flow in a local coordinate system for $\mathcal{M}$. Thus numerical drift can only be within $\mathcal{M}$, where it is less harmful. However if the path leaves the patch of $\mathcal{M}$ where the coordinate system is nonsingular, one must choose new coordinates. This need for DD switching, or pivoting, complicates a numerical algorithm.

The following description of the DDs process is equivalent to that in [6]. However it supports an implementation that makes the housekeeping of DD switching almost trivial.

First form the derivatives of each $f_i = 0$ up to the $c_i$ th, forming the augmented system of $N_f = n + \sum_i c_i$ equations:

$$f_i^{(l)} = 0, \quad l = 0 : c_i, \quad i = 0 : n. \quad (6)$$

Its unknowns are the $N_x = n + \sum_j d_j$ derivatives of the state variables $x_j$ up to the $d_j$ th. View them for now as unrelated algebraic unknowns that we call items, and to emphasise this denote them $x_{jl}$:

$$x_{jl} \text{renames } x_j^{(l)}, \quad l = 0 : d_j, \quad j = 0 : n. \quad (7)$$
The system has fewer equations than variables by the amount $\sum_j d_j - \sum_i c_i$, which equals the number DOF of degrees of freedom. To balance this, DDs judiciously selects a number DOF of items to be state items and adds DOF new equations stating a genuine differential (not algebraic) relation between a state item and its next higher derivative:

$$\dot{x}_{jl} = x_{j, l+1}$$

for $(j,l)$ in a DOF-element set $S$ of index pairs. One requires $l < d_j$, so that $x_{j, l+1}$ shall also be an item. The state vector $x_S$ formed by the $x_{jl}$ for $(j,l) \in S$ is the associated local coordinate system of the manifold $\mathcal{M}$.

Central to the DDs method is an algorithm that at any consistent point, using rank properties of submatrices of the (nonsingular) Jacobian $J$ in (5) there, finds $S$ such that (6, 8) form an index-1 DAE locally equivalent to the original system (1).

Actually, though generally called index-1, it has the stronger property of being an implicit ODE, defined as an SA-friendly DAE whose offsets $c_i$ are all zero. It is then possible (theoretically by the Implicit Function Theorem, practically by root-finding) to solve (6, 8) for all the items $x_{jl}$ as functions of $t$ and the state items.

This includes the items on the right hand side of (8), which therefore yields an ODE system of size DOF,

$$\dot{x}_S = F(t, x_S),$$

locally equivalent to the DAE formed by the $N_x$ equations (6, 8) and hence to the original DAE.

**Example.**

For the pendulum we modify the general $x_{jl}$ notation—e.g., renaming $x, \dot{x}, \ddot{x}$ to $x_0, x_1, x_2$—to get 5 equations in 7 unknowns:

<table>
<thead>
<tr>
<th>Augmented system</th>
<th>After renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 = A = \ddot{x} + x\lambda</td>
<td>0 = A_0 = x_2 + x_0\lambda_0</td>
</tr>
<tr>
<td>0 = B = \dot{y} + y\lambda - G</td>
<td>0 = B_0 = y_2 + y_0\lambda_0 - G</td>
</tr>
<tr>
<td>0 = C = x^2 + y^2 - L^2</td>
<td>0 = C_0 = x_0^2 + y_0^2 - L^2</td>
</tr>
<tr>
<td>0 = \ddot{\lambda} = 2(x\dot{x} + y\dot{y})</td>
<td>0 = C_1 = 2(x_0\dot{x}_1 + y_0\dot{y}_1)</td>
</tr>
<tr>
<td>0 = \ddot{\lambda} = 2(x\dot{x} + \dot{x}^2 + y\dot{y} + \dot{y}^2)</td>
<td>0 = C_2 = 2(x_0\dot{x}_2 + \dot{x}_1^2 + y_0\dot{y}_1 + \dot{y}_1^2)</td>
</tr>
</tbody>
</table>

One possible state vector is $x_S = (x, \dot{x}) = (x_0, x_1)$. It is easily seen that provide $y \neq 0$ one can find all the items as functions of these two, hence the pendulum DAE is equivalent to an ODE (9) in this $x_S$ when $y \neq 0$.

In fact one can choose any of $(x, \dot{x}), (x, \dot{y}), (y, \dot{x}), (y, \dot{y})$ as state vector (one must choose one undifferentiated variable and one first derivative). But only the first and last of these are “convenient” for AD, as the next section shows.

## 3 AD and DAEs

How can an AD tool help automate numerical solution methods for DAEs of the kind discussed so far?

First, the tool must support $d/dt$ as a first-class operator, of equal status with $+, \times, \sin()$, etc. Else, whether using source transformation or operator overloading, it cannot even understand a representation of a DAE in the form (1), let alone manipulate it. At present widely used tools such as ADOL-C and dcc/dco [10, 11] do not have this ability.

Our solver DAETS uses the Ole Stauning’s AD package FADBAD++ [12], written in C++. It did not originally include $d/dt$ but at our request, Stauning included the operator `Diff()` such that `Diff(i, q)` means $d^q dt^q$. For instance, straightforward code for the pendulum, as in the DAETS user guide, is as follows. (It could be made more readable by renaming $z[0], z[1], z[2]$ as $x, y, \lambda$.)

```cpp
template <typename T>
void fcn(T t, const T *z, T *f, void *param) {
  // z[0], z[1], z[2] are x, y, lambda.
  const double G = 9.81, L = 10.0;
  f[0] = Diff(z[0], 2) + z[0]*z[2];
  f[1] = Diff(z[1], 2) + z[1]*z[2] - G;
  f[2] = sqr(z[0]) + sqr(z[1]) - sqr(L);
}
```

Second, an AD tool must be able to differentiate the $f_i$ selectively. For instance in the pendulum, $A$ and $B$ are to be left alone, and $C$ differentiated twice.

At first sight this seems to require a tool based on source code transformation, which could generate code for the last two equations in (10), for instance. However Nedialkov observed that this is not the case. The key is not to treat different derivatives of a given variable in isolation, but store them together as a truncated power series. In fact storage in DAETS is already organised this way. For instance in the pendulum, the unknowns form three objects

- $x = (x_0, x_1, x_2)$ order 2 power series,
- $y = (y_0, y_1, y_2)$ order 2 power series,
- $\lambda = (\lambda_0)$ order 0 power series.
Then normal Taylor series AD by overloading (as in FADBAD++, say) produces the necessary derivatives. For instance evaluating $C = x^2 + y^2 - L^2$ proceeds via the following intermediate steps:

<table>
<thead>
<tr>
<th>input $\mathbf{x}$</th>
<th>$= (x_0, x_1, x_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{y}$</td>
<td>$= (y_0, y_1, y_2)$</td>
</tr>
</tbody>
</table>

**compute** $v_1 = x^2 = (x_0^2, 2x_0x_1, 2x_0x_2 + x_1^2)$

$v_2 = y^2 = (y_0^2, 2y_0y_1, 2y_0y_2 + y_1^2)$

$v_3 = v_1 + v_2 = (x_0^2 + y_0^2, 2(x_0x_1 + y_0y_1), 2(x_0x_2 + y_0y_2 + x_1^2 + y_1^2))$

$C = v_3 - \text{const}(L^2) = (x_0^2 + y_0^2 - L^2, 2(x_0x_2 + y_0y_2 + x_1^2 + y_1^2))$

Returning an order 2 power series object $C$ holding the needed values $(C, \nabla C) = (C_0, C_1, C_2)$.

Evaluating $A$ and $B$ is similar. $\text{diff}(:, 2)$ converts, e.g., the order 2 series $x = (x_0, x_1, x_2)$ to the order 0 series $(2x_2)$. Thus $A, B$ are returned as order 0 series $A = (A_0) = (2x_2 + x_0\lambda_0)$, $B = (B_0) = (2y_2 + y_0\lambda_0)$.

In the context of DDs and reducing the DAE to an explicit ODE, state item values, say values $\mathbf{x}_S = (x_0, x_1)$ are given; the 5 items $\mathbf{x}_F = (x_2, y_0, y_1, y_2, \lambda_0)$ are trial values that produce 5 residual values $\mathbf{r} = (A_0, B_0, C_0, C_1, C_2)$. By root-finding using suitable Jacobians (found by methods not described here) we find $\mathbf{x}_F$ that makes $\mathbf{r} = 0$, thus solving for $\mathbf{x}_F$ as a function of $\mathbf{x}_S$. Extract $x_2$ from $\mathbf{x}_F$ to form $(x_1, x_2)$, which is $\mathbf{x}_S$. This implements the function $\mathbf{F}$ in (9).

To make this work, the state items must comprise a contiguous set of derivatives of each variable, with no gaps. (Hence, see below (10), $(x, \dot{y})$ and $(y, \dot{x})$ are not useful state vectors for the pendulum.) That is, $S$ must have the form $\{(j, l) | 0 \leq l < \delta_j, j = 1:n\}$, where $\delta = (\delta_1, \ldots, \delta_n)$ is an integer DD-acute vector with $0 \leq \delta_j \leq d_j$ and $\sum_j \delta_j = \text{dof}$, which uniquely specifies the DD scheme currently in use. DD switching can be based on changing this $\delta$, and following through the consequences for various associated index sets and Jacobian-related matrices.

Initial experiments by Nedialkov confirm that this is a viable way to implement dummy derivatives, including the root-finding that produces (9), using methods already provided by the DAETS classes.

## 4 Conclusions and further work

We aim to present further developments of this work at the conference in September.

We thank our research students Ross McKenzie and Guangning (Gary) Tan for useful comments on this work, in particular for Ross’s PhD work which has led to new insights into dummy derivatives.

## References


On efficient Hessian computation using the edge pushing algorithm in Julia

Feng Qiang*, Cosmin G. Petra†, Miles Lubin‡, Joey Huchette‡, and Mihai Anitescu*

April 8, 2016

1 Introduction

We focus on the problem of efficient automatic computation of sparse Hessian matrices. The edge pushing (EP) algorithm was introduced in [1] as an alternative to the popular coloring-based compression algorithms [2]. Recent work by [3] has refined the algorithm and corrected issues with its initial implementation. However, the EP algorithm is not yet available in any off-the-shelf automatic differentiation (AD) packages, to our knowledge.

Here we report our experiences implementing the EP algorithm within JuMP [4], an algebraic modeling language for optimization. JuMP currently implements the coloring-based algorithms for Hessian computations and in recent benchmarks was found to perform with a factor of 2.2 of AMPL, a commercial software package with similar functionality to JuMP. Our motivation in implementing the EP algorithm is to potentially further close this performance gap.

Note that, in the context of optimization, we are primarily interested in the case of repeated Hessian evaluations using the same sparsity pattern. On the other hand, many of the reported gains in the EP algorithm over coloring approaches are due to the expense of the coloring step which can be amortized over many Hessian evaluations. As Wang et al. [3] state:

For repeated Hessian computation when the sparsity pattern remains the same, the compression-based approach should be preferred over LIVARH or LIVARHCC.

where LIVARH and LIVARHCC are two variants of the EP algorithm. We would like to identify cases where the EP algorithm is superior even after amortizing the cost of the coloring step.

Our initial implementation following [3] had notable performance issues due to the use of hash tables throughout the algorithm. We will describe an improved data structure we designed to implement the EP algorithm using fast $O(1)$ lookups with fewer memory indirections. This data structure yielded a 3x improvement in performance over our initial implementation, and has revealed some examples where the EP algorithm is indeed superior or competitive with coloring approaches. While some of the slowness of hash tables may be attributed to their current implementation in the Julia language, we believe that our new data structures could yield improvements in implementations in other languages as well.

2 Edge pushing algorithm

Consider a scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that can be representable as a finite sequence of elementary functions $\phi_i$, $i = \{1, 2, \ldots, l\}$, arithmetic operations or elementary mathematical functions. We follow the notation of Griewank and Walther [5] and denote the independent variables by $v_{1-n}, v_{2-n}, \ldots, v_0$ and the dependent variables corresponding to the elementary functions $\phi_i$ by $v_i$. Also we adopt the precedence notation $v_j \prec v_i$ to describe that $v_j$ is an operand or argument of $\phi_i$. The precedence relation between variables allows us to describe the evaluation of the function $f$ by means of a directed acyclic graph, known as computational graph (CG), whose nodes represent the variables and the vertices represent the precedence relations [3].

The edge pushing (EP) algorithm introduced in [1] and refined in [3] is an alternative to the popular coloring algorithms [2] for efficient computation of sparse Hessians. Generally coloring algorithms evaluate the Hessian $H$ by computing the directional derivatives in the form of Hessian-vector products $\nabla^2 f \cdot d$ and coloring algorithms such as [2] are used to reduce the number of columns $d$. The coloring problem is an NP-hard problem and is usually solved by using specialized heuristics such as [6]. The coloring approach has two potential drawbacks: 1) the cost of the coloring heuristic can be unreasonably high and/or 2) the number of colors needed, in either the heuristic solution or the true

---

*Mathematics and Computer Science Division, Argonne National Laboratory, \{fqiang,petra,anitescu\}@mcs.anl.gov
†Corresponding author
‡Operations Research Center, Massachusetts Institute of Technology, \{mlubin,huchette\}@mit.edu
minimal solution, may be excessively large, leading to a large number of Hessian-vector products. In Section 3 we show two classes of optimization problems for which 1) and 2) apply.

The edge pushing algorithm takes a different approach. It applies the chain rule at each node in the computational graph and uses a Hessian graph model that augments the computational graph with new edges that 1) track dependences of the gradients (adjoint variables) on the variables, or 2) carry second-order partial derivatives between the variables with nonlinear relationships. This Hessian graph model can be simplified to remove symmetric redundancies and derive the componentwise, recursive expression for the Hessian entries $H(k,j)$ [3]:

$$\forall i \in \{1,\ldots,1\}, \forall (k,j) \in S_i \times S_l, H_i(k,j) = H_{i+1}(k,j) + \frac{\partial \phi_i}{\partial v_{j}} H_{i+1}(i,k) + \frac{\partial \phi_{i}}{\partial v_{k}} H_{i+1}(i,j) + \frac{\partial^{2} \phi_{i}}{\partial v_{j} \partial v_{k}} H_{i+1}(i,i) + \hat{v}_{i} \frac{\partial^{2} \phi_{i}}{\partial v_{j} \partial v_{k}}.$$

(1)

Here we denote the intermediate Hessians by $H_i \in \mathbb{R}^{(n+l) \times (n+l)}$. The final Hessian $H \in \mathbb{R}^{(n+l) \times (n+l)}$, where $H(k,j)$ is the partial second-order derivative with respect to $v_j$ and $v_k$, is given by $H_0$. The adjoint variable $\hat{v}_i$ is computed during the reverse sweep computation of the gradient. The recursive expression makes use of the so-called live variable sets $S_i$ ($i = \{1,\ldots,1\}$), which tracks the nonlinear relationships between the variables and are defined recursively by

$$S_i = \{S_{i+1} \setminus \{i\}\} \cup \{j|v_j < \hat{v}_i\}.$$

(2)

2.1 A modification of the Hessian edge-pushing algorithm

Numerical EP algorithms such as [3] require frequent random access to (or lookup for) the entries of intermediate Hessians $H_i$ as required by the recursion (1). Since we target very large (but sparse) problems, the Hessian matrices need to be stored in a data structure with low space complexity that takes advantage of the sparsity. Reducing space usually competes with the time complexity of random access, even though it does not necessarily have to do so. In this work we propose a data structure for storing all the terms that contribute to $H_i(k,j)$. Mathematically they can be defined as

$$\forall i = l,\ldots,1, \quad H^{P}_{i}(k,j) = \emptyset \quad \text{and} \quad \forall (k,j) \in S_i \times S_l,$$

$$H^{P}_{i}(k,j) = \left\{ \begin{array}{ll}
H^{P}_{i-1}(k,j) \cup \left\{ \frac{\partial \phi_{i}}{\partial v_{j}} : h \in H^{P}_{i+1}(i,k) \right\}, & v_j < \hat{v}_i, v_k \neq \hat{v}_i, \\
H^{P}_{i-1}(k,j) \cup \left\{ \frac{\partial \phi_{i}}{\partial v_{k}} : h \in H^{P}_{i+1}(i,j) \right\}, & v_j \neq \hat{v}_i, v_k < \hat{v}_i, \\
H^{P}_{i-1}(k,j) \cup \left\{ \frac{\partial \phi_{i}}{\partial v_{j}} \frac{\partial \phi_{i}}{\partial v_{k}} : h \in H^{P}_{i+1}(i,i) \right\} \cup \left\{ \hat{v}_{i} \cdot \frac{\partial^{2} \phi_{i}}{\partial v_{j} \partial v_{k}} : \hat{v}_{i} \neq 0 \right\}, & v_j < \hat{v}_i, v_k < \hat{v}_i.
\end{array} \right.$$

(3)

Our modification of the EP algorithm updates the set-valued entries $H^{P}_{i}(k,j)$ accordingly to the recursion (3) and stores all the terms that contribute to $H_i(k,j)$. One can easily see that $H_i(k,j) = \sum_{h \in H^{P}_{i}(k,j)} h$, and, consequently, the entry $(k,j)$ in the Hessian will be $H(k,j) = \sum_{h \in H^{P}_{e}(k,j)} h$.

2.2 Implementation of the modification

The modification described above allows us to store the intermediary Hessians using CSC storage with duplicate entries. More specifically, we use a vector of $n + l$ vectors, where the $i$th inner vector contains pairs (row, value) corresponding to entries on column $i$, possibly with multiple pairs for the same entry as required by the use of set-valued entries in (3). Iterating over the product of live sets ($S_i$) in (3) is done in one iteration over the inner vector corresponding to column $i$, checking the precedence relations and updating as required by (3). The update for entry $(k,j)$ is done by appending the corresponding (j, value) pair to the $k$th inner vector and has constant time due to the use of vectors. Since it appends at the end of the vector, this approach does not need random access to the entries of the matrix.

At the end of the algorithm, we sum up the entries corresponding to the independent variables to obtain the Hessian. The complexity of this final step is linear in the number of entries corresponding to the independent variables when the accumulation is done in a separate buffer, usually provided by the optimization solver.

Because the entries of the Hessian can potentially be replicated, our proposed data structure has increased space complexity over other approaches, for example, vectors of dictionaries (one dictionary for each row). We have tried out this approach, which has has $O(1)$ time complexity for random access, and discovered that performs worse than the approach we presented above (about three times). As another alternative approach we mention the C++ implementation of the EP algorithm from [3] that is using Standard Template Library map container. Given the logarithmic time complexity for lookups of this container, we believe that, at least asymptotically, our implementation is more efficient.
3 Preliminary numerical experiments and results

We have implemented the Edge Pushing algorithm using our proposed data structure in Julia. Here, we compare the runtime results of our EP implementation with the Hessian coloring algorithm that is currently implemented in JuMP [4]. We express different algebraic functions using JuMP that demonstrate different sparsity structures in their Hessian. For both algorithms, we measure the times in seconds (s) for Hessian evaluations. They are listed under “Hessian” columns in the tables. In order to give a full description of the coloring algorithm, we list the time spent in coloring the adjacency graph (“coloring” column) and the number of colors.

We have chosen two sparsity patterns on which we compare the two algorithms.

3.1 Arrowhead structure

Coloring algorithms are known to perform poorly for Hessians exhibiting arrowhead structure [3]. Our testing function for the Hessian with arrowhead sparsity structure is

\[ \sum_{i=1}^{N} \left[ \cos \left( \sum_{j=1}^{K} x_{i+j} \right) + \sum_{j=1}^{K} (x_i + x_j)^2 \right], \]  

where \( K \) is the width of the border and the number of columns/rows in the Hessian is equal to \( N + K \).

Table 1 presents the performance results for \( K = 8 \) and increased \( N \). We can observe that both Coloring and EP algorithms demonstrate a linear time complexity with \( N \), however the coloring execution times dominate for large \( N \).

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>dim</th>
<th>nnz</th>
<th>Coloring</th>
<th>Edge Pushing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hessian coloring number of colors</td>
<td>Hessian triples</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>2008</td>
<td>31944</td>
<td>0.03 0.39 16</td>
<td>0.07 120000</td>
</tr>
<tr>
<td>4000</td>
<td>8</td>
<td>4008</td>
<td>63944</td>
<td>0.07 1.47 16</td>
<td>0.13 240000</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>8008</td>
<td>127944</td>
<td>0.14 5.93 16</td>
<td>0.26 480000</td>
</tr>
<tr>
<td>16000</td>
<td>8</td>
<td>16008</td>
<td>255944</td>
<td>0.25 25.36 16</td>
<td>0.51 960000</td>
</tr>
<tr>
<td>32000</td>
<td>8</td>
<td>32008</td>
<td>511944</td>
<td>0.49 141.20 16</td>
<td>1.08 1920000</td>
</tr>
<tr>
<td>64000</td>
<td>8</td>
<td>64008</td>
<td>1023944</td>
<td>0.97 689.42 16</td>
<td>2.10 3840000</td>
</tr>
</tbody>
</table>

Table 1: The performance results for function given in (4), when fixing \( K = 8 \)

In Table 2 we fix \( N = 20000 \) and double \( K \) and observe that the coloring is substantially expensive as \( K \) increases. Also we point out that it would require more than 100 Hessian evaluations based on coloring to offset this high cost, which is a high iteration number for most of the optimization problems.

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>dim</th>
<th>nnz</th>
<th>Coloring</th>
<th>Edge Pushing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hessian coloring number of colors</td>
<td>Hessian triples</td>
</tr>
<tr>
<td>20000</td>
<td>2</td>
<td>20002</td>
<td>79998</td>
<td>0.02 16.03 4</td>
<td>0.13 180000</td>
</tr>
<tr>
<td>20000</td>
<td>4</td>
<td>20004</td>
<td>159988</td>
<td>0.08 23.81 8</td>
<td>0.28 440000</td>
</tr>
<tr>
<td>20000</td>
<td>8</td>
<td>20008</td>
<td>319944</td>
<td>0.30 43.78 16</td>
<td>0.65 1200000</td>
</tr>
<tr>
<td>20000</td>
<td>16</td>
<td>20016</td>
<td>639760</td>
<td>1.14 93.97 32</td>
<td>1.68 3680000</td>
</tr>
<tr>
<td>20000</td>
<td>32</td>
<td>20032</td>
<td>1279008</td>
<td>4.10 179.81 64</td>
<td>5.72 12480000</td>
</tr>
</tbody>
</table>

Table 2: The performance results for function given in (4), when fixing \( N = 20000 \)

3.2 Random sparsity structure

The second class of test problems consist of Hessians with random sparsity structure. In this case, our testing function is presented in (5), where \( N \) equals to the dimension of the Hessian matrix, and \( K \) controls the density of the Hessian, more exactly the number of nonzero elements in each row/column. In (5) \( \text{rand}_\text{set}_i(N,K) \) denotes a set of length \( K \) of randomly generated and unique indexes from \( \{1, \ldots, N\} \). The dependence on \( i \) indicates that one such set is generated for each \( i \in \{1,2,\ldots,N\} \).

\[ \sum_{i=1}^{N} \left[ (x_i - 1)^2 + \prod_{j \in \text{rand}_\text{set}_i(N,K)} x_j \right] \]  

(5)

We have conducted two experiments similar to the experiments with arrowhead sparsity structure. First we fixed \( N = 4000 \) and varied \( K \) in Table 3 and then fixed \( K = 32 \) and varied \( N \) in Table 4. Surprisingly, for both experiments the EP algorithm showed much lower execution times than the coloring algorithm, in part because of the high cost of coloring and in part because of the poor quality of coloring approximation (a large number of colors), which results in many Hessian-vector computations and increased execution time for the coloring-based Hessian computation.
Table 3: The performance results for function given in (5), when fixing $N = 4000$

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>dim</th>
<th>nnz</th>
<th>Coloring Hessian coloring number of colors</th>
<th>Edge Pushing Hessian triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>2</td>
<td>4000</td>
<td>7999</td>
<td>0.00 0.02 4</td>
<td>0.01 8000</td>
</tr>
<tr>
<td>4000</td>
<td>4</td>
<td>4000</td>
<td>27946</td>
<td>0.03 0.05 12</td>
<td>0.03 28000</td>
</tr>
<tr>
<td>4000</td>
<td>8</td>
<td>4000</td>
<td>114984</td>
<td>0.09 0.22 49</td>
<td>0.09 116000</td>
</tr>
<tr>
<td>4000</td>
<td>16</td>
<td>4000</td>
<td>468239</td>
<td>1.05 2.57 283</td>
<td>0.27 484000</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>4000</td>
<td>1748609</td>
<td>17.97 38.17 1878</td>
<td>1.01 1988000</td>
</tr>
<tr>
<td>4000</td>
<td>64</td>
<td>4000</td>
<td>5037805</td>
<td>119.83 254.29 3884</td>
<td>4.36 8068000</td>
</tr>
</tbody>
</table>

Table 4: The performance results for function given in (5), when fixing $K = 32$

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>dim</th>
<th>nnz</th>
<th>Coloring hessian coloring number of colors</th>
<th>Edge Pushing Hessian triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>32</td>
<td>500</td>
<td>248500</td>
<td>0.65 0.89 480</td>
<td>0.11 106091</td>
</tr>
<tr>
<td>1000</td>
<td>32</td>
<td>1000</td>
<td>497000</td>
<td>2.32 4.06 901</td>
<td>0.21 309819</td>
</tr>
<tr>
<td>2000</td>
<td>32</td>
<td>2000</td>
<td>994000</td>
<td>7.18 13.56 1484</td>
<td>0.44 774686</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>4000</td>
<td>1988000</td>
<td>18.05 41.08 1884</td>
<td>0.98 1749603</td>
</tr>
<tr>
<td>8000</td>
<td>32</td>
<td>8000</td>
<td>3976000</td>
<td>38.77 94.47 1902</td>
<td>1.93 3725472</td>
</tr>
<tr>
<td>16000</td>
<td>32</td>
<td>16000</td>
<td>7952000</td>
<td>66.13 193.54 1688</td>
<td>4.11 7697333</td>
</tr>
</tbody>
</table>

4 Conclusions and future work

We proposed a modification of the edge pushing algorithm [3] to allow using a specialized data structure for reduced time complexity. Preliminary numerical results show significant benefits of using the proposed approach for Hessians with arrowhead and random structures.

Future work will be dedicated to further developments of the algorithm and integrating the implementation in JuMP. We will also study heuristics for deciding whether the coloring or edge pushing algorithms are appropriate for a given problem. Special attention will be given to performing comprehensive benchmarks using existing testbeds and comparisons with state-of-the-art algebraic modeling languages.

References

Forward-Mode Automatic Differentiation in Julia∗

Jarrett Revels† Miles Lubin‡ and Theodore Papamarkou†

April 2016

1 Introduction

We present ForwardDiff, a Julia package for forward-mode automatic differentiation (AD) featuring performance competitive with low-level languages like C++. Unlike recently developed AD tools in other popular high-level languages such as Python and MATLAB [1,2,3], ForwardDiff takes advantage of just-in-time (JIT) compilation [4] to transparently recompile AD-unaware user code, enabling efficient support for higher-order differentiation and differentiation using custom number types (including complex numbers). For gradient and Jacobian calculations, ForwardDiff provides a variant of vector-forward mode that avoids expensive heap allocation and makes better use of memory bandwidth than traditional vector mode.

In our numerical experiments, we demonstrate that for nontrivially large dimensions, ForwardDiff’s gradient computations can be faster than a reverse-mode implementation from the Python-based autograd package. We also illustrate how ForwardDiff is used effectively within JuMP [5], a modeling language for optimization. According to our usage statistics, 41 unique repositories on GitHub depend on ForwardDiff, with users from diverse fields such as astronomy, optimization, finite element analysis, and statistics.

2 Methodology

ForwardDiff implements a Julia representation of a multidimensional dual number, whose behavior on scalar functions is defined as:

\[
f(x + \sum_{i=1}^{k} y_i \epsilon_i) = f(x) + f'(x) \sum_{i=1}^{k} y_i \epsilon_i, \tag{1}
\]

where \( \epsilon_i \epsilon_j = 0 \) for all indices \( i \) and \( j \). Storing additional \( \epsilon \) components allows for a vector forward-mode implementation of the sort developed by Kahn and Barton [6]. In our formulation, orthogonal \( \epsilon \) components are appended to input vector components to track their individual directional derivatives:

\[
\bar{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_k \end{bmatrix} \rightarrow \bar{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 + 0 \sum_{n=2}^{k} \epsilon_n \\ \vdots \\ x_i + \epsilon_i + 0 \sum_{n\neq i} \epsilon_n \\ \vdots \\ x_k + \epsilon_k + 0 \sum_{n=1}^{k-1} \epsilon_n \end{bmatrix} \rightarrow f(\bar{x}_\epsilon) = f(\bar{x}) + \sum_{i=1}^{k} \frac{\partial f(\bar{x})}{\partial x_i} \epsilon_i \tag{2}
\]

Vector forward mode enables the calculation of entire gradients in a single pass of the program defining \( f \), but at the cost of additional memory and operations. Specifically, every dual number must allocate an \( \epsilon \) vector of equal size to the input vector, and the number of operations required for derivative propagation scales linearly with the input dimension. In practice, especially in memory-managed languages like Julia, the cost of rapidly allocating and deallocating large \( \epsilon \) vectors on the heap can lead to slowdowns that practically outweigh the advantage of fewer passes through \( f \).

ForwardDiff’s implementation works around this pitfall by stack-allocating the \( \epsilon \) vectors, as well as permitting their size to be tunable at runtime relative to the input dimension and performance characteristics of the target function. We call ForwardDiff’s strategy chunk mode, since it allows us to compute the gradient in bigger or smaller chunks of the input vector. The \( \epsilon \) vector length is then the chunk size of the computation. For a chunk size \( N \) and an input

∗We thank Kristoffer Carlsson for significant contributions to the library and Isaac Virshup for compiling usage statistics
†Massachusetts Institute of Technology, jrevels@csail.mit.edu, mlubin@mit.edu
‡University of Glasgow, School of Mathematics and Statistics, Theodore.Papamarkou@glasgow.ac.uk
vector of length $k$, it takes $\lceil \frac{N}{k} \rceil$ passes through $f$ to compute $\nabla f(\vec{x})$. For example, it takes two passes through $f$ to evaluate the gradient at a vector of length $k = 4$ and chunk size $N = 2$:

$$\vec{x}_{e_{1.2}} = \begin{bmatrix} x_1 + \epsilon_1 \\ x_2 + \epsilon_2 \\ x_3 \\ x_4 \end{bmatrix} \rightarrow f(\vec{x}_{e_{1.2}}) = f(\vec{x}) + \frac{\partial f(\vec{x})}{\partial x_1} \epsilon_1 + \frac{\partial f(\vec{x})}{\partial x_2} \epsilon_2$$

(3)

$$\vec{x}_{e_{3.4}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 + \epsilon_1 \\ x_4 + \epsilon_2 \end{bmatrix} \rightarrow f(\vec{x}_{e_{3.4}}) = f(\vec{x}) + \frac{\partial f(\vec{x})}{\partial x_3} \epsilon_1 + \frac{\partial f(\vec{x})}{\partial x_4} \epsilon_2$$

ForwardDiff implements a multidimensional dual number as the type Dual\{N,T\}, where the type parameter N denotes the length of the $\epsilon$ vector and the type parameter T denotes the element type (e.g. Dual\{2,Float64\} has two Float64 $\epsilon$ components). This type has two fields: value, which stores the $x$ component, and partials, which stores the stack-allocated $\epsilon$ vector. It’s straightforward to overload base Julia methods on the Dual type; here’s an example using sin, cos, and - (univariate negation):

```julia
import Base: sin, cos, -

sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)

cos(d::Dual) = Dual(cos(d.value), -(sin(d.value)) * d.partials)

(-)(d::Dual) = Dual(-(d.value), -(d.partials))
```

These method definitions are all that is required to support the following features:

- $n^{th}$-order derivative of sin or cos (through nesting Dual types)
- derivative of complex sin or cos via types of the form Complex\{Dual\{N,T\}\}
- derivative of sin or cos over custom types, e.g. Custom\{Dual\{N,T\}\} or Dual\{N,Custom\}

We unfortunately do not have room in this abstract to adequately cover the latter two items; a proper discussion would require a more thorough exposition of Julia’s multiple dispatch and JIT-compilation facilities.

Instead, we discuss how instances of the Dual type can be nested to enable the use of vector-mode AD for higher-order derivatives. For example, the type Dual\{M,Dual\{N,T\}\} can be used to compute $M \times N 2^{nd}$-order derivatives. As a simple demonstration of the scalar case, we use an instance of the type Dual\{1,Dual\{1,Float64\}\} to take the second derivative of sin at the Julia prompt (The notation $\epsilon$\{d,k\} is used to denote the $k^{th}$ partial nested at level d):

```julia
julia> d = Dual(Dual(1.0 , 1.0), Dual(1.0 , 0.0))
((1.0 + 1.0*\epsilon[1,1]) + (1.0 + 0.0*\epsilon[1,1])\epsilon[2,1])

julia> d2 = sin(d)
((0.84147 + 0.54030*\epsilon[1,1]) + (0.54030 - 0.84147*\epsilon[1,1])\epsilon[2,1])

julia> partials(partials(d2, 1), 1)
-0.8414709898078965
```

Algebraically, the above example is equivalent to the use of hyper-dual numbers described by Fike and Alonso [7]. In fact, a Dual instance with $d$ levels of nesting implements a $d^{th}$-order hyper-dual number, with the added advantage of scaling to arbitrary dimensions. For example, an instance of Dual\{M,Dual\{N,Dual\{L,T\}\}\} can be used to take $M \times N \times L$ third-order derivatives in a single pass of the target function.

## 3 Performance Analysis

In this section, we present timing results for gradient calculations of the Rosenbrock (4) and Ackley (5) functions. Recalling [1] and the discussion in Section 2, increasing chunk size reduces the number of evaluations of the univariate functions within $f$. We choose Ackley and Rosenbrock as our target functions in order to provide a contrast between the relative gains of increasing chunk sizes when the target function contains many and few expensive univariate functions, respectively.

$$\text{Rosenbrock}(\vec{x}) = \sum_{i=1}^{k-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

(4)
\[ \text{Ackley}(\vec{x}) = -a \exp \left( -b \sqrt{ \frac{1}{k} \sum_{i=1}^{k} x_i^2 } \right) - \exp \left( \frac{1}{k} \sum_{i=1}^{k} \cos(cx_i) \right) + a + \exp(1) \] (5)

Table 1 shows evaluation times for calculating gradients of our two target functions using ForwardDiff versus a naive equivalent C++ implementation. Various chunk sizes were tested, while the input size was fixed at 12000 elements. For the sake of simplicity, ForwardDiff’s \( \text{Dual} \{N,T\} \) type was translated into a hardcoded C++ class for each \( N \).

<table>
<thead>
<tr>
<th>chunk size ( N )</th>
<th>C++ Time (s)</th>
<th>ForwardDiff Time (s)</th>
<th>chunk size ( N )</th>
<th>C++ Time (s)</th>
<th>ForwardDiff Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.66744</td>
<td>0.62760</td>
<td>1</td>
<td>4.02078</td>
<td>5.12890</td>
</tr>
<tr>
<td>2</td>
<td>2.71184</td>
<td>0.45541</td>
<td>2</td>
<td>4.35398</td>
<td>2.72033</td>
</tr>
<tr>
<td>3</td>
<td>1.92713</td>
<td>0.44699</td>
<td>3</td>
<td>3.05532</td>
<td>1.86055</td>
</tr>
<tr>
<td>4</td>
<td>1.45306</td>
<td>0.42354</td>
<td>4</td>
<td>2.26095</td>
<td>1.47578</td>
</tr>
<tr>
<td>5</td>
<td>1.24949</td>
<td>0.44045</td>
<td>5</td>
<td>1.91985</td>
<td>1.23500</td>
</tr>
</tbody>
</table>

(a) \( \nabla \text{(Rosenbrock)} \)

(b) \( \nabla \text{(Ackley)} \)

Table 1: Time to evaluate gradients using C++ vs. ForwardDiff, input size \( k = 12000 \)

Table 2 compares the gradient computation time of the reverse-mode implementation of the Python-based autograd package versus the forward-mode implementation of ForwardDiff for varying input sizes. We also include results obtained using our experimental multithreaded implementation, which show a \(~2x\) speed-up using 4 threads compared to our single-threaded implementation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Size ( k )</th>
<th>autograd Time (s)</th>
<th>ForwardDiff Time (s)</th>
<th>ForwardDiff Multithreaded Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackley</td>
<td>10</td>
<td>0.001204</td>
<td>0.000007</td>
<td>0.000007</td>
</tr>
<tr>
<td>Ackley</td>
<td>100</td>
<td>0.008472</td>
<td>0.000058</td>
<td>0.000056</td>
</tr>
<tr>
<td>Ackley</td>
<td>1000</td>
<td>0.081499</td>
<td>0.006351</td>
<td>0.002620</td>
</tr>
<tr>
<td>Ackley</td>
<td>10000</td>
<td>0.835441</td>
<td>0.564828</td>
<td>0.253798</td>
</tr>
<tr>
<td>Ackley</td>
<td>100000</td>
<td>8.361769</td>
<td>56.850198</td>
<td>24.394373</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>10</td>
<td>0.000866</td>
<td>0.000003</td>
<td>0.000003</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>100</td>
<td>0.004395</td>
<td>0.000034</td>
<td>0.000041</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>1000</td>
<td>0.040702</td>
<td>0.003010</td>
<td>0.001582</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>10000</td>
<td>0.411095</td>
<td>0.302277</td>
<td>0.159703</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>100000</td>
<td>4.173851</td>
<td>30.365882</td>
<td>14.11776</td>
</tr>
</tbody>
</table>

Table 2: Time to evaluate gradients using autograd (reverse mode) vs. ForwardDiff, chunk size \( N = 10 \)

Table 2 compares the gradient computation time of the reverse-mode implementation of the Python-based autograd package versus the forward-mode implementation of ForwardDiff for varying input sizes. We also include results obtained using our experimental multithreaded implementation, which show a \(~2x\) speed-up using 4 threads compared to our single-threaded implementation.

Both functions have linear complexity in the input dimension \( k \); therefore reverse mode, which requires \( O(1) \) passes through each function, scales linearly, while our forward mode, which requires \( O(k) \) passes through each function (with fixed \( N \)), scales quadratically. The results in Table 2 agree with this complexity analysis. Nevertheless, there is a huge performance gap between these two implementations such that autograd is slower on these examples when \( k \leq 10000 \), despite reverse mode being a superior algorithm in principle for computing gradients.

The code used to generate the timings in this section can be found at https://github.com/JuliaDiff/ForwardDiff.jl/tree/jr/benchmarks/benchmark. Julia benchmarks were run using Julia version 0.5.0-dev+3200, C++ benchmarks were compiled with clang-600.0.57 using -O2, and Python benchmarks were run using Python version 2.7.9.

## 4 ForwardDiff within JuMP

Effective use of ForwardDiff has brought improvements to JuMP [5], a domain-specific language for optimization embedded in Julia where users provide closed-form algebraic expressions using a specialized syntax. JuMP, and similar commercial tools like AMPL [8], compute derivatives of user models as input to nonlinear optimization solvers, which is quite different from ForwardDiff’s original target case of differentiating general user-defined code.

JuMP computes sparse Hessians by using the graph coloring approach of [9], which requires computing a small number of Hessian-vector products in order to recover the full Hessian. JuMP’s forward-over-reverse mode implementation
for Hessian-vector products makes use of ForwardDiff’s chunk mode, essentially computing Hessian-matrix products instead of independent Hessian-vector products. This use of chunk mode yielded speedups of 30% on benchmarks presented in [5] (under review). The results in [5] include this speedup but are not accompanied by a discussion of the methodology of chunk mode.

On the user-facing side, ForwardDiff has enabled JuMP to be the first AML to our knowledge which performs automatic differentiation of user-defined functions embedded within closed-form expressions. We reproduce an example from [5] illustrating a user-defined square root function within a JuMP optimization model:

```julia
function squareroot(x)
    # Start Newton’s method at x
    z = x
    while abs(z*z - x) > 1e-13
        z = z - (z*z-x)/(2z)
    end
    return z
end
```

```julia
registerNLFunction(:squareroot, 1, squareroot, autodiff=true)
m = Model()
@defVar(m, x [1:2], start=0.5)
@setObjective(m, Max, sum(x))
@addNLConstraint(m, squareroot(x[1]^2+x[2]^2) <= 1)
solve(m)
```

JuMP computes gradients of `squareroot` with ForwardDiff which are then integrated within the reverse-mode computations of JuMP. We do not yet support 2nd-order derivatives of user-defined functions. While this functionality is immature and leaves room for improvement (we could attempt to tape the user-defined functions and calculate their gradients in reverse mode), it already creates a new and useful way for JuMP users to seamlessly interact with AD when small parts of their model cannot easily be expressed in closed algebraic form.

5 Future Work

We are currently investigating several avenues of research that could improve ForwardDiff’s performance and usability. We are in the preliminary phases of implementing SIMD vectorization of derivative propagation. We intend to address perturbation confusion [10] by intercepting unwanted perturbations at compile time using Julia’s metaprogramming capabilities. Finally, we wish to improve our support for matrix operations such as eigenvalue computations by directly overloading linear algebra functions, a technique which has already seen use in [2].

References

Bidirectional Partitioning is no Better than Unidirectional Partitioning when Computing the Diagonal Entries of a Sparse Jacobian Matrix

H. Martin Bücker†, Michael Lülfesmann‡, M. Ali Rostami‡ and Andreas Wolf§
March 2016

1 Compression Techniques in Derivative Computations

The formulation and solution of problems involving sparse matrices is often based on graph theoretical considerations. Gebremedhin, Manne and Pothen [1] thoroughly review the connections between sparse matrices and graphs for the area of computing derivatives. Sparse Jacobian or Hessian matrices are ubiquitous in various areas of computational engineering and data science including nonlinear optimization, differential equations, and sensitivity analysis. Curtis, Powell and Reid [2] introduced the concept of “structurally orthogonal columns” to minimize the computational effort spent in computing all nonzero entries of a sparse Jacobian matrix. The work on graph models for these types of problems started with Coleman and Moré [3] in the early 1980s. In this pioneering work, a partition of the columns of a sparse Jacobian into a small number of groups of structurally orthogonal columns is modeled by a distance-1 graph coloring problem. Since then, various graph coloring problems were formulated and carefully investigated in an attempt to reduce computational effort [4, 5, 6, 7]. The common idea behind these different compression techniques is to exploit an a priori known sparsity pattern as follows. First, a compressed representation of the derivative matrix is obtained by computing linear combinations of columns (or rows) at a time, rather than computing each column (or row) individually. Then, the nonzero entries are retrieved from this compressed representation. The computational effort needed for the first step of a compression technique is characterized by the number of different groups representing different linear combinations. Throughout this extended abstract, we identify the computational effort of a compression technique by the number of these groups. That is, we neglect the computational work to retrieve the nonzero entries from the compressed representation of the derivative matrix.

Compression techniques were originally developed for numerical differentiation where sparsity is exploited by forming suitably chosen linear combinations of the columns. This carries over to the forward mode of automatic differentiation. The reverse mode of automatic differentiation, in contrast, allows linear combinations of rows. The term unidirectional partitioning refers to forming linear combinations of either columns or rows. A mixture of linear combinations of columns and rows is called bidirectional partitioning. It is well known that bidirectional partitioning, which corresponds to a combination of the forward and the reverse mode, requires fewer linear combinations of columns and rows than unidirectional partitioning where each of these modes is applied separately.

Rather than computing all nonzero elements of a Jacobian matrix, the focus of this extended abstract is on computing a proper subset of these nonzero elements.

2 Graph Coloring for Partial Jacobian Computation

Throughout this extended abstract, we consider a non-singular $n \times n$ Jacobian matrix with a given non-symmetric sparsity pattern. The following bipartite graph model $G = (V_r, V_c, E)$ is associated to every Jacobian matrix. The sets of vertices $V_r$ and $V_c$ represent all rows and columns, respectively. There is an (undirected) edge $(r_i, c_j) \in E$ with $r_i \in V_r$ and $c_j \in V_c$, iff the Jacobian entry in position $(i,j)$ is nonzero.

Rather than computing all nonzero entries of the Jacobian matrix, we assume that a proper subset of nonzero entries is required. Computing such a set of required nonzero elements is called partial Jacobian computation as opposed to full Jacobian computation whose purpose is to compute all nonzero entries. Throughout this extended abstract, we are interested in computing the diagonal Jacobian entries without computing the off-diagonal nonzero entries. To this end, let

$$E_D := \{(r_i, c_i) \mid 1 \leq i \leq n\} \subseteq E$$

(1)

denote the set of required elements. We are now interested in unidirectional and bidirectional partitioning for partial Jacobian computation restricted to $E_D$. 

---

*Friedrich Schiller University Jena, martin.buecker@uni-jena.de
†Düsseldorf, Germany
‡Corresponding Author, Friedrich Schiller University Jena, a.rostami@uni-jena.de
§Technische Universität Darmstadt, Andreas.Wolf@hrz.tu-darmstadt.de
For unidirectional partitioning, minimizing the computational effort in terms of linear combinations of columns and rows translates to finding a distance-2 coloring of $G$ when restricted to $E_D$ with the smallest number of colors. This problem is called restricted distance-2 coloring problem. The minimal number of colors is denoted by $\chi_{d2}$ and is called chromatic number. For bidirectional partitioning, the corresponding minimization problem in terms of the bipartite graph is given as follows: Find a star bicoloring of $G$ when restricted to $E_D$ with the smallest number of colors. This problem is referred to as the restricted star bicoloring problem. The corresponding chromatic number is denoted by $\chi_{sb}$. More details on partial Jacobian computation is given in [1].

3 An Illustrating Example

We introduce a new motivating example showing that, in general, the chromatic number decreases not only when going from unidirectional to bidirectional partitioning, but also when going from full to partial Jacobian computation. To this end, we consider a bipartite graph $G = (V_r, V_c, E)$ associated to an $n \times n$ sparse Jacobian matrix consisting of $n$ vertices in $V_r$ to represent the rows and $n$ vertices in $V_c$ to represent the columns. Given a set of edges $E$ representing the nonzero entries of the Jacobian and a proper subset $E' \subset E$ representing the required nonzero entries we study four different coloring problems:

- Full Jacobian computation using a unidirectional partitioning. (Distance-2 coloring problem.)
- Partial Jacobian computation using a unidirectional partitioning. (Restricted distance-2 coloring problem.)
- Full Jacobian computation using a bidirectional partitioning. (Star bicoloring problem.)
- Partial Jacobian computation using a bidirectional partitioning. (Restricted star bicoloring problem.)

We construct the set of edges $E$ and its subset $E'$ in such a way that the chromatic numbers given in Table 1 are obtained. By taking a look at the chromatic numbers in that table we find the overall idea behind this example. The example is designed to illustrate the following: The chromatic numbers are reduced

- when going from full to partial Jacobian computation (no matter whether unidirectional or bidirectional partitioning is used), and
- when going from unidirectional to bidirectional partitioning (no matter whether full or partial Jacobian computation is considered).

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>Partial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unidirectional</td>
<td>$n$</td>
<td>$n/2 + 1$</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Chromatic numbers of the four different coloring problems.

4 Distance-2 and Star Bicoloring Restricted to Diagonal Entries

The example sketched in the previous section illustrates that, in general, we would expect a reduction in the chromatic number when going from full to partial Jacobian computation. However, in contrast to this intuition, we now demonstrate that there is no reduction in the chromatic number when going from full to partial Jacobian computation if we consider the restriction to the diagonal elements $E_D$ defined in (1). This is surprising since we do not assume any structure in the Jacobian pattern. That is, we consider a general, non-symmetric Jacobian pattern.

We sketch the outline of the proof of this result in this extended abstract and leave the details for the post-conference publication. The proof consists of a sequence of different statements. We first show that a distance-2 coloring of the rows of a graph $G$ when restricted to $E_D$ is equivalent to a corresponding distance-2 coloring of the columns. Thus, the computational work to determine the diagonal nonzero entries by unidirectional partitioning is independent from the choice of row or column partitioning. The second and third step is to show that

$$\chi_{d2} \leq \chi_{sb} \quad \text{and} \quad \chi_{sb} \leq \chi_{d2}$$

as long as the coloring problems are restricted to $E_D$. Thus, the computational work to determine the diagonal nonzero entries is independent from the choice of unidirectional or bidirectional partitioning.
5 Concluding Remarks

Intuition tells us that bidirectional partitioning requires fewer computational effort than unidirectional partitioning when computing a general subset of nonzero elements of a sparse Jacobian. We construct a new example that illustrates not only this statement, but also the reduction in computational effort when going from full to partial Jacobian computation. The main new contribution of this extended abstract is to show the following counter-intuitive result: Given a Jacobian with a general, non-symmetric sparsity pattern, bidirectional partitioning does not save linear combinations compared to unidirectional partitioning if one is interested solely in the diagonal nonzero elements.

We believe that this theoretical result is of high practical relevance. The reason is that the problem of determining the diagonal entries of a sparse Jacobian matrix without determining all remaining nonzero Jacobian entries could be an important building block in future preconditioning algorithms. Suppose that you want to construct a diagonal preconditioner for a system of linear equations whose non-singular coefficient matrix is a Jacobian. Then, the result of this extended abstract tells us that, for diagonal preconditioning based on automatic differentiation, there is no reduction in the computational effort when using the reverse mode. In fact, the forward mode which is easier to handle will compute the diagonal entries with the same computational effort.

References


Expression templates for primal value taping in the reverse mode of AD

Max Sagebaum, Tim Albring and Nicolas R. Gauger

Chair for Scientific Computing, Technische Universität Kaiserslautern

March 2016

0 Introduction

AD tools using operator overloading in C++ and having implemented the reverse mode can be divided into two major groups. The first group consists of the ones that store the local Jacobian of each expression during taping, we refer to this as the Jacobian approach. Examples are Sacado, AD4CL, Adept and CoDiPack. R. Hogan showed in [1] that this method can be efficiently implemented by means of expression templates [2]. The second group are the ones that store the primal input and output values of each expression. We call this method the primal value approach. FADBAD, CPPAD, ADOL-C and AutoDiff_Lib are some of the representatives in this group. The implementations for this method range from traditional tape structures (ADOL-C) to graph representations (AutoDiff_Lib), but to our knowledge there is no tool available that uses expression templates also for that case. Consequently there is currently no possibility for a fair comparison between both approaches. The aim of this work is to provide a unified AD framework based on expression templates that enables the comparison of different taping methods regarding run time and memory requirements. Furthermore we will highlight the strengths and weaknesses of either of the methods.

1 Expression templates

Assume that we have a computation type \( RReal \) that stores the primal value and a unique index. Usually C++ only allows the overloading of unary and binary operators. A regular reverse mode implementation that follows the layout \( RReal \circ RReal \rightarrow RReal \) would split the statement

\[
 w = ((a + b) \cdot (c - d))^2
\]

into

\[
 t_1 = a + b; \quad t_2 = c - d; \quad t_3 = t_1 \cdot t_2; \quad w = t_3^2.
\]

Hence we would need to store 3 binary operations and 1 unary operation. If the layout of the operator is changed to \( Expr_A \circ Expr_B \rightarrow Expr_{AB} \) where \( Expr \) is some class that stores information about the operations, then multiple operations can be chained together and create one large expression at the end. For (1) the generated structure for the whole expression would be

\[
 POW < MULT < ADD < RReal, RReal >, SUB < RReal, RReal >>> .
\]

Because of this structure no intermediate variables are created and therefore no assignments to temporary variables need to be stored. The interface of the expression class can now be designed in such a way that the primal value and the Jacobian of the expression can be evaluated. The AD tool can use this information to store one big expression instead of 4 operations.

The expression template technique is used to implement this recursion. The basic idea is to define a template class that gets the extending class as an argument. Figure 1 shows this in an example. The base class provides a cast method in line 3 to return a reference to the extending class. Every method in the base class uses this method to call the implementation of the extending class. In line 8 the \( SinExpr \) is implemented and \( Expression \) is used as a base class.

*Corresponding author max.sagebaum@scicomp.uni-kl.de
template<typename A>
class Expression {
    A & cast() { return static_cast<A&>(*this); }
    double value() { return this->cast().value(); }
};

template<typename A>
class SinExpr : public Expression<SinExpr<A>> {
    const A & a;
    SinExpr(const A & a) : a(a) {}
    double value() { return sin(a.value()); }
};

Figure 1: Expression template example.

double dw_da = derivativeA(a.value(), b.value(), this->value());
double dw_db = derivativeB(a.value(), b.value(), this->value());
a.calcJacobi(dw_da * multiplier);
b.calcJacobi(dw_db * multiplier);

Figure 2: Implementation for the Jacobi computation in an expression template.

2 Jacobian approach

Consider a statement
\[ y = f(x) \] (2)
with \( x \in \mathbb{R}^m, y \in \mathbb{R}^n \) and \( f : \mathbb{R}^m \rightarrow \mathbb{R}^n \). The corresponding AD reverse statement given in Griewank et al. [3] is
\[ \bar{x} + = \frac{\partial f}{\partial x}^T(x)\bar{y} . \] (3)

Since we only consider single statements in a computer code we have \( n = 1 \). In general \( m \) can be arbitrarily large, however here we assume that one statement has not more than 255 arguments.

In order to compute the local Jacobian \( \frac{\partial f}{\partial x}^T(x) \) for (3), the interface in figure 1 is extended by the function \( \text{calcJacobi} \) in figure 2. The implementation shown here is for a general binary operator \( w = a \circ b \). The first two lines in the body compute the Jacobian \( \frac{dw}{d(a,b)} \) and the 3-rd and 4-th lines perform the scalar product \( \frac{dw}{d(a,b)}^T \bar{y} \). The recursive nature of \( \text{calcJacobi} \) ensures that the full Jacobi of an arbitrary statement like (1) can be evaluated.

In order to evaluate (3) in the reverse sweep, the Jacobian approach needs to store the Jacobian of \( f \) and the indices of the arguments \( x \). The required memory for one statement is then \( m \times 8 \) bytes for the Jacobian, \( m \times 4 \) bytes for the indices of the arguments and 1 byte for the number of arguments. The indices of variables occurring on the left-hand side of a statement are distributed in an incremental fashion. Since we sequentially loop through all statements during the reverse sweep there is no need to store the index of \( y \). In total the Jacobian approach has to store \( 1 + 12 \times m \) bytes of information for each statement.

3 Primal value approach

The primal value approach stores only the indices of the input and output values of each statement during the forward sweep and computes the Jacobian \( \frac{df}{dx}(x) \) during the reverse sweep. Because of the incremental indexing scheme each value has a unique index and therefore the primal value can always be accessed by this index. The memory that is required by each statement is then \( 4 \times m \) bytes for the indices of the arguments \( x \), 8 bytes for the primal value of the output \( y \) and a 8 byte pointer to a function handle for the reverse evaluation logic of this statement.

The latter is a static structure that is generated for each expression. All the static information about the expression like the number of arguments can be stored in this structure. This has the advantage that each occurrence of the same expression will access the same static structure. Therefore only a 8 byte pointer needs to be stored and not the pointer and an instance of the structure. The implementation is shown in figure 3. It uses static member initialization with templates in order to achieve the described behavior. Therefore the memory for the static structure can be ignored.

The total memory required for one statement is then \( 16 + 4 \times m \) bytes. The direct comparison with the Jacobian approach therefore shows that the constant memory for each statement is 16 times higher but the memory for each argument is only one third.
Figure 3: Static member initialization for the information about the expression data.

For a first analysis the TRACE code [4] software, which is a 3D turbomachinery flow solver developed at German Aerospace Center (DLR), is taken. We measured $n_{state} = 7.3 \times 10^8$ statements and $n_{arg} = 2.7 \times 10^9$ arguments. The Jacobian approach would need $n_{state} \times 1$ byte + $n_{arg} \times 12$ byte = 30.9 GB and for the primal value approach this would result in $n_{state} \times 16$ byte + $n_{arg} \times 4$ byte = 20.9 GB. This example shows that the primal value approach would save 10 GB of memory which is a clear indication that it is necessary to further compare both approaches especially in the case of large-scale applications.

We also want to compare the expression template primal value approach with a traditional approach. The example (1) from the expression template introduction has 3 binary operations and one unary operator if they are stored separately, then 92 bytes are required. If expression templates are used then the statements has 4 arguments and needs only 32 bytes of memory. This is a clear indication that the introduction of expression templates can reduce the memory of traditional primal value approaches.

4 Results

The coupled burgers equations [5] are chosen in the following as a demonstration test case. It can be used to do a rapid evaluation on how some changes in CoDiPack [6] affect the performance of various tape implementations. Because of the simplicity all measures like the number of operations, statements etc. can be evaluated by hand or can be computed for various configurations of the AD tool.

The equations

\[ u_t + uu_x + vv_y = \frac{1}{R} (u_{xx} + u_{yy}) \]
\[ v_t + uu_x + vv_y = \frac{1}{R} (v_{xx} + v_{yy}) \]

are discretised with an upwind finite difference scheme. The initial conditions are:

\[ u(x, y, 0) = x + y \quad (x, y) \in D \]
\[ v(x, y, 0) = x - y \quad (x, y) \in D \]

and therefore the exact solution is after [5]

\[ u(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R} \]
\[ v(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R} \]

The computational domain $D$ is the unit square $D = [0, 1] \times [0, 1] \subset \mathbb{R} \times \mathbb{R}$ and the boundary conditions are taken from the exact solution. For the test runs a grid size of 601x601 grid points and 32 discrete time steps are chosen. The
computations are evaluated on the Elwetritsch cluster of the TU Kaiserslautern. The test case is evaluated on one node of the cluster which consists of two Intel E5-2640v3 processors. Two load cases are considered. For the first one only one process is run. For the second one the sequential program is run on 16 cores simultaneously. This simulates an environment where the memory bandwidth of the node is fully utilized.

Figure 4 shows the results for both test cases. It can be clearly seen that the evaluation speed is similar for both approaches and they do not differ by large factor e.g. 20. If the recording time and reverse evaluation time is added for the single processes case, then both approaches take nearly the same time. The reverse evaluation process takes longer for the primal value approach because the computation of the gradient is moved from the recording of the tape to the evaluation phase of the tape. For the case with the 16 processes the memory bandwidth slows both processes down, but because of the reduced memory the primal value approach is faster and the additional computations can be hidden in the memory latency.

The tape memory for this configuration is 4.8 GB for the Jacobian approach and 4.0 GB for the primal value approach.

5 Outlook

The first preliminary results presented in this extended abstract show that the extension of the primal value approach technique with expression templates can reduce the required amount of memory and improve the evaluation speed with respect to the Jacobian approach.

In addition to more elaborate measurements including large-scale applications we will also include a more detailed description of the primal value method and a full analysis of the minimal memory requirement for the statements and the reverse evaluation process. Furthermore we will discuss the handling of passive and constant values and the assumption of an incremental indexing scheme will be extended to a more complex non-incremental indexing method which enables the reuse of indices.

References


Developments of a Discrete Adjoint Structural Solver for Shape and Composite Material Optimization

Marc Schwalbach, Tom Verstraete and Nicolas R. Gauger

March 2016

1 Introduction

In the field of turbomachinery, a multidisciplinary optimization may seek to optimize an objective in the discipline of fluid dynamics, e.g. maximizing efficiency, while satisfying constraints in the discipline of structural mechanics, e.g. keeping the maximum von Mises stress beneath a defined threshold. This enables the design of components that are both aerodynamically optimized and structurally feasible. In order to respect structural constraints, gradient-based optimization methods require the sensitivities of the structural objectives with respect to the design variables.

A discrete adjoint structural solver, based on the finite element method (FEM), and differentiated using adjoint algorithmic differentiation (AD), plays a key role in computing the structural sensitivities efficiently. The gradients can be computed at a cost of a single adjoint run for each objective function, independent on the size of the design space. Not only does this give us the opportunity to include a large number of design parameters, e.g. a large number of control points for a CAD-based shape parametrization, but even to go beyond the usual shape design parameters and include composite material design parameters as well. This paves the way for an efficient simultaneous shape and material optimization using adjoint AD.

In this work, we introduce the ongoing developments of a discrete adjoint structural solver that has the capabilities of computing sensitivities with respect to shape design parameters, as well as composite material design parameters, with a single adjoint run for each structural objective function. The solver is written in C++ and differentiated using the AD tool CoDiPack [3]. The theory behind this concept is discussed first, followed by preliminary results and a conclusion.

2 Theory

Given an aerodynamic design objective of e.g. maximizing efficiency \( \eta \in \mathbb{R} \), the objective function \( J(x) \in \mathbb{R} \) can be defined as

\[
J(x) = -\eta.
\]

The constrained optimization problem is then formulated as

\[
\begin{align*}
\mathbf{x}^* &= \min J(x), \\
s.t. \quad &\sigma_{\text{max}}(x) \leq \sigma_{\text{yield}},
\end{align*}
\]

where \( \mathbf{x}^* \in \mathbb{R}^d \) defines the optimum, \( \mathbf{x} \in \mathbb{R}^d \) the design space, \( \sigma_{\text{max}}(x) \in \mathbb{R} \) the maximum von Mises stress, and \( \sigma_{\text{yield}} \in \mathbb{R} \) the critical von Mises stress. Gradient based optimization methods require the sensitivities with respect to the design space of the objective

\[
\frac{\partial J}{\partial \mathbf{x}^i} \in \mathbb{R}^{1 \times d}
\]

and the constraint

\[
\frac{\partial \sigma_{\text{max}}}{\partial \mathbf{x}^i} \in \mathbb{R}^{1 \times d}
\]

at each optimization step \( i \) to iterate towards the optimum \( \mathbf{x}^* \). A single design objective and a large design space is considered, such that \( 1 \ll d \), allowing a cheap computation of the gradients (4, 5) using adjoint AD. This work focuses on the computation of the structural sensitivities (5) using a discrete adjoint structural solver.

*Corresponding Author, von Karman Institute for Fluid Dynamics, marc.schwalbach@vki.ac.be
†Queen Mary University of London, t.verstraete@qmul.ac.uk
‡Technische Universität Kaiserslautern, nicolas.gauger@scicomp.uni-kl.de
2.1 Adjoint Model of Structural Solver

Consider the structural solver \( S \), which takes the design parameters \( x \) as an input and outputs the structural objective

\[
\sigma_{\text{max}} = S(x). \tag{6}
\]

The adjoint model is given by

\[
x_{(1)} = \frac{\partial \sigma_{\text{max}}}{\partial x} \cdot \sigma_{(1), \text{max}}, \tag{7}
\]

with adjoint variables \( x_{(1)} \in \mathbb{R}^d \), and \( \sigma_{(1), \text{max}} \in \mathbb{R} \). Following adjoint variables will be denoted with the subscript \( s_{(1)} \) as well. Seeding the adjoint model (7) with \( \sigma_{(1), \text{max}} = 1 \), allows the computation of the gradient (5) with a single adjoint run. The solver \( S \) involves solving a linear system, which is treated separately when differentiated with AD. This is done by splitting the solver \( S \) into three stages:

1. \( A, b = \text{Setup}(x) \)
2. \( u = \text{Solve}(A, b) \)
3. \( \sigma_{\text{max}} = \text{StressAnalysis}(u, x) \)

Here, \( A \in \mathbb{R}^{d \times d} \) defines the stiffness matrix, \( b \in \mathbb{R}^d \) the load vector, and \( u \in \mathbb{R}^d \) the displacements. The adjoint models of the three stages are given by

\[
\begin{pmatrix}
  u_{(1)} \\
  x_{(1)}
\end{pmatrix} = \begin{pmatrix}
  \frac{\partial \sigma_{\text{max}}}{\partial u} & \frac{\partial \sigma_{\text{max}}}{\partial x}
\end{pmatrix}^T \cdot \sigma_{(1), \text{max}}, \tag{8}
\]

\[
\begin{pmatrix}
  A_{(1)} \\
  b_{(1)}
\end{pmatrix} = \begin{pmatrix}
  \frac{\partial u}{\partial A} & \frac{\partial u}{\partial b}
\end{pmatrix}^T \cdot u_{(1)}, \tag{9}
\]

\[
x_{(1)} = x_{(1)} + \left( \frac{\partial A}{\partial x} \cdot b_{(1)} \right), \tag{10}
\]

The differentiation of (8) and (10) with reverse AD is straightforward. To avoid differentiating the linear solver, a derivation for an alternate computation of the adjoint variables in (9) is well known to be

\[
\begin{align*}
  b_{(1)} &= A_{(1)}^T u_{(1)} \\
  A_{(1),i,j} &= -u_j b_{(1),i}. \tag{11-12}
\end{align*}
\]

Due to the self-adjoint form of the stiffness matrix \( A \), (11) can be solved by solving the linear system

\[
A b_{(1)} = u_{(1)}, \tag{13}
\]

with the same stiffness matrix \( A \) used in the forward run. \( b_{(1)} \) and \( A_{(1)} \) can then be plugged into (10) to solve for \( x_{(1)} \).

2.2 Extension of Design Space

The design parameters \( x \) are usually composed of shape design parameters \( \alpha \in \mathbb{R}^s \), such as FEM node coordinates or CAD control points that dictate the shape of the geometry. With the inclusion of composite materials in the structural solver, the design space can be expanded to include material design parameters \( \beta \in \mathbb{R}^m \) that influence the material properties:

\[
x = \begin{pmatrix}
  \alpha \\
  \beta
\end{pmatrix} \in \mathbb{R}^d, \quad d = s + m \tag{14}
\]

The adjoint model described in section 2.1 can then be used to compute the sensitivities of the structural objective with respect to shape and material design parameters

\[
\frac{\partial \sigma_{\text{max}}}{\partial x} = \begin{pmatrix}
  \frac{\partial \sigma_{\text{max}}}{\partial \alpha} & \frac{\partial \sigma_{\text{max}}}{\partial \beta}
\end{pmatrix}^T \tag{15}
\]

without any additional adjoint runs. When not considering aeroelastic effects, the sensitivities of the aerodynamic objective are not dependent on the material design parameters:

\[
\frac{\partial J}{\partial x} = \begin{pmatrix}
  \frac{\partial J}{\partial \alpha} & 0
\end{pmatrix}^T \tag{16}
\]

This leads us to a two-tiered design space. The shape design parameters \( \alpha \) have a direct effect on both the aerodynamic objective and the structural constraint, while the material design parameters \( \beta \) only directly affect the structural...
constraint. However, the entire design space $x$ can be treated as one, such that the optimization method optimizes the shape and material at the same time.

\[
\begin{align*}
\alpha^i & \xrightarrow{CFD \text{ forward and reverse}} \frac{\partial J}{\partial \alpha^i} \\
\beta^i & \xrightarrow{FEM \text{ forward and reverse}} \frac{\partial \sigma_{\text{max}}}{\partial \beta^i}
\end{align*}
\rightarrow \text{optimizer} \rightarrow x^{i+1} = \left( \frac{\alpha^{i+1}}{\beta^{i+1}} \right)
\]

Figure 1: single optimization step with extended design space (14)

Assuming one has an adjoint computational fluid dynamics (CFD) solver to compute the aerodynamic objective sensitivities, one step of the optimization loop for the extended design space is portrayed by figure 1. Note that this is a simplified multidisciplinary model, where a fluid-structure interaction, i.e. a tight coupling between the CFD and FEM solvers is not considered.

3 Preliminary Results

The current design space $x$ is limited to shape design parameters $\alpha$, i.e. FEM node coordinates in this case. At this moment, the adjoint FEM solver allows the computation of the structural sensitivities (5) using both forward and reverse AD. As shown in figure 2, AD computed sensitivities match up well compared to finite difference (FD) approximations up to a certain degree. Forward AD and reverse AD computed sensitivities match up exactly.

![Figure 2: comparison of gradient (5) computed using forward AD vs reverse AD vs FD](image)

4 Conclusion and Further Work

We've introduced the continuing developments of a discrete adjoint structural solver for simultaneous shape and material optimization. Currently, structural constraints with respect to shape design parameters can be computed. Implementations are currently underway to include composite material capabilities such that the design space can be extended to include material design parameters $\beta$ as described in section 2.2. Future work would involve differentiating the FEM mesh generation and CAD kernel to replace the current node-based shape parametrization with a CAD-based parametrization.

5 Acknowledgements

The authors would like to acknowledge the financial support of the IODA project, which is financed by the European Commission under the Horizon 2020 program.
References


Binomial Checkpointing for Arbitrary Programs with No User Annotation

Jeffrey Mark Siskind* and Barak A. Pearlmutter†

April 2016

Heretofore, automatic checkpointing at procedure-call boundaries [1], to reduce the space complexity of reverse mode, has been provided by systems like Tapenade [2]. However, binomial checkpointing, or treeverse [3], has only been provided in AD systems in special cases, e.g., through user-provided pragmas on DO loops in Tapenade, or as the nested taping mechanism in ADOL-C for time integration processes, which requires that user code be refactored. We present a framework for applying binomial checkpointing to arbitrary code with no special annotation or refactoring required. This is accomplished by applying binomial checkpointing directly to a program trace. This trace is produced by a general-purpose checkpointing mechanism that is orthogonal to AD.

Consider the code fragment in Listing 1. This example, \( y = f(x) \), while contrived, is a simple caricature of a situation that arises commonly in practice, e.g., in adaptive grid methods. Here, the duration of the inner loop varies wildly as some function \( l(x, i) \) of the input and the outer loop index, perhaps \( 2^{\lfloor \log(n) \rfloor - \lfloor (1+\log(1007^{\lfloor \log n \rfloor}) \mod n) \rfloor} \), that is small on most iterations of the outer loop but \( O(n) \) on a few iterations. Thus the optimality of the binomial schedule is violated. The issue is that the optimality of the binomial schedule holds at the level of primitive atomic computations but this is not reflected in the static syntactic structure of the source code. Often, the user is unaware or even unconcerned with the micro-level structure of atomic computations and does not wish to break the modularity of the source code to expose such. Yet the user may still wish to reap the benefits of an optimal binomial checkpointing schedule [4]. Moreover, the relative duration of different paths through a program may vary from loop iteration to loop iteration in a fashion that is data dependent, as shown by the above example, and not even statically determinable. We present an implementation strategy for checkpointing that does not require user placement of checkpoints and does not constrain checkpoints to subroutine boundaries, DO loops, or other syntactic program constructs. Instead, it can automatically and dynamically introduce a checkpoint at an arbitrary point in the computation that need not correspond to a syntactic program unit.

We have previously introduced vlad, a pure functional language with built-in AD operators for both forward and reverse mode. Here, we adopt slight variants of these operators with the following signatures.

\[
\begin{align*}
\mathcal{F} : f \ x \ x' &\rightarrow y \ y' \\
\mathcal{J} : f \ x \ y &\rightarrow y \ x
\end{align*}
\]

The \( \mathcal{F} \) operator calls a function \( f \) on a primal \( x \) with a tangent \( x' \) to yield a primal \( y \) and a tangent \( y' \). The \( \mathcal{J} \) operator calls a function \( f \) on a primal \( x \) with a cotangent \( y \) to yield a primal \( y \) and a cotangent \( x \).

Here, we restrict ourselves to the case where (co)tangents are ground data values, i.e., reals and (arbitrary) data structures containing reals and other scalar values, but not functions (i.e., closures). For our purposes, the crucial aspect of the design is that the AD operators are provided within the language, since these provide the portal to the checkpointing mechanism.

In previous work, we introduced StalInV, a highly optimizing compiler for vlad. Here, we formulate a simple evaluator (interpreter) for vlad (Fig. 1) and extend such to perform binomial checkpointing. The operators \( \circ \) and \( \bullet \) range over the unary and binary basis functions respectively. This evaluator is written in what is known in the programming-language community as direct style, where functions (in this case \( \mathcal{E} \), denoting ‘eval’, \( \mathcal{A} \) denoting ‘apply’, and the implementations of \( \mathcal{F} \) and \( \mathcal{J} \) in the host) take inputs as function-call arguments and yield outputs as function-call return values [5]. AD is performed by overloading the basis functions in the host, in a fashion similar to FADBAD++

---

*Corresponding Author, School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu
†Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net

Listing 1: Fortran example

```
subroutine f(x, y)
n = 100003
y = x
c$ad binomial-ckp n+1 30 1
do i = 1, n
 m = l(x, i)
do j = 1, m
 y = y+y
 end do
end do
end do
```

Figure 1: Direct-style evaluator for vlad.
[6], $x \triangleright \hat{x}$ denotes recursively bundling a data structure containing primals with a data structure containing tangents, or alternatively recursively unbundling such when used as a binder, and $y \triangleleft \hat{y}$ denotes running the reverse sweep on the tape $y$ with the output cotangent $\hat{y}$, or alternatively extracting the primal $y$ and input cotangent $\hat{x}$ from the tape when used as a binder $y \triangleleft \hat{x}$.

We introduce a new AD operator $\mathcal{J}$ to perform binomial checkpointing. The crucial aspect of the design is that the signature (and semantics) of $\mathcal{J}$ is identical to $\mathcal{F}$; they are completely interchangeable, differing only in the space/time complexity tradeoffs. This means that code need not be modified to switch back and forth between ordinary reverse mode and binomial checkpointing, save interchanging calls to $\mathcal{F}$ and $\mathcal{J}$.

Conceptually, the behavior of $\mathcal{J}$ is shown in Fig. 2. In this inductive definition, a function $f$ is split into the composition of two functions $g$ and $h$ in step 1, the checkpoint $u$ is computed by applying $g$ to the input $x$ in step 2, and the cotangent is computed by recursively applying $\mathcal{J}$ to $h$ and $g$ in steps 3 and 4. This divide-and-conquer behavior is terminated in a base case, when the function $f$ is small, at which point the cotangent is computed with $\mathcal{F}$, in step 0. If step 1 splits a function $f$ into two functions $g$ and $h$ that take the same number of computational steps, the recursive divide-and-conquer process yields the logarithmic asymptotic space/time complexity of binomial checkpointing.

The central difficulty in implementing the above is performing step 1, namely splitting a function $f$ into two functions $g$ and $h$, ideally ones that take the same number of computational steps. A sophisticated user can manually rewrite a subroutine $f$ into two subroutines $g$ and $h$. A sufficiently powerful compiler or source transformation tool might also be able to, with access to nonlocal program text. But an overloading system, with access only to local information, would not be able to.

We solve this problem by providing an interface to a general-purpose checkpointing mechanism orthogonal to AD.

```
PRIMOPS f x \rightarrow (y, n)
CHECKPOINT f x n \rightarrow u
RESUME u \rightarrow y
```

This interface allows (a) determining the number of steps of a computation, (b) interrupting a computation after a specified number of steps, usually half the number of steps determined by the mechanism in (a), and (c) resuming an interrupted computation to completion. A variety of implementation strategies for this interface are possible. We present one in detail momentarily and briefly discuss others below.

Irrespective of how one implements the general-purpose checkpointing interface, one can use it to implement $\mathcal{J}$ as shown in Fig. 3. The function $f$ is split into the composition of two functions $g$ and $h$ by taking $g$ as $\lambda x.\text{CHECKPOINT } f x n$, where $n$ is half the number of steps determined by $\text{PRIMOPS } f x$, and $h$ as $\lambda u.\text{RESUME } u$.

One way of implementing the general-purpose checkpointing interface is to convert the evaluator from direct style to continuation-passing style (CPS, [7]), where functions (in this case $\mathcal{E}$, $\mathcal{A}$, $\mathcal{J}$, and $\mathcal{F}$ in the host) take an additional continuation input $k$ and instead of yielding outputs via function-call return, do so by calling the continuation with said output as arguments (Fig. 5). In such a style, functions never return; they just call their continuation. With tail-call merging, such corresponds to a computed go to and does not incur stack growth. This crucially allows the interruption process to actually return a checkpoint data structure containing the saved state of the evaluator, including its continuation, allowing the evaluation to be resumed by calling the evaluator with this saved state. This ‘level shift’ of return to calling a continuation allowing an actual return to constitute checkpointing interruption is analogous to the way backtracking is classically implemented in Prolog, with success implemented as calling a continuation and failure implemented as actual return. In our case, we further instrument the evaluator to thread two values as inputs and outputs: the count $n$ of the number of evaluation steps, which is incremented at each call to $\mathcal{E}$, and the limit $l$ of the number of steps, after which a checkpointing interrupt is triggered.

With this CPS evaluator, it is possible to implement the general-purpose checkpointing interface (Fig. 4), not for programs in the host, but for programs in the target; hence our choice of formulating the implementation around an evaluator (interpreter). We remove this restriction below. The implementation of PRIMOPS calls the evaluator with no limit and simply counts the

```
PRIMOPS f x = A (\lambda n l v. (v, n))) \ 0 \ f x
CHECKPOINT f x n = A \ 0 \ n \ f x
RESUME [k, l, p, e] = E k l \ p \ e
```

Figure 4: Implementation of the general-purpose checkpointing interface using the CPS evaluator.
number of steps to completion. The implementation of checkpoint calls the evaluator with a limit that must be smaller than that needed to complete so a checkpointing interrupt is forced and the checkpoint data structure \([k, l, \rho, e]\) is returned. The implementation of resume calls the evaluator with arguments from the saved checkpoint data structure.

With this, it is possible to reformulate the Fortran example from Listing 1 in VLAD (Listing 2). Then one achieves binomial checkpointing simply by calling \(\mathcal{F} f\).

The efficacy of our method can be seen in the plots (Fig. 5) of the space and time usage, relative to that for the leftmost datapoint, of the above Fortran and VLAD examples with varying \(n\). Tapenade was run without checkpointing, with manual checkpointing only around the body of the outer loop, with manual checkpointing only around the body of the inner loop, with manual checkpointing around the bodies of both loops, and with binomial checkpointing. VLAD was run with \(\mathcal{F}\) and \(\mathcal{J}\). Note that Tapenade exhibits \(O(n)\) space and time usage for all cases, while VLAD exhibits \(O(n)\) space and time usage with \(\mathcal{J}\), but \(O(1)\) space usage and \(O(n)\) time usage with \(\mathcal{F}\). The space complexity of \(\mathcal{F}\) is the sum of the space required for the checkpoints and the space required for the tape. For a general computation of length \(t\) and maximal live storage \(w\), the former is \(O(w \log t)\) while the latter is \(O(w)\). For the code in our example, \(t = O(n)\) and \(w = O(1)\), leading to the former being \(O(\log n)\) and the latter being \(O(1)\). We observe \(O(1)\) space usage since the constant factors of the latter overpower the former. The time complexity of \(\mathcal{F}\) is the sum of the time required to (re)compute the primal and the time required to perform the reverse sweep. For a general computation, the former is \(O(t \log t)\) while the latter is \(O(t)\). For the code in our example, the former is \(O(n \log n)\) and the latter is \(O(n)\). We observe \(O(n)\) time usage since, again, the constant factors of the latter overpower the former.

Figure 5: CPS evaluator for VLAD.

Listing 2: VLAD example

```lisp
(define (f x)
  (let ((n 100003))
    (let outer ((i 1) (y x))
      (if (> i n)
        y
        (outer (+ i 1)
          (let ((m (l x i)))
            (let inner ((j 1) (y y))
              (if (> j m)
                y
                (inner (+ j 1)
                  (sqrt (* y y))))))))))
```

Figure 5: Space and time usage of reverse-mode AD with various checkpointing strategies, relative to the space and time for the first datapoint for each respective strategy.
Other methods present themselves for implementing the general-purpose checkpointing interface. One can use POSIX `fork()` much in the same way that it has been used to implement the requisite nondeterminism in probabilistic programming languages like probabilistic C [8]. A copy-on-write implementation of `fork()`, as is typical, would make this reasonably efficient and allow it to apply in the host, rather than the target, and thus could be used to provide an overloaded implementation of binomial checkpointing in a fashion that was largely transparent to the user. Alternatively, direct-style code could be compiled into CPS using a CPS transformation. A compiler for a language like VLD can be constructed that generates target code in CPS that is instrumented with step counting, step limits, and checkpointing interruptions. A driver can be wrapped around such code to implement \( \mathcal{J} \). Existing high-performance compilers, like SML/NJ [9], for functional languages like SML, already generate target code in CPS, so by adapting such to the purpose of AD with binomial checkpointing, it seems feasible to achieve high performance. In fact, the overhead of the requisite instrumentation for step counting, step limits, and checkpointing interruptions need not be onerous because the step counting, step limits, and checkpointing interruptions for basic blocks can be factored, and those for loops can be hoisted, much as is done for the instrumentation needed to support storage allocation and garbage collection in implementations like MLTON [10], for languages like SML, that achieve very low overhead for automatic storage management.

Acknowledgments

This work was supported, in part, by NSF grant 1522954-IIS and by Science Foundation Ireland grant 09/IN.1/I2637. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References


Efficient Implementation of a Higher-Order Language with Built-In AD

Jeffrey Mark Siskind* Barak A. Pearlmutter†

April 2016

We show that AD operators can be provided in a dynamic language without sacrificing numeric performance. To achieve this, general forward and reverse AD functions are added to a simple high-level dynamic language, and support for them is included in an aggressive optimizing compiler. Novel technical mechanisms are discussed, which have the ability to migrate the AD transformations from run-time to compile-time. The resulting system, although only a research prototype, exhibits startlingly good performance. In fact, despite the potential inefficiencies entailed by support of a functional-programming language and a first-class AD operator, performance is competitive with the fastest available preprocessor-based Fortran AD systems. On benchmarks involving nested use of the AD operators, it can even dramatically exceed their performance.

The Problem

Numerical programmers face a tradeoff. They can use a high-level language, like MATLAB or Python, that provides convenient access to mathematical abstractions like function optimization and differential equation solvers, or they can use a low-level language, like FORTRAN, to achieve high computational performance. The convenience of high-level languages results in part from the fact that they support many forms of run-time dependent computation: storage allocation and automatic reclamation, data structures whose size is run-time dependent, pointer indirection, closures, indirect function calls, tags and tag dispatching, etc. This comes at a cost to the numerical programmer: the instruction stream contains a mix of floating-point instructions and instructions that form the scaffolding that supports run-time dependent computation. FORTRAN code, in contrast, achieves high floating-point performance by avoiding dilution of the instruction stream with such scaffolding.

This tradeoff is particularly poignant in the domain of automatic differentiation. Since the derivative is a higher-order function, it is most naturally incorporated into a language that supports higher-order functions in general. But on the other hand, efficiency of AD is often critical.

AD Implementation Strategies

One approach to AD involves a preprocessor performing a source-to-source transformation. In its simplest form, this can be viewed as translating a function:

```cpp
double f(double x) {...}
```

into:

```cpp
struct bundle {double primal; double tangent;};
struct bundle f_forward(struct bundle x) {...}
```

that, when passed a bundle of \(x\) and \(\bar{x}\), returns a bundle of the primal value \(f(x)\) and the tangent value \(\bar{x} f'(x)\). When implemented properly, repeated application of this transformation can be used to produce variants of \(f\) that compute higher-order derivatives. Herein lies the inconvenience of this approach. Different optimizers might use derivatives of different order. Changing code to use a different optimizer would thus entail changing the build process to transform the objective function a different number of times. Moreover, the build process for nested application, such as multilevel optimization, would be tedious. One would need to transform the inner objective function, wrap it in a call to `optimize`, and then transform this resulting outer function.

A High-Performance Testbed Dynamic Language and Aggressive Compiler

We present a powerful and expressive formulation of forward AD based on a novel set of higher-order primitives, and develop the novel implementation techniques necessary to support highly efficient implementation of this formulation. We demonstrate that it is possible to combine the speed of FORTRAN with the expressiveness of a higher-level functional-programming language augmented with first-class AD.

We exhibit a small but powerful language that provides a mechanism for defining a derivative operator that offers the convenience of the first approach with the efficiency of the second approach. This mechanism is formulated

---

*Corresponding Author, School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu
†Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net
in terms of run-time reflection on the body of f, when computing (derivative f), to transform it into something like f-forward. An optimizing compiler then uses whole-program inter-procedural flow analysis to eliminate such run-time reflection, as well as all other run-time scaffolding, yielding numerical code with FORTRAN-like (or super-FORTRAN) efficiency.

These results are achieved by combining (a) a novel formulation of forward and reverse AD in terms of a run-time reflexive mechanism that supports first-class nestable nonstandard interpretation with (b) the migration of the nonstandard interpretation to compile-time by whole-program inter-procedural flow analysis.

It should be noted that the implementation techniques invented for this purpose are, in principle, compatible with the optimizing compilers for procedural languages like Fortran. In other words, these techniques could be used to add in-language AD constructs to an aggressive optimizing Fortran or C compiler. In fact, a proof-of-concept has been exhibited which uses a small subset of these methods to build a Fortran AD pre-preprocessor which accepts a dialect of Fortran with in-language AD block constructs and which allows EXTERNAL FUNCTION arguments, and rewrites/expands the code, generating pure Fortran, and then uses existing tools like Tapenade to perform the required AD [1, 2].

**Sketch of Implementation Technology**

We present a novel approach that attains the advantages of both the overloading and transformation approaches. We define a novel functional-programming language, vlad,\(^1\) that contains mechanisms for transforming code into new code that computes derivatives.\(^2\) These mechanisms apply to the source code that is, at least conceptually, part of closures, and such transformation happens, at least conceptually, at run time. Such transformation mechanisms replace the preprocessor, support a callee-derives programming style where the callee invokes the transformation mechanisms on closures provided by the caller, and allow the control flow of a program to determine the transformations needed to compute derivatives of the requisite order. Polyvariant flow analysis is then used to migrate the requisite transformations to compile time.\(^3\)

We present a compiler that generates FORTRAN-like target code from a class of programs written in a higher-order functional-programming language with a first-class derivative operator. Our compiler uses whole-program inter-procedural flow analysis to drive a code generator. Our approach to flow analysis differs from that typically used when generating non-FORTAN-like code. First, it is polyvariant. Monovariant flow analyses like 0-CFA [7] are unable to specialize higher-order functions. Polyvariant flow analysis is needed to do so. The need for polyvariant flow analysis is heightened in the presence of a higher-order derivative operator, i.e., one that maps functions to their derivatives. Second, it is union free. The absence of unions in the abstract interpretation supports generation of code without tags and tag dispatching. The further absence of recursion in the abstract interpretation means that all aggregate data will have fixed size and shape that can be determined by flow analysis allowing the code generator to use unboxed representations without indirection in data access or run-time allocation and reclamation. The polyvariant analysis determines the target of all call sites allowing the code generator to use direct function calls exclusively. This, combined with aggressive inlining, results in inlined arithmetic operations, even when such operations are performed by (overloadable) function calls. The polyvariant analysis unrolls finite instances of what is written as recursive data structures. This, combined with aggressive unboxing, eliminates essentially all manipulation of aggregate data, including closures. Our limitation to union-free analyses and finite unrolling of recursive data structures is not as severe a limitation as it may seem. The main limitation relative to FORTRAN-like code is that we currently do not support arrays, though this restriction is easily lifted. Finally, the polyvariant analysis performs finite instances of reflection, migrating such reflective access to and creation of code from run time to compile time. This last aspect of our flow analysis is novel and crucial. Our novel AD primitives perform source-to-source transformation within the programming language rather than by a preprocessor, by reflective access to the code associated with closures and the creation of new closures with transformed code. Our flow analysis partially evaluates applications of the AD primitives, thus migrating such reflective access and transformation of code to compile time.

For those who are not compiler experts, an intuition for these techniques would be that hunks of code, including both entire procedures and smaller code blocks within procedures, are duplicated to make specialized versions for the different arguments that actually occur. For example, if an optimization routine argmax is called from two different places in the program using two different objective functions, two versions of argmax would be generated, each specialized to one of the objective functions. If argmax takes a gradient of the objective function it is passed, each of these two versions of argmax would have a known objective function, which would allow the AD transformation of that function to be migrated to compile time. Similar machinations allow much of the scaffolding around the numeric computation to be removed.

The effectiveness of these methods at attaining high floating point performance should be apparent from the benchmarking results shown in Figure 1.

---

\(^1\)VLAD is an acronym for Functional Language for AD with a voiced F.

\(^2\)This differs from previous work on forward AD in a functional context [3, 4, 5, 6] which adopts an overloading approach.

\(^3\)Existing transformation-based AD preprocessors, like ADIFOR and Tapenade, use inter-procedural flow analysis for different incomparable purposes: not to eliminate run-time reflection, but to determine which subroutines to transform and which variables need tangents.
<table>
<thead>
<tr>
<th>AD Tool</th>
<th>Particle (s)</th>
<th>Saddle (s)</th>
<th>Probabilistic-Lambda-Calculus (s)</th>
<th>Probabilistic-Prolog (s)</th>
<th>Backprop (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLAD</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>STALIN</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>ADFOR</td>
<td>2.05</td>
<td>5.44</td>
<td>15.51</td>
<td>1.51</td>
</tr>
<tr>
<td></td>
<td>TAPENADE</td>
<td>5.51</td>
<td>8.09</td>
<td>14.97</td>
<td>22.75</td>
</tr>
<tr>
<td>C</td>
<td>ADIC</td>
<td></td>
<td></td>
<td></td>
<td>12.16</td>
</tr>
<tr>
<td>C++</td>
<td>ADOL-C</td>
<td></td>
<td></td>
<td></td>
<td>2.05</td>
</tr>
<tr>
<td>CppAD</td>
<td></td>
<td></td>
<td>93.32</td>
<td>132.31</td>
<td>12.16</td>
</tr>
<tr>
<td>FADBAD++</td>
<td></td>
<td></td>
<td>93.32</td>
<td>132.31</td>
<td>12.16</td>
</tr>
<tr>
<td>ML</td>
<td>MLTON</td>
<td>20.44</td>
<td>247.57</td>
<td>180.43</td>
<td>415.64</td>
</tr>
<tr>
<td>OCAML</td>
<td></td>
<td>217.03</td>
<td>291.26</td>
<td>249.40</td>
<td>54.74</td>
</tr>
<tr>
<td>SML/NJ</td>
<td></td>
<td>153.01</td>
<td>271.84</td>
<td>234.62</td>
<td>181.93</td>
</tr>
<tr>
<td>SCHEME</td>
<td>GHHC</td>
<td>627.78</td>
<td>1004.85</td>
<td>983.12</td>
<td>1283.29</td>
</tr>
<tr>
<td>SCHEME</td>
<td>Bigloo</td>
<td>1453.06</td>
<td>2276.69</td>
<td>2324.54</td>
<td>1283.29</td>
</tr>
<tr>
<td></td>
<td>Chicken</td>
<td>578.94</td>
<td>958.73</td>
<td>1033.46</td>
<td>915.28</td>
</tr>
<tr>
<td></td>
<td>Gambit</td>
<td>266.56</td>
<td>424.75</td>
<td>497.48</td>
<td>743.26</td>
</tr>
<tr>
<td></td>
<td>Ikarus</td>
<td>964.18</td>
<td>1565.53</td>
<td>1658.27</td>
<td>1203.34</td>
</tr>
<tr>
<td></td>
<td>Larceny</td>
<td>260.98</td>
<td>958.73</td>
<td>1033.46</td>
<td>743.26</td>
</tr>
<tr>
<td></td>
<td>MIT Scheme</td>
<td>291.26</td>
<td>114.07</td>
<td>129.11</td>
<td>1203.34</td>
</tr>
<tr>
<td></td>
<td>MzC</td>
<td>958.73</td>
<td>114.07</td>
<td>129.11</td>
<td>1203.34</td>
</tr>
<tr>
<td></td>
<td>MsScheme</td>
<td>260.98</td>
<td>958.73</td>
<td>1033.46</td>
<td>743.26</td>
</tr>
<tr>
<td></td>
<td>Scheme-&gt;C</td>
<td>291.26</td>
<td>114.07</td>
<td>129.11</td>
<td>1203.34</td>
</tr>
<tr>
<td></td>
<td>Scmutils</td>
<td>627.78</td>
<td>1004.85</td>
<td>983.12</td>
<td>1283.29</td>
</tr>
<tr>
<td></td>
<td>Stalin</td>
<td>1453.06</td>
<td>2276.69</td>
<td>2324.54</td>
<td>1283.29</td>
</tr>
</tbody>
</table>

Figure 1: Comparative benchmark results for the particle and saddle examples [8], the probabilistic-lambda-calculus and probabilistic-prolog examples [9] and an implementation of backpropagation in neural networks using AD. Column labels are for AD modes and nesting: F for forward, Fv for forward-vector aka stacked tangents, RF for reverse-over-forward, etc. All run times normalized relative to a unit run time for STALIN\(\nabla\) on the corresponding example except that run times for backprop-Fv are normalized relative to a unit run time for STALIN\(\nabla\) on backprop-F. Pre-existing AD tools are named in blue, others are custom implementations. Key: ■ not implemented but could implement, including FORTRAN, C, and C++; ● not implemented in pre-existing AD tool; □ problematic to implement. All code available at http://www.bcl.hamilton.ie/~qobi/ad2016-benchmarks/.
Novelty and Significance

This paper makes two specific novel contributions:


2. A novel approach for using polyvariant flow analysis to eliminate such run-time reflection along with all other non-numerical scaffolding.

These are significant because they support AD with a programming style that is much more expressive and convenient than that provided by the existing preprocessor-based source-to-source transformation approach, yet still provides the performance advantages of that approach.

An alternative perspective would be that the language discussed here is in simple terms an eager lambda calculus augmented with a numeric basis and an AD basis. This is isomorphic to the intermediate forms used inside high-performance compilers, including aggressive optimizing Fortran compilers. As such, the techniques we discuss are useful not only for adding AD operators to functional programming languages, but also for adding AD support to compilers for imperative languages.

Acknowledgments

This work was supported, in part, by NSF grant 1522954-IIS and by Science Foundation Ireland grant 09/IN.1/I2637. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References


[3] Jerzy Karczmarczuk. Functional differentiation optimizing FORTRAN compilers. As such, the techniques we discuss are useful not only for adding AD operators to functional programming languages, but also for adding AD support to compilers for imperative languages.

Acknowledgments

This work was supported, in part, by NSF grant 1522954-IIS and by Science Foundation Ireland grant 09/IN.1/I2637. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References


A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Machine Learning and Computer Vision

Filip Srajer\(^1\)\(^*\) Zuzana Kukelova\(^2\) Andrew Fitzgibbon\(^2\)

\(^1\)Czech Technical University in Prague

srajefil@fel.cvut.cz

\(^2\)Microsoft Research Ltd

\{zukuke,awf\}@microsoft.com

1 Introduction

We look at three objectives from machine learning and computer vision, and fifteen ways of computing their derivatives (including hand-derived and finite differences). These objectives are for the most part simple, in the sense that no iterative loops are involved, and conditional statements are encapsulated in functions such as \texttt{abs} or \texttt{logsumexp}.

However, it is important for the success of algorithmic differentiation that such “simple” objective functions are handled efficiently, as so many problems in these fields are of this form.

The objective functions considered are: log-likelihood of a Gaussian mixture model, bundle adjustment, and hand tracking. These functions include features such as sparse Jacobians, matrix expressions, and domain-specific special functions such as \texttt{logsumexp(x: vector)}, defined stably as \texttt{log(sum(exp(v - max(v)))) + max(v)}.

We first describe the objective functions used for benchmarking. Next, we give an overview of the selected AD tools. Then, we present the results and finally give our conclusions, foremost among which is that even with reasonable care devoted to efficiency in each of the input languages, the runtimes vary through four orders of magnitude.

Objective GMM: Gaussian Mixture Model Fitting

The Gaussian mixture model with Wishart prior has log-posterior function

\[
\log\left( p(x; w, \mu, \Sigma) \right) = \log\left( \prod_{i=1}^{N} \sum_{k=1}^{K} w_k \det(2\pi \Sigma_k)^{-\frac{1}{2}} \exp \left( -\frac{1}{2} (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) \right) \right)
\]

\[\times \prod_{k=1}^{K} C(D, m) \mid \Sigma_k \mid^n \exp \left( -\frac{1}{2} \text{trace}(\Sigma_k) \right)\]

\[\text{s.t.} \sum_{k=1}^{K} w_k = 1 \text{ and } \Sigma_k \text{ is positive-semidefinite } \forall k \in \{1, \ldots, K\}\]

where \(x \in \mathbb{R}^{D \times N}\) are data points, \(w \in \mathbb{R}^K\) weights, \(\mu \in \mathbb{R}^{D \times K}\) means, \(\Sigma \in \mathbb{R}^{D \times D \times K}\) covariance matrices, \(m\) is a Wishart hyperparameter and \(C\) is a function not dependent on independent variables. To integrate the constraints on weights and covariances into the objective function, we reparametrize the GMM function (1). After simplification, the final function to be optimized looks like

\[
\log(p(x; \alpha, \mu, q, l)) = \sum_{i=1}^{n} \log\text{sumexp} \left( \left[ \alpha_k + \text{sum}(q_k) - \frac{1}{2} ||Q(q_k, l_k)(x_i - \mu_k)||^2 \right]_{k=1}^{K} \right) - n \log\text{sumexp}(\alpha) + \frac{1}{2} \sum_{k=1}^{K} \left( ||\exp(q_k)||^2 + ||l_k||^2 \right) - m \sum(q_k) + C'(D, m)
\]

where \(\alpha \in \mathbb{R}^K\) corresponds to weights, \(q \in \mathbb{R}^{D \times K}\) and \(l \in \mathbb{R}^{\frac{D(d-1)}{2} \times K}\) to covariance matrices, \(C'\) does not depend on independent variables and \(Q\) assembles a \(D \times D\) lower triangular matrix.

We benchmark the AD tools on gradient computation of Eq. (2). The size of the gradient changes with \(D\) and \(K\), while \(\alpha, \mu, q\) and \(l\) are independent variables.

*Corresponding Author
Objective BA: Bundle Adjustment

Consider a weight \( w \in \mathbb{R} \), a 3D point \( X \in \mathbb{R}^3 \) and a camera with parameters \( p = [r; C; f; x_0; \kappa] \in \mathbb{R}^{11} \), i.e., rotation, camera center, focal length, principal point and radial distortion. The point \( X \) can be projected by the camera as

\[
\text{project}(r, C, f, x_0, \kappa, X) = \text{distort}(\kappa, p2e(\text{rodriguez}(r, X - C)))f + x_0
\]

\[
\text{distort}(\kappa, x) = x(1 + \kappa_1||x||^2 + \kappa_2||x||^4)
\]

\[
p2e(X) = X_{1,2}/X_3
\]

\[
\text{rodriguez}(r, X) = X \cos \theta + (v \times X) \sin \theta + v(v^T X)(1 - \cos \theta), \quad \theta = ||r||, v = \frac{r}{||r||}
\]

The observed image point is \( m \in \mathbb{R}^2 \) and the residual \( e \) concatenates its reprojection error and \( w \)'s regularizer:

\[
e = \left[w(m - \text{project}(r, C, f, x_0, \kappa, X))^\top; 1 - w^2\right]^\top
\]

The goal of BA is to optimize 3D points, camera parameters and weights simultaneously in a system with multiple cameras and points [1][2]. The Jacobian has only 15 non-zero entries in every reprojection-error row and one non-zero in every weight-term row. This is typically [2] exploited by computing only small independent Jacobians and then inserting them into the large sparse one.

Objective HT: Hand Tracking

The hand tracking problem [3] has independent variable \( p \in \mathbb{R}^{26} \) parameterizing motion. A hand is modelled by a set of points \( X \in \mathbb{R}^{3 \times N} \) and their triangulation, i.e., the model is a collection of adjacent triangles, which make up a surface. Then, there are \( N \) correspondences between observed data points \( Y \in \mathbb{R}^{3 \times N} \) and the triangles. The \( q \)-th correspondence has independent variable \( u_q \in \mathbb{R}^2 \) defining an exact spot inside a triangle to which \( Y_q \) corresponds.

The procedure for computing the error for all measurements is the following:

1. Compute \( Z \in \mathbb{R}^{3 \times M} \) by transforming \( X \) according to \( p \). This includes constructing transformation matrices (similar approach as Eq. [3]) and multiplying the matrices with the points.
2. For \( q \)-th measurement corresponding to the triangle \((i, j, k)\), compute \( W_q = u_{q,1}Z_i + u_{q,2}Z_j + (1 - u_{q,1} - u_{q,2})Z_k \).
3. For \( q \)-th measurement, compute error \( e_q = Y_q - W_q \).

The Jacobian has a dense part composed of columns of \( p \) and a sparse part corresponding to \( u \), where every row has two non-zero entries. AD tools supporting sparsity compute the whole sparse part with only two passes. Several tools are not run as they do not support dynamic sparsity (in the tested version [2]).

2 Tools

We have chosen several well-known or promising AD tools (see Tab. [1]). The selection covers various languages and AD approaches as well as symbolic differentiation. The newest version of all the tools that was available in the period July-August 2015 was used. In addition, we give results for finite differences and manual, i.e., a hand-derived optimized implementation.

From the chosen tools, we did not benchmark clad as it does not have support for arrays, ADiGator because it generated syntactically incorrect code for GMM and ADIC2 as our attempts to compile it were unsuccessful.

For tools that have both forward and reverse mode, they are called with the one that is more suitable for the given objective. Diffsharp in particular runs significantly slower in default mode so it is called in its special forward and reverse modes for first-order derivatives. Tapenade offers differentiation of both clean C and Fortran code but we use it only with C. Unfortunately, its source transformation occasionally produces non-compiling output, so the user has to fix a few errors. MuPAD optimizes code using common subexpression elimination and compiles it via C++ to MEX. Theano is written in a modified Python and compiles either into optimized Python or C++. Theano is always run in CPU mode to allow a fair comparison since all the tools use only CPU.

3 Experiments

To benchmark the AD tools, we first ran pre-processing routines (e.g. source transformation, symbolic differentiation, taping). All of the routines that need to be run only once for different data are not included in the runtimes that we provide. This is justified since a user of AD tools would typically run it only once on the objective before calling the differentiated function many times to optimize parameters.

The benchmarking is done on random data. The resulting runtimes are averaged over 1000 runs if one run is less than 5 seconds, over 100 runs if 5-30 seconds and over 10 runs if 30-120 seconds. Otherwise, the runtimes are not
Table 1: List of tools. OO: operator overloading, ST: source transformation; F: forward, R: reverse, •: not run (see text). For BA and HT, runtimes are shown in milliseconds, to 2 significant digits. Theano’s AD-like R-op mode is used for HT. Standard symbolic mode would not handle sparsity.

A single machine with a processor Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz, memory 32GB and OS Windows 10 64-bit was used for all the experiments.

Fig. 1 shows gradient computation runtimes for GMM. We have noticed that some of the tools do not handle bigger instances. The biggest instance size ($D = 64$, $K = 200$) was taken from [4]. We help out some of the tools by manually splitting the gradient computation

$$\nabla \log(p(x; \alpha, \mu, q, l)) = \sum_{i=1}^{N} \nabla f(x_i; \alpha, \mu, q, l) + \nabla g(\alpha, q, l) \quad (8)$$

which is symbolized by (split) in the figures. Moreover, GMM allows for a vectorized implementation (denoted by (vector)), where most necessary computations are done in one huge matrix multiplication. We show this (vector) version with languages that are able to utilize it.

Next, Tab. 1 shows runtimes for BA. We have tried running BA on various problem sizes ranging from 21 cameras, 11k 3D points and 36k observations to 14k cameras, 4M 3D points, 29M observations. The runtimes are, as expected (see above), linearly dependent on the number of observations only. Thus, we show only runtimes for the smallest problem size.

Tab. 1 also gives results for HT. We show results for a small model suitable for a real-time application and a big one which would be run offline. The small instance has 544 3D points and 192 correspondences whereas the big one has 10k 3D points and 100k correspondences.

4 Conclusion

First, we have described three real-world objective functions from areas of machine learning and computer vision. Second, we have chosen several approaches and tools for computing derivatives to be benchmarked. Then, we have provided runtimes for computing derivatives.

We have seen that the runtimes of derivative computation range through four orders of magnitude. This is partially dependent on a programming language. It will also depend on programmer skill, and familiarity with the tools, so we have made open source all our materials[1] in order that others may improve on our efforts. However, we contend that this paper presents an important datapoint: a skilled programmer devoting roughly a week to each tool produced the timings above. For many projects, these will represent typical results achieved before a tool is selected.

We conclude that there are useful tools in most languages but there is also still some space for improvement. Availability of various features proves to be crucial for the success and efficiency of algorithmic differentiation. Important features for our objectives range from sparsity support (at least manually specified) and support of matrix libraries to memory optimizations for big problem instances and the option to choose from both forward and reverse mode. Importantly, note that we benchmarked only computation of the first-order derivatives and some tools do not support higher-order derivatives.

1https://github.com/awf/autodiff
Figure 1: Runtimes in seconds for GMM. Some of the tools were run with (split) or (vector) implementations (see Sec. 3). The black dots emphasize the end of a curve symbolizing that the tools crashed on bigger problem sizes. Note that both axes are log-scaled. Best viewed in color.

5 Acknowledgements

This work was done during an internship in Microsoft Research Ltd. We thank Jonathan Taylor for an example implementation of a hand tracking function in Python.

References

Newton-like Methods for Solving Piecewise Smooth Systems of Equations Based on Successive Piecewise Tangent or Secant Linearization

Tom Streubel†, Andreas Griewank‡ and Richard Hasenfelder†

March 2016

The Newton method and most of its variants either need or benefit from derivative information. While having actual derivatives in mind, finite difference approximations of local linear models resemble a secant based approach and thus one can expect rapid convergence in real world applications. Secants can be propagated in an AD like fashion as well, which reduces the evaluation costs as we will point out later. Algorithmic differentiation evaluates the derivative with working precision and hence makes the quadratic order of convergence of the Newton method possible on large scale problems. Even more, the forward mode of AD runs simultaneously with the code execution and hence provides a fast, efficient evaluation of the function values together with one, several or all directional derivatives as required by the user. It seems quite natural in this framework to seek out new strategies for solving problems that are not everywhere differentiable. Here we want to present a new approach to apply the principles and techniques of algorithmic differentiation to nonsmooth systems of equations \(0 = F(x)\) whose right-hand sides are piecewise smooth, or more precisely, described by a composite piecewise smooth evaluation procedure \(F : \mathbb{R}^n \rightarrow \mathbb{R}^n\) (short \(\text{CPS}^{1,1})\). These are compositions of \textit{elementary operations} from some fixed library \(\Phi_{\text{abs}} \supseteq \{\text{abs}\} \cup \{+, -, *, /, \exp, \log, \text{pow}, \ldots\}\). Apart from the absolute value function as the only exception, every other elementary operation has to be continuously differentiable with locally Lipschitz continuous first derivatives \((\text{C}^{1,1})\). One encounters such problems for example after time discretization of dynamical flow networks with flap valves

\[
0 = F_{\text{net}}(\dot{x}(t), x(t), t) \quad \text{ implicit Euler } \quad \text{Solve } 0 = F_{\text{net}}(\langle x_h - x_0 \rangle/h, x_h, h) \text{ for } x_h.
\]

Concrete examples are water distribution networks, gas transportation networks and the human cardio vascular system, which is usually modelled as an electrical circuit \([1]\). In the latter example the heart valves are represented by ideal diodes that do not allow any backflow once the flaps have closed tightly. A possible model is a piecewise linear and continuous interpolation of the binary open-close state, where the opening and closing transitions are linear in the pressure difference around the flap. A smoothing would further increase the inherent stiffness of that valve type and may not be recognized by the simulator. Instead, one can propagate piecewise linear tangent models \(\Delta F = F(\dot{x}) + \Delta F(\dot{x}; x - \dot{x})\) that preserve an error of second order \(F(x) = \Delta F(\dot{x}) + O(\|x - \dot{x}\|^2)\) (see \([2]\) for details). Here \(\|\ldots\|\) denotes the Euclidean norm and throughout \(\dot{x} \in \mathbb{R}^n\) the reference point and \(\Delta F(\dot{x}, x - \dot{x})\) a piecewise linear increment propagated by the following set of rules \([2, 4]\)

\[
\begin{align*}
\dot{v} &= c \cdot \dot{u} + d \cdot \dot{w} & \Delta v &= c \Delta u + d \Delta w & \text{(linearity)} \\
\dot{v} &= \dot{u} \cdot \dot{w} & \Delta v &= \Delta u \cdot \dot{u} + \dot{w} \cdot \Delta w & \text{(product rule)} \\
\dot{v} &= \varphi(\dot{u}) & \Delta v &= c_\varphi \cdot \Delta u & \text{(unary smooth operation)} \\
\dot{v} &= \text{abs}(\dot{u}) & \Delta v &= \text{abs}(\dot{u} + \Delta u) - \dot{v}, & \text{(absolute value function)}
\end{align*}
\]

where \(c_\varphi = \varphi'(u)\). Thus, only slight modifications of standard AD-Tools are necessary which have already been

---

†Corresponding Author, 1st affiliation: Zuse Institute Berlin, 2nd affiliation: Humboldt University of Berlin, streubel@zib.de
‡Yachay Tech, griewank@yachaytech.edu.ec
†Humboldt University of Berlin, hasenfer@cms.hu-berlin.de
performed in Adol-C by its developers. One can adapt the propagation rules above and calculate secant piecewise linearizations denoted by \( \dot{O}_x^S F(x) = (F(\dot{x}) + F(\dot{x})) / 2 + \Delta F(\dot{x}, \dot{x}; \Delta x) \). In this context we use the terms tangent and secant to clearly distinguish between them. In contrast to the tangent linearization process the secant based models achieve a bilinear approximation error \( F(x) = \dot{O}_x^S F(x) + O(\|x - \dot{x}\| \cdot \|x - \dot{x}\|) \) \(^2\). In the smooth univariate case \((n = 1)\) our secant linearization coincides with the usual secant approximation, but in the multivariate case \((n > 1)\) the latter involves \(n + 1\) evaluation points, whereas the new secant propagation process is based on just the two reference points \( \dot{x} \) and \( \dot{x} \).

Now one can define a Newton-like iteration via successive tangent or secant piecewise linearization

\[
x_{j+1} \in \arg \min \left\{\|x - x_j\| \mid 0 = \dot{O}_{x_j} F(x)\right\}, \quad x_{j+1} \in \arg \min \left\{\|x - x_j\| \mid 0 = \dot{O}_{x_{j+1}}^T F(x)\right\}.
\]

Under certain conditions the sequence produced by the tangent iteration scheme converges quadratically to a locally unique root \( x^* \). This is exactly the convergence rate of the Newton method that requires \( F \) to be smooth and have a nonsingular Jacobian. One should note here that the described piecewise linear model propagation of an abs-free evaluation graph yields already the linear Jacobi operator because the remaining propagation rules are identical to those defined in \(^3\). Then the successive tangent piecewise linear iteration and the Newton method are totally equivalent. Under similar conditions the sequence generated by the secant iteration achieves a rate of convergence of \((1 + \sqrt{5})/2\) (golden ratio) to a locally unique root \( x^* \). Both results rely on strong Lipschitz continuity properties of the propagated models

\[
\|\dot{O}_x^S F(x) - \dot{O}_x^T F(x)\| \leq \gamma_F \cdot \max \left[\|\dot{z} - \dot{y}\| \max(\|x - y\|, \|x - \dot{z}\|)\right], \quad \text{(secant linearization)}
\]

\[
\|\dot{O}_x F(x) - \dot{O}_x^T F(x)\| \leq \gamma_F \cdot \max(\|x - \dot{y}\|, \|x - \dot{z}\|), \quad \text{(tangent linearization)}
\]

Here \( \gamma_F \) and \( \beta_F \) are Lipschitz constants that can be propagated tightly through the evaluation procedure of \( F \). This stability of the local models may appear trivial in the smooth case, but it is definitively not in our setting, where the usual generalized derivative concepts are very unstable. Regarding the solvability of both iterations \(^1\) we have found out, using Brouwer’s fixed point theorem, that the injectivity of a model developed at the desired root \( x^* \) of \( F \) guarantees the existence of at least one nearby root of all models \( \dot{O}_x F \) propagated at any base point \( \dot{x} \) in the local neighborhood. For the general characterization of the solvability of systems of piecewise linear equations \( H(x) = 0 \) the chain

\[
H \text{ is injective} \implies H \text{ is coherently oriented} \implies H \text{ is surjective}
\]

of implications is crucial (proofs can be found in \(^5\)). Coherent orientation is equivalent to \( H \) being an open mapping, but can also be understood as follows: Piecewise linear functions are almost everywhere differentiable. This follows from Rademacher’s Theorem since piecewise linear functions are Lipschitz continuous \(^5\). So a piecewise linear function is called coherently oriented if all Jacobian matrices developed wherever the function is differentiable have positive determinants. The piecewise Newton method as described in \(^2\) takes advantage of this property. In \(^2\) it was observed that coherent orientation implies that the fibres \( \{\dot{x} \mid x \in \mathbb{R}^n \mid H(x) = \lambda H(\dot{x}), 0 < \lambda \in \mathbb{R}\} \) are bifurcation-free piecewise linear paths for almost all \( \dot{x} \in \mathbb{R}^n \). Then their closure contains a solution, but even in the case of singular fibres, there are efficient strategies to reduce the residual towards a solution. Many other approaches for direct methods are discussed in \(^6\). Theoretically it is also possible to transform systems of piecewise linear equations into linear complementarity problems \(^7\) since there is a one to one solution correspondence \(^6\). However, the transformation process is expensive and will substantially alter the structure of nonsmoothness. Another promising and semi-direct approach for solving systems of piecewise linear equations \( 0 = H(x) \) is to find a global minimizer \( x^* \) of its norm \( h_\infty(x) = \|H(x)\|_\infty \). A suitable optimization process is described in \(^8\). It will be globally convergent if the system function \( H \) is coherently oriented. Furthermore, this approach is still applicable or naturally extendable in under- or overdetermined cases. We will verify the convergence results numerically on two examples: first the successive tangent piecewise linearization on the cardio vascular model as described above, second, an academic example that we can perturb with a small but highly oscillatory. In that case the tangent based successive piecewise linearization scheme
demonstrates a very erratic and unstable convergence behavior for slight disturbances of the initial point.

Outlook

In the future we intend to prove a semilocal convergence statement for the successive tangent and secant piecewise linearization. We would like to find estimates for the Newton steps $\| x_{j+1} - x_j \|$ and use this to derive new semi-implicit methods for solving differential algebraic and ordinary differential equations as it was done in [9] for smooth system functions. We also want to apply this AD based Newton-like iteration to optimize objective functions $f : \mathbb{R}^n \to \mathbb{R}$ that are exactly once continuously differentiable, such that the first derivative $F(x) \equiv \nabla f(x)$ is still composite piecewise differentiable. Another promising application is the modelling of pressure limits in gas or water networks as piecewise linear equations $\max(p(t), u) = u$. An important aim is the integration of our experimental code as robust implementation into Plan-C, which is a numerical piecewise linear algebra package for storing, solving and manipulating piecewise linear operators in C.

References


1 Problem Description

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm [7] is a widely used iterative method to solve the Navier-Stokes equations of Computational Fluid Dynamics (CFD). It decouples momentum from mass conservation (continuity) equations, thus allowing them to be solved independently. The momentum and continuity equations are discretized into linear equation systems, e.g. using a finite volume scheme. In the context of Algorithmic Differentiation (AD) it is desirable not to differentiate the solvers of those linear systems in a strictly black-box fashion, due to prohibitive memory and run time penalties, but to differentiate them symbolically [3]. Parallel propagation of adjoints can be implemented with the Adjoint Message Passing Interface [9] (AMPI), which acts as a wrapper around MPI and handles communication during the data flow reversal. Symbolically differentiated linear solvers are a challenge for adjoint parallel communication, as adjoint inter process communication in the solvers during the solution is lost and has to be recovered during data flow reversal. In this paper we discuss the process of data flow reversal in presence of sparse linear systems which arise from finite volume schemes in combination with distributed communication, reversed using the AMPI library. The described procedure is of course not only applicable to SIMPLE but can be applied virtually unchanged to other algorithms such as PISO (Pressure implicit with splitting of operator) for unsteady cases.

2 Symbolic Differentiation of Linear Solvers combined with AMPI

The SIMPLE algorithm involves the assembly of linear equation systems for the momentum and mass conservation equations. To establish a coupling between those equations several correction steps are applied. Those correction steps and the assembly of the equation systems are assumed to have a lower computational complexity and are hence treated algorithmically. The equation systems can be described as

\[ A_v \cdot V = B_v, \quad A_v \in \mathbb{R}^{n \times n}, V \in \mathbb{R}^{3 \times n}, B_v \in \mathbb{R}^{3 \times n} \]  

\[ A_p \cdot p = b_p, \quad A_p \in \mathbb{R}^{n \times n}, p \in \mathbb{R}^n, b_p \in \mathbb{R}^n, \]  

where \( n \) corresponds to the number of cells in the domain, \( V = [v_x, v_y, v_z] \) denotes the three dimensional velocity vector, \( p \) the pressure vector and \( B_v \) and \( b_p \) the corresponding source terms. As the finite volume coefficients for all velocity directions are identical only one matrix \( A_v \) is needed. The linear equation is then solved for all components and their associated source terms independently. Usually the matrix \( A_p \) is symmetric, while \( A_v \) is not (due to upwinding for the convection term). In the following we focus on the general case \( A x = b \) with regular \( A \in \mathbb{R}^{n \times n}, x \in \mathbb{R}^n \) and \( b \in \mathbb{R}^n \).

When calculating the solution to the linear system \( A x = b \) without applying AD the adjoints \( A(1) \) and \( b(1) \) are not calculated automatically (we use the notation from [6], where \( \bullet(1) \) denotes the first order adjoint). However the adjoints of the solution \( x(1) \) are known and can be used to obtain the symbolic adjoints [3]:

During primal solution:

\[ A \cdot x = b \quad \rightarrow \quad x := \text{solve}(A, b) \]  

During data flow reversal:

\[ A^T \cdot b(1) = x(1) \quad \rightarrow \quad b(1) := \text{solve}(A^T, x(1)) \]  

\[ A(1) := -x^T \cdot b(1). \]  

Usually the matrix \( A \) is stored in a sparse representation (e.g. compressed row storage or coordinate format). Thus the outer vector product in equation (5) forming \( A(1) \) needs only to be applied to the adjoints corresponding to nonzero entries.

*Corresponding Author, RWTH Aachen University, towara@stce.rwth-aachen.de
†RWTH Aachen University, naumann@stce.rwth-aachen.de
Note that this approach assumes that \( x \) is actually the solution to \( Ax = b \). When using iterative linear solvers, higher accuracy may be required during the primal evaluation in order to obtain the desired accuracy in the symbolic adjoint. Also the concept of relative tolerance (where one does not prescribe an absolute residual threshold, but a reduction of the residual in relation to the initial guess by a certain factor) should not be applied when differentiating the linear system symbolically. Our operator overloading tool \texttt{dco/c++} allows to stop the recording of data flow and required intermediate values (switch to passive mode) and restart the recording at a later time (switch to active mode). The resulting gap has to be filled when the adjoints are propagated from the outputs to the inputs during the interpretation phase. For this purpose \texttt{dco/c++} allows to create functions to be called at a specific time in the interpretation process using a callback interface.

For distributed CFD calculations often a ghost cell approach is used, where values of other processors are cached locally and updated on demand to reduce communication complexity. Normally that communication will be handled by AMPI. However if information is passed in the passive section during the linear solver calls the adjoint information has to be supplied in the callback function. Not all iterations of the SIMPLE algorithm have to be captured, instead reverse accumulation [1] can be utilized to obtain the gradient faster for steady cases. Alternatively piggy-backing [4] can be used to get a one-shot approach for optimization.

### 3 Application to Discrete Adjoint OpenFOAM

OpenFOAM [5] is an open source package of utilities geared at solving typical CFD-problems using the finite volume scheme. A discrete adjoint version of OpenFOAM was developed by our group. Prior work includes the introduction of checkpointing [10] and AMPI used without symbolically differentiated linear solvers [11]. OpenFOAM stores linear systems in the \texttt{ldu} sparse format (lower diagonal upper, a variant of the coordinate format), where only the values of nonzero coefficients and their respective row and column indices are stored. In the \texttt{lduMatrix} data structure coefficients on the diagonal correspond to the central coefficients in the finite volume scheme (see e.g. [2]), while coefficients in the upper and lower section correspond to the influence from the surrounding cells, interpolated to the faces of the cell. If a domain, and thus its system matrix, is decomposed onto several processors the upper and lower coefficients only capture the influence of the cells on the corresponding processor. Thus when the linear equation system is solved (iteratively), the missing information from the cells on other processors has to be supplied on the fly. The missing coefficients (vectors \( B \) and \( I \) in Algorithm 1) are factored in directly in the (sparse) matrix vector product after all the local entries have been multiplied. Every cell can have multiple neighboring processors. However every face can either be an internal face (thus connecting two cells on the same processor) or it lies on a processor patch (a patch is a named list of faces to which boundary conditions can be applied) which means that it connects two cells on two different processors. The values of cells adjacent to a processor patch are available via ghost cells which can be distributed on a per patch basis.

In Algorithm 1 we outline the implementation of the symbolic calculation of adjoints \( A^{(1)} \) and \( B^{(1)} \). The sparse storage and linear solvers exploit symmetry, storing and using only one triangular part of the matrix in the symmetric case. Thus the adjoints of the stored triangle must also include the influence of the omitted opposing triangular part.

### 4 Case Study

In the following we observe the flow over a rearward-facing step [8], which is often used as a benchmark case for OpenFOAM. The solver used is a variant of SIMPLE which includes a penalty term in the momentum equations thus allowing to be used in the context of topology optimization (see [10] for details). It calculates the gradient of pressure loss with respect to the penalty parameters which are defined at every cell of the domain. The k-\( \epsilon \) turbulence model is used to obtain a steady solution. The mesh consists of 0.8 million hexahedral cells.

Figure 1 shows the converged velocity field, with a recirculation area behind the step. Figure 2 depicts the sensitivity field. The highest sensitivity, thus the most important region for the pressure loss, is at the edge of the step. The region with negative sensitivity in the lower left corner would be filled by a topology optimization algorithm to lower the pressure loss.

We see a major reduction of tape memory when utilizing the symbolically differentiated linear solvers (here from 150 GB down to 40 GB). For less well conditioned problems which require more solver iterations the memory improvements are much higher, as the memory consumption with symbolically differentiated solvers is independent of the number of iterations. We also generally see a reduction in run time, due to less memory allocation, less interpretation and less adjoint communication which outweighs the need to solve additional linear equation systems during the data flow reversal. Both the run time and the memory consumption are dominated by the solution of the pressure equation, for which the geometric-algebraic multi-grid solver (GAMG) is used. The remaining equations are solved with a Gauss-Seidel solver variant. The run times for a modest amount of processors are plotted in Figure 3, showing that both the black box and the symbolic version scale. The run time factor between the passive evaluation of one SIMPLE iteration and its adjoint evaluation (recording + interpretation) ranges between 6 and 8 in this example.
Figure 1: Primal velocity field

Figure 2: Sensitivity field with zero contour (black)

**Input**: primal solution $\mathbf{x}$, incoming adjoints $\mathbf{x}_\text{(1)}$

**Data**: sparse matrix coefficients: $(\mathbf{l}, \mathbf{d}, \mathbf{u})$, corresponding lduAddressing $(\mathbf{L}, \mathbf{U})$, boundary coefficients $\mathbf{B}$

**Output**: adjoints $\mathbf{b}_\text{(1)}$, $(\mathbf{l}_\text{(1)}, \mathbf{d}_\text{(1)}, \mathbf{u}_\text{(1)})$, $\mathbf{B}_\text{(1)}$

forall the diagonal entries $d_i$ at index $i$ in $\mathbf{d}$ do

$\quad d_\text{(1)}_i \leftarrow d_\text{(1)}_i - b_\text{(1)}_i \cdot x_i$

end

forall the lower entries $l_i$ at index $i$ in $\mathbf{l}$ do

$\quad j \leftarrow U_i$
$\quad k \leftarrow L_i$
$\quad l_\text{(1)}_i \leftarrow l_\text{(1)}_i - b_\text{(1)}_{ij} \cdot x_k$

if $\mathbf{A}$ symmetric with no upper part then

$\quad l_\text{(1)}_i \leftarrow l_\text{(1)}_i - b_\text{(1)}_{ik} \cdot x_j$

end

end

forall the upper entries $u_i$ at index $i$ in $\mathbf{u}$ do

$\quad j \leftarrow L_i$
$\quad k \leftarrow U_i$
$\quad u_\text{(1)}_i \leftarrow u_\text{(1)}_i - b_\text{(1)}_{ij} \cdot x_k$

if $\mathbf{A}$ symmetric with no lower part then

$\quad u_\text{(1)}_i \leftarrow u_\text{(1)}_i - b_\text{(1)}_{ik} \cdot x_j$

end

end

forall the processor boundary fields $\mathbf{p}_j$ with index $j$ do

update boundary cells of $\mathbf{x}$ and $\mathbf{b}_\text{(1)}$ on patch $\mathbf{p}_j$;

$\quad \mathbf{x}^* \leftarrow$ boundary cell values of $\mathbf{x}$ from neighboring processor;
$\quad \mathbf{b}_\text{(1)}^* \leftarrow$ boundary cell values of $\mathbf{b}_\text{(1)}$ from this processor;

forall the faces $f_i$ with index $i$ on $\mathbf{p}_j$ do

$\quad \mathbf{B}_\text{(1)}_{ij} \leftarrow \mathbf{B}_\text{(1)}_{ij} + \mathbf{b}_\text{(1)}^* \cdot \mathbf{x}^*$

end

end

Algorithm 1: Calculation of the sparse entries of $\mathbf{A}_\text{(1)}$, $\mathbf{b}_\text{(1)}$ including processor boundary coefficients

5 Outlook

Following brief introductions to Discrete Adjoint OpenFOAM, dco/c++ and AMPI the talk will focus on the algorithmic challenges faced when applying adjoint AD to the MPI-parallel SIMPLE scheme. We describe the communication pattern and compare the various approaches in terms of their computational efficiencies. Further case studies as well as a higher degree of parallelism will be presented.
Figure 3: Run time of the recording of one SIMPLE step, the average run time needed for the interpretation of that step in the process of reverse accumulation and for reference the primal run time of one SIMPLE step

References


Enhancing sparsity of Hermite polynomial expansions with Automatic Differentiation

Mu Wang *, Guang Lin †, Alex Pothen ‡

1 Introduction

Uncertainty quantification (UQ) plays an important role in constructing computational models as it helps to understand the influence of uncertainties on the quantity of interest. Let \( \xi(\omega) \) be a \( d \)-dimensional random vector \( \xi(\omega) = (\xi_1(\omega), \xi_2(\omega), \ldots, \xi_d(\omega)) \), where \( \omega \) is an event in a complete probability space \( \Omega \), and for simplicity we denote \( \xi(\omega) \) as \( \xi \). Our goal is to approximate the quantity of interest \( u(\xi) \) with a generalized polynomial chaos (gPC) expansion [1]:

\[
    u(\xi) = \sum_{n=1}^{N} c_n \psi_n(\xi) + \epsilon(\xi),
\]

where \( \epsilon \) is the truncation error, \( N \) is a positive integer, \( c_n \) are coefficients, and \( \psi_n \) are multivariate polynomials which are orthonormal w.r.t the distribution of \( \xi \). Hence

\[
    \int_{\mathbb{R}^d} \psi_i(\xi) \psi_j(\xi) \rho(\xi) d\xi = \delta_{ij},
\]

where \( \rho(\xi) \) is the probability distribution function PDF of \( \xi \) and \( \delta_{ij} \) is the Kronecker delta. If \( u(\xi) \) is in the Hilbert space associated with the measure of \( \xi \), the approximation in Eq. 1 converges in the \( L_2 \) space as \( N \) increases. Theoretically, the coefficients \( c \) can be evaluated by solving the following linear system:

\[
    \Psi c = u + \epsilon,
\]

where \( u = (u^1, u^2, \ldots, u^M)^T \) is the vector of \( M \) output samples, \( \Psi \) is an \( M \times N \) matrix with \( \Psi = \phi_i(\xi^2) \), and \( \epsilon \) is a vector of error samples.

In many practical problems, it is costly to obtain samples \( u \), so due to the limited computational resources, we will often have \( M < N \) or even \( M \ll N \). Thus the linear system in Eq. 3 is under-determined. The compressive sensing method effectively solves Eq. 3 while keeping \( c \) sparse.

2 Enhance Sparsity

In this work, we consider the specific case where \( \xi \) is i.i.d Gaussian random variables, i.e., \( \xi \sim N(0, I) \). We aim to find a new set of i.i.d Gaussian random variables \( \eta \) via a mapping \( g : \mathbb{R}^d \rightarrow \mathbb{R}^d \) which maps \( \xi \) to \( \eta \) as \( \eta = g(\xi) \) such that the gPC expansion of \( u \) w.r.t \( \eta \) is sparser. In other words,

\[
    u(\xi) \approx \sum_{n=1}^{N} c_n \tilde{\psi}_n(\xi) = \sum_{n=1}^{N} \tilde{c}_n \tilde{\psi}_n(\eta(\xi)) \approx u(\eta(\xi)),
\]

where \( \tilde{\psi}_n \) are orthonormal polynomials associated with the new random vector \( \eta \) and \( \tilde{c}_n \) are the corresponding coefficients. Since we focus on the special case where \( \eta \) is also i.i.d Gaussian random variables from \( N(0, I) \), we have \( \tilde{\psi}_n = \psi_n \). Furthermore, we restrict the mapping \( g \) to be linear, \( \eta = g(\xi) = A \xi \), in which \( A \) is an orthogonal matrix to ensure \( \eta \) is i.i.d Gaussian, i.e., \( \eta \sim N(0, I) \).

In order to obtain the matrix \( A \), we adopt the active subspace approach [2]. We first define the “gradient matrix”:

\[
    G = E \left[ \nabla u(\xi) \cdot \nabla u(\xi)^T \right] = U A U^T, U U^T = I,
\]

where \( G \) is symmetric, \( \nabla u(\xi) \) is the gradient vector of \( u(\xi) \), and \( U A U^T \) is the eigendecomposition of \( G \). Then we choose \( A = U^T \) which as a unitary matrix defines a rotation in \( \mathbb{R}^d \) and the linear mapping \( g^{-1} \) projects \( \eta_i \) on the eigenvectors \( U_i \). Consequently, when the differences between \( |\lambda_i| \) are large, \( g \) helps to concentrate the dependence of

*Purdue University, wang970@purdue.edu
†Speaker, Purdue University, guanglin@purdue.edu
‡Purdue University, apothen@purdue.edu
u predominantly on the first few random variables ηi, which are known as the active subspace. And hence we can expect ˜c to be sparser than c.

To evaluate G, we apply Automatic Differentiation (AD) while collecting sample points uq = u(ξq), 1 ≤ q ≤ M. Each uq = u(ξq) can be considered as a scalar function, and its gradient vector ∇u(ξq) can be evaluated efficiently via reverse mode of AD [3] at a cost linear in the cost of collecting the sample u(ξq). Then G can be evaluated by averaging ∇u(ξq)∇u(ξq)T over all the samples, which is essentially a Monte Carlo method. The entire procedure is summarized in Algorithm 1. Further details of the algorithm are in [4].

Algorithm 1 Compressive sensing method with Automatic Differentiation

1: For a given random vector ξ and quantity of interest u, collect M samples of u1, u2, ..., uM.
2: While sampling, using AD to compute ∇u(ξ1), ∇u(ξ2), ..., ∇u(ξM).
3: Construct G = 1/M ∑Mq=1 ∇u(ξq) · ∇u(ξq)T. Then decompose G as G = UΛUT, UUT = I.
4: Define η = UTξ, and compute samples ηq = UTξq, 1 ≤ q ≤ M. Also, construct the new measurement matrix Ψ with Ψij = ψj(ηq).
5: Solve the optimization problem (P_h,c) in a suitable norm h to obtain c:
   \[
   \text{arg min}_c ||c||_h, \text{subject to } ||\Psi c - u|| \leq \epsilon.
   \]

3 Benefits of Introducing Automatic Differentiation

In previous work without Automatic Differentiation [4], an iterative approach is needed to compute the gPC expansion. The reason is that we have to approximate the “Gradient matrix” G as

\[
G = E\left[ \nabla \left( \sum_{n=1}^{N} (c_n \psi_n(\xi)) \right) \cdot \nabla \left( \sum_{n=1}^{N} (c_n \psi_n(\xi)) \right)^T \right].
\]  (6)

The approximation is based on the fact that \( \sum_{n=1}^{N} c_n \psi_n(\xi) \) approximates \( u(\xi) \). With the approximated G, we replace Steps 3-5 in Algorithm 1 by an iterative procedure. Hence we begin with coefficients \( c^{(0)} \), construct \( G^{(0)} \) according to Eq. 6, then follow Steps 4 and 5 to update the coefficients \( c^{(1)} \). This procedure is repeated until a termination condition is met. So the first benefit of introducing AD is that we can avoid the iterative procedure by directly evaluating \( G = E\left[ \nabla u(\xi) \cdot \nabla u(\xi)^T \right] \) from samples of u.

Another benefit of AD is it is more efficient than evaluating the approximation. Generally, the approximation in Eq. 6 is evaluated as

\[
G_{ij} = \sum_{n=1}^{N} \sum_{n'=1}^{N} c_n c_n' E \left[ \frac{\partial \psi_n(\xi)}{\partial \xi_i} \cdot \frac{\partial \psi_{n'}(\xi)}{\partial \xi_j} \right] = c^T K_{ij} c,
\]  (7)

where \( K_{ij} \) is a “stiffness” matrix with entries

\[
(K_{ij})_{kl} = E \left[ \frac{\partial \psi_k(\xi)}{\partial \xi_i} \cdot \frac{\partial \psi_l(\xi)}{\partial \xi_j} \right].
\]  (8)

The block matrix K can be precomputed since the functions \( \{\psi_i\} \) are normalized Hermite polynomials. However, each \( K_{ij} \) is an \( N \times N \) matrix since there are \( N \) basis functions. Since \( N \) could be very large, directly evaluating G from M samples using AD is usually more efficient than evaluating the “stiffness” matrix K since \( M \ll N \).

4 Numerical Results

We implemented Algorithm 1 in MATLAB and linked it with the AD tool ReverseAD [5] to evaluate the gradient \( \nabla u(\xi) \). We also implemented the iterative approach in [4]. We compare the effectiveness of these two approaches on two functions. The first one (F1) is with equally weighted random variables:

\[
u(\xi) = \frac{\sum_{i=1}^{d} \xi_i + 0.25 \left( \sum_{i=1}^{d} \xi_i^2 \right)^2 + 0.025 \left( \sum_{i=1}^{d} \xi_i \right)^2}{d}.
\]  (9)

The second one (F2) is similar but the random variables \( \xi_i \) have decreasing weights.

\[
u(\xi) = \frac{\sum_{i=1}^{d} \frac{\xi_i}{d} + 0.25 \left( \sum_{i=1}^{d} \frac{\xi_i}{\sqrt{i \times d}} \right)^2}{d}.
\]  (10)
These two cases have closed form expressions for the gradient. We use them here to show case the effectiveness of AD tools. One can expect that AD is the way to go in cases where the gradient does not (or not easily) have a closed form.

First, we set \( d = 12 \) and the maximum order of the Hermite polynomial \( P \) to be 3, hence the number of basis functions \( N \) is 455. We set a threshold \( \theta = 10^{-5} \), that any entry in \( c \) smaller than \( \theta \) is considered as zero. The results of the two different methods are summarized in Table 1.

From the results, firstly, both methods get almost the same results. That is, the quantity of interest \( u(\xi) \) generated by gPC has almost the same mean and variance as the test function. Secondly, we see that by applying AD the performance is greatly increased. We can expect the performance gap to increase when we have higher dimensions \( d \), because the cost of sampling \( G \) using AD growth linearly with \( d \). The number of basis functions \( N \), and thus the cost of evaluating the \( K_{ij} \), grows exponentially with \( d \). Finally if we count the number of nonzeros with the threshold \( \theta \), the coefficients in the gPC expansion generated by evaluating the gradient matrix via AD is sparser than that obtained from the iterative method.

In the second test, we increase the dimension \( d \) to 50 and keep \( P = 3 \), so the number of basis functions is 23426. The results on F1 and F2 are summarized in Table 2. As expected, the performance gap between using AD and not using AD is larger than the previous case. Moreover, Figure 1 gives the magnitude of the first 1000 gPC coefficients \( |c_n| \). We can observe that the coefficients computed by using AD is more sparse, while the coefficient compute by iterative method slowly convergence to zero. As a result using AD gives a better approximation of the \( u(\xi) \) as the mean and variance are closer to the real function.

![Figure 1: Magnitude of the first 1000 gPC coefficients |c_n| for F2.](image)

From previous results, we found that by using AD to accurately evaluate the gradient matrix \( G \), we can often reduce the dimension of the active subspace to just a few directions. This inspires to introduce a truncation trick into...
Table 3: Numerical results ($d = 1000$, $P = 3$) for gradient matrix evaluated by AD.

<table>
<thead>
<tr>
<th></th>
<th>Reference solution</th>
<th>Direct with AD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Var</td>
</tr>
<tr>
<td>F1</td>
<td>0.2577</td>
<td>9.6377</td>
</tr>
<tr>
<td>F2</td>
<td>0.0019</td>
<td>0.0010</td>
</tr>
</tbody>
</table>

Figure 2: Magnitude of the gPC coefficients for example KdV equation.

the algorithm. The idea is to set the dimension of $\eta$ as $k << d$ so that $\eta \in \mathbb{R}^k$ instead of $\mathbb{R}^d$. So that we can solve a problem with much more smaller dimensions. This enables us to solve problems with large dimensions. For example, we tested on F1 and F2 with $d = 1000$, $P = 3$ and $k = 12$. The result is as Figure 3. We see that the gPC expansion gives very close approximation of $u(\xi)$ even if we truncate the entire space of $\mathbb{R}^d$ into $\mathbb{R}^k$. And the runtime is much smaller than solving $d = 50$ in full dimension.

As an example application of our new method to a more complicated and nonlinear differential equation, we consider the Korteweg-de Vries (KdV) equation with time-dependent additive noise:

\[
\begin{aligned}
& u_t(x,t;\xi) - 6u(x,t;\xi) + u_{xxx}(x,t;\xi) = f(t;\xi), x \in (-\infty, \infty), \\
& u(x,0;\xi) = -2\text{sech}^2(x).
\end{aligned}
\] (11)

The details of the problem can be found in [4]. Here we show a preliminary result in Figure 2 that by using AD, the magnitude of the gPC coefficients are reduced by only leaving few dominate terms.

5 Conclusion

We show that by introducing Automatic Differentiation, we can transform an iterative solution procedure in a compressive sensing-based gPC method into a direct one, thus improving its performance. This approach has greater potential for high dimensional problems or when non-linear mappings $g$ are used, because Automatic Differentiation can precisely compute the gradient information of the quantify of interest. These will be the directions of future work.

References


Evaluating High Order Derivative Tensors in Reverse Mode of Automatic Differentiation

Mu Wang * and Alex Pothen†

1 Introduction and Background

Let $y = f(x)$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$ denote a scalar objective function of real values. Following the notations in [1], both here and elsewhere in the abstract, let $v_1, \ldots, v_0$ denote the independent variables ($x$), then the execution of the objective function can be decomposed as:

$$
\text{for } k = 1, 2, \ldots, l
v_k = \varphi_k(v_i)_{\{v_i \prec v_k\}},
$$

where each $v_k = \varphi_k(v_i)_{\{v_i \prec v_k\}}$ represents an elementary function, and $v_i \prec v_k$ denotes that variable $v_k$ directly depends on variable $v_i$. We call each $v_k = \varphi_k(v_i)_{\{v_i \prec v_k\}}$ a Single Assignment Code (SAC). Evaluating the objective function $f$ is thus equivalent to evaluating the SAC sequence. Here $l$ denotes the number of SACs the sequence contains, and so it represents the complexity of numerically evaluating the objective function $f$.

To evaluate the first order derivative, i.e., the gradient of the objective function, we can apply the first order chain rule on the SAC sequence. This yields two modes of AD as we can evaluate the derivatives in the same order as the SAC sequence is evaluated, or in the reverse order. The former yields the forward mode of AD and the latter yields the reverse mode of AD. The forward mode can be carried out simultaneously with the function evaluation. The reverse mode requires the initial evaluation of the SAC sequence to make a reverse sweep possible, and we call this set of values the trace of the function.

Derivatives other than first order can theoretically be evaluated by applying a hierarchy of forward and/or reverse modes. For example, the second order derivative (Hessian) can be evaluated by four such combinations: forward-over-forward, forward-over-reverse, reverse-over-forward and reverse-over-reverse. However, these four modes evaluate the Hessian in an indirect way: The forward-over-forward mode gives a single entry in the Hessian matrix, and the other three modes give a Hessian-vector product in a single run. Hence they lack the ability to fully exploit the symmetry available in the second order derivative during its evaluation. This limits the applicability of these approaches in evaluating derivatives higher than the second order.

A practical approach for high order derivatives is by interpolating the derivative through propagating a family of Taylor coefficients as described in [2]. This algorithm can be viewed as a high order forward mode. The complexity of this approach for evaluating the full derivative tensor of order $d$ is $O(d \cdot \binom{n+d}{d} \cdot l)$. In this expression, the binomial coefficient could be replaced by the number of nonzeros in the final derivative tensor if we have prior knowledge of the sparsity pattern of the derivative tensor. Another form of high order forward mode is developed in COSY Infinity [3]. By careful design of data structures, this algorithm gathers nonzero derivative terms when evaluating each SAC, and thus exploits the sparsity.

In this work, we propose a high order reverse mode algorithm of AD. The new algorithm is obtained by extending an invariant that we identified in the second order incremental reverse mode to higher orders [4]. Originally, the second order invariant was proposed to give a simpler derivation of the edge pushing algorithm of Gower and Mello [5]. Now we’ll show that this invariant can be extended to arbitrarily higher orders, and hence may be viewed as a fundamental concept for higher order reverse mode of AD.

2 Algorithm Details

The high order invariant in reverse mode of AD can be explicitly stated in terms of an equivalent function and a set of live variables (to be defined below).

Proposition 1 In the reverse mode of evaluating the derivative tensor of order $d$, after processing the SAC $v_k = \varphi_k(v_i)_{\{v_i \prec v_k\}}$, the intermediate results we maintain are all derivatives up to order $d$ of a suitably defined equivalent function $f_k(S_k)$, in which $f_k$ is obtained from the processed SAC sequence $\{\varphi_1, \varphi_{l-1}, \ldots, \varphi_k\}$, and the set of independent variables for $f_k$ is the live variable set $S_k$.

*Speaker, Purdue University, Department of Computer Science, West Lafayette IN 47907 USA, wang970@purdue.edu
†Purdue University, Department of Computer Science, West Lafayette IN 47907 USA, apothen@purdue.edu
In this proposition, a live variable is a concept we borrow from data-flow analysis in compiler theory. In this analysis, at a certain point of the function evaluation, a variable is considered as live if it holds a value that might be used in the future. Hence liveness analysis is a “backward-may” analysis. In reverse mode, since the trace only contains the information of a certain execution path through the program, we can use a stricter definition on the live variable. A variable is live if it holds a value that will be used in the future. We can explicitly state how the live variable set changes during the reverse mode after processing the SAC $v_k = \varphi_{k}(v_i)_{\{v_i : v_i < v_k\}}$ by the following equation.

$$S_k = S_{k+1} \setminus \{v_k\} \cup \{v_i : v_i < v_k\}. \quad (1)$$

In the reverse mode, the initial live variable is the dependent variable, and the final live variable set is the set of independent variables.

Now we consider the equivalent function $f_k(S_k)$. Initially in the reverse mode, $k = l$, and this function is $f_{l+1}(v_l) = v_l$. Now we can derive $f_k(S_k)$ from $f_{k+1}(S_{k+1})$ by treating $v_k$ as an independent variable in $f_{k+1}(S_{k+1})$ (so $v_k$ belongs to $S_{k+1}$). Now $f_k(S_k)$ is a composite function obtained by replacing $v_k$ in $f_{k+1}(S_{k+1})$ using $\varphi_{k}(v_i)_{\{v_i : v_i < v_k\}}$ (so we have $v_i \in S_k, v_i < v_k$ and $v_k \notin S_k$). Then the high order reverse mode is a transcript to obtain derivatives of $f_k(S_k)$ from $f_{k+1}(S_{k+1})$ by viewing $v_k$ in $f_{k+1}(S_{k+1})$ as an inner function defined by $v_k = \varphi_{k}(v_i)_{\{v_i : v_i < v_k\}}$.

With this invariant in place, we can write a general expression for a derivative of arbitrary order. By $\mathcal{P}(D)$ we denote the power multisets of a multiset $D$, and a $k$-partition of a set is a partition into $k$ subsets.

**Proposition 2** For a multiset (distinct members are allowed to appear more than once) $D = \{v_1, v_2, \ldots, v_{|D|}\}$, define a partial operator as $\frac{\partial^{(k)}}{\partial D} = \frac{\partial^{(k)}}{\partial v_2 \ldots \partial v_{|D|}}$. Then for the composite function defined as $f_k(S_k) = f_{k+1}(S_{k+1} \setminus \{v_k\}, v_k = \varphi_{k}(v_i)_{\{v_i : v_i < v_k\}})$, the derivative $\frac{\partial^{(k)} f_k(S_k)}{\partial D}$ is given by $\mathcal{D}$:

$$\frac{\partial^{(D)} f_k(S_k)}{\partial D} = \frac{\partial^{(D)} f_{k+1}(S_{k+1})}{\partial D} + \sum_{|D|}^{t} \sum_{Z \in \mathcal{P}(D)} \left( \sum_{Z = t}^{Z = t} \frac{\partial^{(D)} f_{k+1}|\varphi_{k}}{\partial D_{t-z}} \frac{\partial^{(D)} f_{k+1}(S_{k+1})}{\partial Z^{\partial^{z} v_k}} \right). \quad (2)$$

An equivalent form is

$$\frac{\partial^{(D)} f_k(S_k)}{\partial D} = \frac{\partial^{(D)} f_{k+1}(S_{k+1})}{\partial D} + \sum_{|D|}^{t} \sum_{Z \in \mathcal{P}(D)} \left( \sum_{Z = t}^{Z = t} \frac{\partial^{(D)} f_{k+1}|\varphi_{k}}{\partial D_{t-z}} \right) \frac{\partial^{(D)} f_{k+1}(S_{k+1})}{\partial Z^{\partial^{z} v_k}} \right. \quad (3)$$

In Eq.3 intuitively, we can claim that the value of $\frac{\partial^{(D)} f_k(S_k)}{\partial D}$ is the summation of two parts. The first part is $\frac{\partial^{(D)} f_{k+1}(S_{k+1})}{\partial D}$ which can be considered as the derivatives inherited from $f_{k+1}(S_{k+1})$ without considering the composite part $v_k = \varphi_{k}(v_i)_{\{v_i : v_i < v_k\}}$. The second part can be considered as a weighted summation of the derivatives of $f_{k+1}(S_{k+1})$ where the composite part is considered. Each term in the summation, is a product of $\frac{\partial^{(D)} f_k(S_k)}{\partial \partial^{z} v_k}$, a derivative term of $f_k(S_k)$ involves $v_k$, and a weight term given by $\frac{\partial^{(D)} f_{k+1}|\varphi_{k}}{\partial D_{t-z}}$, i.e., a product of derivatives of the elementary function $\varphi_{k}$.

With Eq.3 we can directly write down a non-incremental version of the high order reverse mode algorithm (omitted here). The “non-incremental” comes from the behavior that we evaluating the value of the entire right-hand-side on Eq.3 for each $D$, where $D = \{v_1, v_2, \ldots, v_{|D|}\}, v_i \in S_k$. This non-incremental version of the high order reverse mode is not optimal in either efficiency or complexity. Hence we developed an incremental version of the algorithm that exploits both sparsity and symmetry.

Due to space limitations, we can only briefly introduce the concepts we use in deriving the incremental version of the reverse mode algorithm. Exploiting sparsity means we only perform updates with nonzero values in the algorithm. To do that, we break the evaluation of the right-hand-side of Eq.3 into individual terms. And we perform an “incremental” update only for each non-zero term. Exploiting symmetry has two aspects. The first aspect is that the high order derivative tensor is highly symmetric, and so we need to keep unique elements For example, only the lower half of the Hessian matrix is kept in the second order case. The second aspect is that since $D$ is a multiset, there might be many duplicated terms when evaluating all the right-hand-side terms of Eq.3. We gather the duplicate terms by evaluating its value only once and multiply the value by the number of copies, which can be computed by combinatorial mathematics. The complexity of the incremental reverse mode for evaluating the derivative of order $d$ is given by $O(B_d \cdot s^{d-1} \cdot l)$, where $s$ is the maximum size of all live variable sets $S_k$, and $B_d$ is the ($d+1$)-th Bell number. The final algorithm is as shown in Figure4.

We have implemented the algorithm in C++ in the code ReverseAD and have made it publicly available at https://github.com/wangmu0701/ReverseAD. In the implementation, we adopt a monotonic indexing scheme for the function trace in order to best suit the features of the algorithm. We use a code generator for small values of $d$ to generate flat code to maximize the efficiency and avoid some redundant coefficient computations.

197
3 Performance Evaluation

To evaluate the performance, we designed a synthetic function which has three parameters: the number of independent variables \( n \); the size of live variable set \( s \); and the number of SACs \( l \), which represents the complexity of the function. All three parameters can be set independently. The mathematical form of the synthetic function is:

\[
y = \prod_{i=1}^{n} t_i, \\
t_i = ID_k \circ \cdots \circ ID_1(z_i), \\
z_i = t, \\
t = \sum_{i=1}^{n} x_i.
\]

Hence by varying \( k \), we vary the complexity of the function \( l \). And each \( ID_j \) is an identity function randomly chosen among six predefined forms to make the SAC sequence to cover a variety of elementary functions.

We compared our high order reverse mode algorithm, ReverseAD, with Taylor series methods. For the Taylor series approach, we tried ADOL-C and Rapsodia, both of which implement the Taylor series algorithm with different design philosophies. In ADOL-C, the Taylor series is evaluated after the objective function is computed. So an overloaded approach, we tried among six predefined forms to make the SAC sequence to cover a variety of elementary functions. Hence by varying \( k \), the SAC sequence to cover a variety of elementary functions.

We found that Rapsodia is the fastest of the three algorithms when the number of independent variables is smaller than 20. But it does not scale up to more independent variables in either time or memory. Since we experiment with larger numbers of independent variables, we exclude Rapsodia from the reported results. In Figure 2 we show our results for computing the third and fourth order derivative of the synthetic test function where the complexity of the objective function \( l \) is kept as a constant, while the number of independent variables \( n \) and the size of live variable set \( s \) are changing simultaneously. It can be seen that ReverseAD is more efficient than the Taylor series approach due to its lower time complexity.

\[
ID(z) = \begin{cases} 
\sqrt{z + z}, & z \neq 0 \\
2.0 + z - 2.0, & z = 0 \\
\log(\exp(z)), & 1.0/(1.0/z), \\
sin(\sin(z)). & \end{cases}
\]

Figure 1: Incremental high order reverse mode.
Figure 2: The runtime on the synthetic function, in which we vary the number of independent variables and the size of live variable set simultaneously. Note that the run times are plotted in a logarithmic scale, and the tables show the run times.

Objective Function \( y = \varphi_l \circ \varphi_{l-1} \circ \cdots \circ \varphi_1(x) \)

First order chain rule for: \( g_k = \varphi_k \circ g_{k-1} \)

First Order Forward

High order chain rule for: \( g_k = \varphi_k \circ g_{k-1} \)

High Order Forward

First order chain rule for: \( f_k = f_{k+1} \circ \varphi_k \)

First Order Reverse

High order chain rule for: \( f_k = f_{k+1} \circ \varphi_k \)

High Order Reverse

Figure 3: Relationship between different AD algorithms.

the derivatives of \( g_k = \varphi_k \circ g_{k-1} \) in each step. A reverse mode AD algorithm begins with \( f_{l+1}(y) = y \), and evaluates the derivatives of \( f_k = f_{k+1} \circ \varphi_k \) in each step. In both modes, the computation at a step applies the chain rule, for any order derivative.

Moreover, in the forward mode, when evaluating the derivatives of \( g_k = \varphi_k \circ g_{k-1} \), the chain rule exploits the fact that the outer function \( \varphi_k \) is elementary. As each \( g_k \) is a function w.r.t the independent variables \( x \), the complexity of a forward mode algorithm is determined by the number of independent variables \( n \), the highest order of evaluated derivative \( d \), and the number of SACs \( l \). On the other hand, in reverse mode, when evaluating the derivatives of \( f_k = f_{k+1} \circ \varphi_k \), the chain rule (as in Prop. 2) is in a form that can exploit the fact that the inner function \( \varphi_k \) is elementary. As we have shown, each \( f_k \) is a function w.r.t the live variable set at that point, so the complexity in each step of reverse mode does not depend on the number of independent variables \( n \), but on the size of live variable set \( s \).

With this complete hierarchy of AD algorithms, we can also build cross-country modes by combining them. For example, one can apply a first order forward mode on top of a second order reverse mode to get a directional third order derivative, the result being a matrix which is a third order tensor-vector product.

References


Performance Evaluation of Automatic Differentiation Algorithms for Hessian Computation

Mu Wang¹, Assefaw Gebremedhin², and Alex Pothen¹

¹Department of Computer Science, Purdue University
²School of Electrical Engineering and Computer Science, Washington State University

Emails: wang970@purdue.edu, assefaw@eecs.wsu.edu, apothen@purdue.edu

Introduction

The second order derivative of a scalar function, i.e., the Hessian matrix, plays an important role in many applications. Several Automatic Differentiation (AD)-based Hessian evaluation algorithms have been proposed and implemented in the literature. While the high-level difference between the algorithms is generally understood, the detailed behavior of the algorithms and their performance characterization with respect to the structure and complexity of the computer code representing the function to be differentiated has not been systematically studied and understood. In this work, we design a synthetic function that allows us to change the structural properties of the function by tuning parameters, and we use the designed function to study the performance of four different Hessian computation algorithms and the sensitivity of the performance to variations in those structural properties.

Algorithms

We study three different approaches for Hessian evaluation. In all four algorithms, the sparsity available in the final Hessian is exploited; that is, computing with and storing zero entries is avoided. All four of the algorithms compared are implemented within the operator overloading paradigm.

(i) Compression-based methods The first two methods we study are based on the forward-over-reverse mode, or Second-Order-Adjoint-Mode (SOAM), of Automatic Differentiation [5, 6]. Each run of a forward-over-reverse mode gives a Hessian-vector product \(Hv\). Thus a trivial approach to get the entire Hessian matrix is to run the forward-over-reverse mode \(n\) times, where \(n\) is the number of columns in the Hessian, so that each run evaluates the \(i\)-th column of the Hessian, \(He_i\). This approach is clearly not efficient when the Hessian is sparse. Therefore a compression-based method is used instead [3].

The idea behind the compression-based approach is to determine an \(n\) by \(p\) seed matrix \(S\) that encodes the set of columns of the Hessian that can be “compressed” together and evaluated via one Hessian-vector product. The seed matrix is typically obtained by coloring the graph associated with the sparsity pattern of the Hessian. The number of columns \(p\) in the seed matrix is exactly the number of colors needed. In this manner, only \(p\) Hessian-vector products, instead of \(n \gg p\) Hessian-vector products, is needed to obtain a compressed version of the Hessian. The desired Hessian is then obtained via a suitable recovery step.

Depending on how the recovery step is performed, two different kinds of coloring are needed in the determination of the seed matrix. If the recovery is direct (no further arithmetic is involved in obtaining entries of the original Hessian), \(star\) coloring is the needed coloring; and if the recovery is indirect (entries of the original Hessian are obtained via substitutions), \(acyclic\) coloring is needed. In what follows, we refer to these two compression-based approaches as Direct and Indirect, respectively. We use the implementations in ADOL-C [7] and ColPack [2], where ADOL-C has the routines corresponding to sparsity pattern detection and computation of the compressed Hessian, and ColPack has the routines corresponding to coloring of the adjacency graph and recovery of the Hessian from the compressed representation.

(ii) Taylor coefficient based method The third Hessian evaluation method we study corresponds to evaluating the Hessian by propagating a family of second order Taylor coefficients [4]. By assigning pre-calculated initial values for each Taylor series, each entry in the Hessian matrix can be determined by a linear combination of the final Taylor coefficients. Each initial value is called a direction. The complexity of evaluating of the final Hessian using this method
is $O(n^2 \cdot \text{eval}(f))$, where $\text{eval}(f)$ is the temporal complexity of evaluating the objective function. The complexity can be reduced to $O(nnz \cdot \text{eval}(f))$, where $nnz$ is the number of non-zeros in the final Hessian matrix, if the sparsity pattern is known a priori. We use the implementation in \texttt{Rapsodia} \cite{1} and provide a patch which allows \texttt{Rapsodia} to only evaluate the non-zero entries in the Hessian matrix for a given sparsity pattern. To obtain the latter information, we use the sparsity pattern computed by \texttt{ADOL-C}. The design philosophy of \texttt{Rapsodia} is to use flat code to evaluate the Taylor coefficients for all directions. Given the number of directions, \texttt{Rapsodia} explicitly generates this flat code. As we will show, when the number of directions is large, the generated code is very large and compilation takes huge amounts of time (several hours even for a moderate size problem), making the method infeasible.

(iii) Live variable based method The last method we study is second order reverse mode based on the \textit{live variables} approach \cite{8}. The key idea in the algorithm is to extend a useful invariant in reverse mode AD into the second order. In particular, the insight is that, at each point, the intermediate state of the reverse mode algorithm is the derivative of an equivalent function with respect to the live variable set at that point. The equivalent function is defined by the single assignment codes processed in earlier steps. The complexity of this method is $O(s \cdot \text{eval}(f))$, where $s$ is the maximum size of live variable sets during the evaluation of the function $f$ \cite{8}. We implement this algorithm in a tool we called \texttt{ReverseAD} and use this implementation in the comparison.

Design of test cases

We construct testcases specially designed to enable in-depth empirical analysis of \textit{structural factors} that determine the performance of the different Hessian algorithms. To construct the testcases, we wrote a simple code that works as follows.

Suppose we have $n$ independent variables $x_i, 1 \leq i \leq n$, and let $y = f(x)$ be the dependent variable. Let the average number of nonzero columns per row in the Hessian $H = y''$ be $\rho$, and fixed. For each row $i, 1 \leq i \leq n$, we randomly pick $\frac{n}{2}$ column indices $r_j$ where each location $H(i, r_j)$ will be made nonzero. To achieve this, we make $s$ copies of $x_i$ (as $x_{ij}, 1 \leq j \leq s$) and manipulate them in such a way that the size of live variable sets during the function evaluation is at least $s$ for most of the time. We also let $x_i$ pass through $k$ compositions of identity functions—transformations that involve computation without changing value—to make it possible for the test function $y$ to cover a variety of kinds of operators and elementary functions. This also provides us with a way to control the complexity of the function evaluation $\text{eval}(f)$. Each identity function is randomly chosen from six pre-determined forms.

All random numbers are generated by a deterministic random number generator. Thus we can reproduce the function evaluations by giving the same initial seed.

With this setting, we can control the number of rows in the Hessian matrix ($n$), the density/sparsity of the Hessian ($\rho$), the complexity of the function evaluation ($k$), and the size of live variable sets during the function evaluation ($s$).

Details of The Synthetic Function The synthetic function has the following mathematical form:

$$
\begin{align*}
    y &= \sum_{i=1}^{n} z_i \cdot t_i \\
    z_i &= ID_k \circ \cdots \circ ID_1(x_i) \\
    t_i &= \sum_{j=1}^{\rho/2} x_{r_j} + \sum_{j=1}^{n} x_{ij}
\end{align*}
$$

$$
\text{ID}(w) =
\begin{cases}
    \sqrt{w \cdot w}, & \\
    2.0 + w - 2.0, & \\
    w \cdot 2.0 + 0.5, & \\
    \log(\exp(w)), & \\
    1.0/(1.0/w), & \\
    \sin(\text{asin}(w)).
\end{cases}
$$

The display at the right shows the six identity functions $\text{ID}(w)$ we used; the reader can verify that each of these functions returns as output what it is given as input. The variables $x_{ij}$ are the $s$ duplicated variables for $x_i$, designed to make the size of the live variable set at least equal to $s$. By varying $k$, we correspondingly vary the complexity of the function $\text{eval}(f)$ while keeping the function value unchanged.

Results

All the tests for the results reported here are performed on a Quad Core 2.5 GHz Intel I5-2400S processor with 8 GB memory and the code is compiled with \texttt{gcc/g++ 4.8.2}.

(a) General Runtime We first use some small tests to get a general sense of the performance of the algorithms. We set $\rho = 6$, $k = 30$ and $s = 30$. Table 1 lists the execution times (in seconds) we obtained when we run the four algorithms with different values of $n$. The table also lists the runtime for evaluating the input function $f$, both in its plain form and when taped (overloaded). One can see in both \texttt{ADOL-C} and \texttt{ReverseAD}, the time needed for evaluating
the objective function with the overloaded type is roughly about 10 times the runtime for plain function evaluation. For Rapsodia, the evaluation of derivative happens at the same time as evaluation of the function, so we cannot separate them.

Regarding the performance of the Hessian evaluations, the main observation here is that Rapsodia takes much longer time compared with the other three methods. Furthermore, it takes hours to compile the “flat-code” when there are more than 10,000 directions. Therefore, we exclude this method from further tests we report on.

<table>
<thead>
<tr>
<th>Function</th>
<th>Compression</th>
<th>Taylor coeff.</th>
<th>Live variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>#nnz in H</td>
<td>eval(f)</td>
<td>eval(f)</td>
</tr>
<tr>
<td>2,000</td>
<td>7,990</td>
<td>0.003</td>
<td>0.020</td>
</tr>
<tr>
<td>3,000</td>
<td>11,989</td>
<td>0.003</td>
<td>0.029</td>
</tr>
</tbody>
</table>

Table 1: Runtime (in seconds) for Direct, Indirect, Rapsodia and ReverseAD. The methods Direct and Indirect use functionalities implemented in the toolkit composed of ADOL-C and ColPack.

(b) Function Complexity  Our goal here is to see how the performance of Hessian evaluation changes as the function complexity (eval(f), gauged by $k$) is varied. We set $n = 20,000$, $\rho = 6$, $s = 30$, and vary $k$ from 20 to 60. The results we obtained are summarized in Table 2. We found that the runtime of Direct and Indirect increases super-linearly with increase in the function complexity. Further, we found that more than 90% (or 99% in the last two tests) of the time is spent in the sparsity pattern detection phase, see Figure 1. It is likely that our synthetic function is among the “worst-case” instances for the current sparsity pattern detection procedure implemented in ADOL-C. Finally, we see that the runtime of ReverseAD is within the same order of the function complexity.

<table>
<thead>
<tr>
<th>k</th>
<th>Direct</th>
<th>Indirect</th>
<th>ReverseAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>9.325</td>
<td>7.138</td>
<td>6.367</td>
</tr>
<tr>
<td>30</td>
<td>25.268</td>
<td>22.007</td>
<td>8.220</td>
</tr>
<tr>
<td>40</td>
<td>181.435</td>
<td>177.539</td>
<td>9.824</td>
</tr>
<tr>
<td>50</td>
<td>1765.766</td>
<td>1760.754</td>
<td>11.763</td>
</tr>
<tr>
<td>60</td>
<td>14114.265</td>
<td>14124.760</td>
<td>13.242</td>
</tr>
</tbody>
</table>

Table 2: Runtime (in seconds) for Direct, Indirect and ReverseAD with different function complexity.

(c) Size of Live Variables  Here we want to see how the performance changes as we vary the size of the live variable set. We fix $n = 20,000$, $\rho = 6$, $k = 30$ and vary $s$ from 20 to 60. The results, summarized in Table 3, suggest that performance of ReverseAD is much more sensitive to the size of live variable set than the other two algorithms. This is in accordance with our expectation and the complexity analysis of the algorithm [8].

<table>
<thead>
<tr>
<th>s = 20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>23.653</td>
<td>25.138</td>
<td>26.484</td>
<td>27.666</td>
</tr>
<tr>
<td>Indirect</td>
<td>21.071</td>
<td>22.310</td>
<td>23.110</td>
<td>23.843</td>
</tr>
</tbody>
</table>

Table 3: Runtime (in seconds) for Direct, Indirect and ReverseAD with different size of live variables.

(d) Sparsity/Density  Here we fix $s$ to be 30, and perform tests on three sets of settings: $n = 15,000$, $\rho = 8$, $k = 40$; $n = 20,000$, $\rho = 6$, $k = 30$; and $n = 30,000$, $\rho = 4$, $k = 20$. We choose those parameters so that the complexity of the function evaluation and the total number of non-zeros in the final Hessian matrix are roughly the same across the different settings. Table 4 lists the runtime results we obtained. We see that the runtime of Direct and Indirect decreases as the density of the Hessian matrix decreases (sparsity increases), which is in agreement with expectations. Here again, the lion’s share of the difference in runtime as density/sparsity varies is taken by the sparsity pattern detection step, as the breakdown in Figure 2 shows. Finally, we see that ReverseAD takes more time when the density of the Hessian matrix increases, but the growth rate is relatively small.
Table 4: Runtime (in seconds) for Direct, Indirect and ReverseAD with different sparsity/density.

<table>
<thead>
<tr>
<th></th>
<th>n = 15,000</th>
<th>20,000</th>
<th>30,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>139.841</td>
<td>25.745</td>
<td>12.833</td>
</tr>
<tr>
<td>Indirect</td>
<td>134.210</td>
<td>22.232</td>
<td>9.664</td>
</tr>
<tr>
<td>ReverseAD</td>
<td>7.616</td>
<td>8.634</td>
<td>9.745</td>
</tr>
</tbody>
</table>

Figure 2: Percentile breakdown for Direct and Indirect with different sparsity/density.

Conclusion

We compared the performance of four different Hessian evaluating algorithms under various systematically constructed input scenarios. We found that the approach in Rapsodia is feasible only when evaluating derivatives for function with a few number of independent variables. We also found that the bottleneck in the compression based methods is mainly the sparsity pattern detection step. The ReverseAD approach is the more stable and efficient approach in most cases.

References


Abstracts of Poster Presentations

Abstracts are presented alphabetically by the surname of the first author. If an author is first author on multiple abstracts then alphabetical order of the first significant word of the title is used to order such papers.
A Case Study of ADOL-C and CoDiPack applied to the Ice Sheet System Model

Tim Albring∗ Christian Bischof† Nicolas R. Gauger∗ Alexander Graf†
Alexander Hück‡ Benjamin Jurgelucks§ Max Sagebaum∗ Jean Utke¶

March 2016

1 Introduction

The Ice Sheet System Model (ISSM, [1]) is an ice flow modeling software developed by NASA/JPL. It has a C++ code base of about 95,000 lines of code with data pre- and post-processing being handled in Matlab. The computation of large scale gradients for, e.g., sensitivity studies is achieved by employing the operator overloading AD-tool ADOL-C [2]. The adaption of the code base for the gradient computations took place over an extended period of time from 2011 on. All major parts of the gradient computation were considered fully operational in 2014 [3]. Noteworthy changes to the code base to enable algorithmic differentiation included:

- Separation between active and passive (with respect to differentiation) data types using alias definitions called IssmDouble (for overloading with ADOL-C) and IssmPDouble (built-in double), respectively,
- Refactoring of the code to introduce a thin wrapper layer for MPI to allow for serial or parallel model computation, and serial or parallel gradient computation with the same code base, and
- Treatment of calls to (direct) linear solvers with special wrappers to exploit this numerical paradigm for the most efficient gradient computation.

In the past few years, however, several new operator overloading-based user-defined AD classes have been developed promising performance improvements. One of such AD implementation is CoDiPack [4] which includes the following main features:

- Forward and Reverse mode derivative computation using operator overloading with expression templates [5],
- AdjointMPI (AMPI) interface [6], and
- External function interface (e.g., for calls to linear solvers).

CoDiPack has been successfully used in the CFD software package SU2 [4,7] and other smaller software implementations. We are interested in further exploring aspects such as performance, feature completeness and interface design of CoDiPack, and to this end, we believe ISSM to be a good testbed. The existing code base enhanced with ADOL-C provides a validated AD implementation, that can be used to compare result. In particular, we want to investigate the following issues:

1. What steps are necessary to replace ADOL-C with CoDiPack? Here, one particular consideration is the aforementioned external function interface which is exposed through different APIs.

2. Do AD implementations of ADOL-C and CoDiPack reproduce the same results within roundoff error? As ADOL-C and CoDiPack use different implementation strategies and intrinsic implementations, the obvious answer “yes” may not be so obvious after all.

3. How does CoDiPack compare to ADOL-C, regarding both runtime and memory consumption? We believe that a thorough investigation of the different AD-enhanced versions of ISSM can lead to new insights and suggestions for improvements for both AD tools.

∗Scientific Computing, TU Kaiserslautern, Germany
†Scientific Computing, TU Darmstadt, Germany, to present
‡Corresponding Author, Scientific Computing, TU Darmstadt, Germany, alexander.hueck@sc.tu-darmstadt.de
§Institute of Mathematics, Paderborn University, Germany
¶Quantitative Research & Analytics, Allstate Insurance Company, USA
2 Aspects of a Type Change in ISSM

Algorithmic Differentiation via operator overloading is introduced to ISSM by changing the alias for the active type `IssmDouble`. The ISSM code base uses several abstraction mechanisms that have been introduced for dealing specifically with ADOL-C, e.g., casting operations on active types where the underlying data type is exposed through the ADOL-C specific API. Thus, most of the necessary changes to go from ADOL-C to CoDiPack are straightforward and contained to few code locations.

We have already begun to modify ISSM to make it compatible with CoDiPack. ISSM has a modular design where desired features (such as AD with ADOL-C) are selected with conditional preprocessor flags, e.g., specific code that is conditionally included when ADOL-C is used. We are pursuing this same approach for CoDiPack. So far, we have made the following modifications to the code base:

- The build system of ISSM (Autotools) was modified to support CoDiPack.
- The alias declaration of `IssmDouble` to CoDiPack’s AD type was added, based on the compile time settings.
- The preprocessor flags to support CoDiPack for the conditional compilation of parts of the code were added.
- ADOL-C partially uses C-style overloaded mathematical functions, i.e., `fmin` or `fmax`. CoDiPack provides `min` and `max`, on the other hand. To minimize changes to the ISSM code, we decided to add additional overloads for these functions to CoDiPack.

Currently, we are working on the compatibility of external function calls in ISSM with CoDiPack. ISSM can make use of different solver implementations, depending on compile time flags, namely the GSL (GNU Scientific Library) solver for dense matrices and MUMPS (a Multifrontal Massively Parallel sparse direct Solver) for sparse computations. As a result, both of these solvers need to be integrated using the external function interface of the AD tool. Thus, code locations where the external function interface of ADOL-C is used will require more sophisticated refactoring as the two tools differ in their respective implementation.

In the ADOL-C context the fundamental design decision to structure the execution trace into an operation, location, and floating point value part (so as to retain a simple persistent representation of the trace) imposes constraints on the external function interface. In particular, all external function call parameters have to be written to either the location (integers) or the floating point value part. This means that for the solver wrapping the system matrix and the right-hand side have to be packed into a floating point value sequence. Likewise, any sparse system matrix representation must have the non-zero indices packed into an integer sequence. To save space on the ADOL-C locations it is highly desirable to store the floating point sequence as an array with contiguous ADOL-C locations such that only the start location of the sequence has to be recorded. This packing is cumbersome, error prone, and imposes an overhead on the ADOL-C location management.

In contrast, CoDiPack allows full access to the internals of the trace and thus offers direct visibility of external function parameters. Arbitrary data can be stored for the external function, but the user has to take care that all updates are performed correctly. Therefore, the user needs more knowledge about the internals of CoDiPack but saves on the packing overhead identified for ADOL-C and therefore this may yield a considerable performance advantage that grows with the problem size. Clearly, this is a usability as much as a performance issue. The MUMPS solver integration is particularly important to lessen the impact of the solver itself on the overall computational time and to better differentiate between the performance characteristics of ADOL-C and CoDiPack.

The ability to scale the problem up to a realistic size is only possible with MPI parallelization. To that end, ADOL-C and CoDiPack both support AD for MPI communication but they use different library implementations, called AdjoinableMPI [8] and AdjointMPI [6], respectively. Thus, the MPI wrapper in ISSM needs to be modified. We believe that AdjoinableMPI can easily be exchanged with AdjointMPI because of the common interface that both adjoint MPI tools provide.

3 Outlook

This work, which we will have concluded by the time of the AD2016 conference, will shed light on both usability and performance aspects of different AD tool designs. Similar to ISSM/ADOL-C, we expect our new ISSM/CoDiPack version to support both dense and sparse solvers, in both serial and parallel computations. With these codes, we will conduct a thorough study, comparing the runtime and memory performance of both AD tools for serial and parallel gradient computations. We also believe that insights gained in switching AD tools, and in particular a detailed consideration of the hookup of external functions, will be of value for the enhanced usability of AD tools, as well as the general issue of providing a better encapsulation of AD coding issues in the AD augmentation of sophisticated simulation tools.

4 Acknowledgement

We thank Eric Larour of the NASA Jet Propulsion Laboratory for providing us with access to the source code and valuable guidance.
References


Parallel Algorithmic Differentiation in OpenModelica

Bernhard Bachmann, Willi Braun, Martin Flehmig, Rüdiger Franke, Leo Gall, Benjamin Jurgelucks*, Kshitij Kulshreshtha, Kilian Link, Ulf Markwardt, Vitalij Ruge, Andrea Walther, and Niklas Worschech

1 Institute for Mathematics, Paderborn University
2 Center for Information Services and High Performance Computing (ZIH), TU Dresden
3 Applied Mathematical Modeling & Optimization, Bielefeld University of Applied Sciences
4 Siemens AG, Power and Gas
5 ABB AG, Process Optimization Technology
6 LTX Simulation GmbH
7 Bosch Rexroth AG

June 2016

1 Motivation

Electrical power generation is more and more shifting from classically large power plants to many distributed small power producers using a large variety of technologies like solar power, wind energy, biogas, water powered turbines or energy storage systems. These distributed power generation systems together with conventional power plants may be interpreted as a set of combined virtual power plants which, on their own, need real-time optimization for efficient and economic operation. For the simulation of such virtual power plants very large heterogeneous models with many variables are needed.

2 OpenModelica

Many large physical models can be described with the modeling language Modelica. OpenModelica is an open source tool developed by the Open Source Modelica Consortium to simulate and optimize models using the Modelica language. OpenModelica parses and compiles the models given and creates C and C++ code via either the C interface or the C++ interface, respectively. The C/C++ code can then be compiled and the generated executable can be run in order to obtain the simulation results. In the past, OpenModelica was modified to additionally generate ADOL-C instrumented C++ code to model many variables of type adouble and thereby enabling the computation of derivatives [1]. However, this procedure poses different challenges:

- For large models the C and C++ code generation can be very slow.
- Only few structural information of the model itself is available in the generated code.
- Running the generated code completely using variables of adouble type may be computationally very expensive.
- Parallel execution to speed up the simulation is not available out of the box.

3 Efficient Implementation

The ADOL-C library uses the execution of a function with adouble type variables to generate an internal representation of the function called the trace. This trace, once created, can be repeatedly used to compute derivatives at different evaluation points. The authors have implemented functionality to read and write ASCII-based traces into the ADOL-C branch ascii2tape. The methods write_ascii_trace(FILENAME, tag) and read_ascii_trace(FILENAME, tag2) are provided to read and write ASCII traces. Such an ASCII trace can be generated by OpenModelica while it compiles the Modelica model in order to simulate it. This allows the authors to generate and evaluate traces without executing the C++ code generated by OpenModelica: Given the ability to read and write traces of a computer program ADOL-C's drivers like function, jacobian or hessian can be called outside of the C and C++ code via Python or Julia bindings.

*Corresponding Author, Benjamin.Jurgelucks@math.uni-paderborn.de
Figure 1: Execution time (realtime) of \( J \in \mathbb{R}^{n \times n} \) called from C++ (blue) and Julia (orange). Times exclude trace generation and compilation time for C++ code (approx. 200 ms only required once).

Pure trace evaluation is usually much faster than the execution of the generated code. The C++ code need not even be generated. Moreover, by generating the traces before the generation of the C or C++ code of the model the authors can exploit structural information like sparsity information explicitly available at this step of the process.

Furthermore, the authors are implementing functionality to calculate a full Jacobian by calling ADOL-C’s driver \texttt{jacobian} in parallel. Given a prerecorded trace the columns of the seed matrix are divided into partitions which will be independently handled in parallel. This is especially important for very large simulation cases where the serial evaluation of a full Jacobian may take a large amount of time.

4 Calling ADOL-C drivers in Julia

For the general usage of the techniques discussed above we wish to use the high-level programing language \texttt{Julia} within internals of OpenModelica. However, as this may lead to a slower derivative evaluation, we tested calling the ADOL-C driver \texttt{jacobian} from within Julia. This can be achieved by calling

\[
\text{ccall}(\text{:\texttt{jacobian}},\text{`PATH/TO/libadolc.so'}),\text{Cshort}, (\text{Cint},\text{Cint},\text{Cint},\text{Ptr[Cdouble]},\text{Ptr[Ptr[Cdouble]]}), \text{tag,}m,n,\text{args,refAr})
\]

with \( \text{jac} = \text{zeros(Cdouble, m,n)} \) and \( \text{refAr} = [\text{Ref(jac,i)} \text{ for i=1:size(jac,i):length(jac)}] \).

We can show (see Fig. 1) that calling the Jacobian driver of ADOL-C from Julia only has a small constant offset of about 0.3 sec.

5 Acknowledgements

The authors thank the German Federal Ministry of Education and Research (BMBF) for funding of the project PARADOM under the support code 01IH15002A.

References

Piecewise Linear AD via Source Transformation

Torsten Bosse,* Sri Hari Krishna Narayanan† Laurent Hascoët‡

Overview Algorithmic differentiation (AD) allows the efficient numerical computation of sensitivities for any mathematical function \( y = F(x) \), \( F : \mathbb{R}^n \rightarrow \mathbb{R}^m \) that is sufficiently smooth and given by a finite straight-line code.

\[
\begin{align*}
  v_{i-n} &= x_i & \text{for } i = 1, \ldots, n & \text{Initialization} \\
  v_i &= \varphi(v_j) \quad & \text{for } i = 1, \ldots, l & \text{Evaluation} \\
  y_{m-i} &= v_{l-i} & \text{for } i = 0, \ldots, m-1 & \text{Extraction}
\end{align*}
\]

Here, \( x \in \mathbb{X} \subseteq \mathbb{R}^n \) denotes a vector of input variables, \( y = g(x) \in \mathbb{R}^m \) the corresponding output variables, and \( \varphi \) some smooth intermediate assignments from a library \( \Phi \equiv \{ +, -, *, /, \sqrt{\cdot}, \exp, \sin, \cos, \ldots \} \). However, the smoothness assumption is violated in most real applications. For example, the evaluation routines of many physical applications contain nonsmooth expressions such as the absolute value, the maximum, and/or the minimum function, in order to avoid unrealistic quantities. In this case, the standard differentiation rules do not necessarily apply any more. Thus, the simpler models are questionable even if the derivatives are evaluated at points \( x \in \mathbb{R}^n \) where the function is differentiable, since they do not take nearby kinks or nonsmoothness into account. A remedy for this situation was recently proposed in [1], where the author presents a method to compute (directional) piecewise linear models of the original abs-factorable function instead of a simple linearization.

The piecewise linearization \( \Delta y = \Delta F(x; \Delta x) \), \( \Delta F : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m \), at a point \( x \in \mathbb{R}^n \) for a directional increment \( \Delta x \in \mathbb{R}^n \) represents the function in a more appropriate way and can be derived by a minor modification of the original code. Similar to the standard forward mode, the idea is based on defining an extended evaluation routine using the propagation rules

\[
\begin{align*}
  \Delta v_i &= \Delta v_j \pm \Delta v_k & \text{for } v_i &= v_j \pm v_k \\
  \Delta v_i &= \Delta v_k \Delta v_k + \Delta v_j & \text{for } v_i &= v_j \ast v_k \\
  \Delta v_i &= c_{ij} \ast \Delta v_j & \text{for } v_i &= \varphi(v_j) \neq \text{abs}(),
\end{align*}
\]

for all smooth intermediate expressions with corresponding partial derivatives \( c_{ij} = (\partial v_i / \partial v_j)_{j \neq i} \). Instead of propagating the direction for the absolute value by \( \Delta v_i = \text{sign}(v_j) \Delta v_j \), one employs the rule for the absolute value:

\[
\begin{align*}
  \Delta z_j &= v_j + \Delta v_j \\
  \Delta v_i &= \text{abs}(\Delta z_j) - v_i & \text{for } v_i &= \varphi(v_j) \equiv \text{abs}().
\end{align*}
\]

Accordingly, one can define corresponding rules for the maximum and minimum using the identities \( \text{max}(p, q) = (p + q + |p - q|)/2 \) and \( \text{min}(p, q) = (p + q - |p - q|)/2 \). As was shown in [2], the extended evaluation routine then provides a piecewise linearization of \( F \), which can be algebraically represented by an abs-normal form (ANF)

\[
\begin{bmatrix} \Delta z \\ \Delta y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} Z & L \\ J & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta z \end{bmatrix}
\]

using an additional vector of switching variables \( \Delta z \in \mathbb{R}^s \) for some suitable dimensions \( m, n, s \in \mathbb{N} \) and matrices/vectors

\[
\begin{align*}
  a & \in \mathbb{R}^s, & Z & \in \mathbb{R}^{s \times m}, & L & \in \mathbb{R}^{s \times s}, & b & \in \mathbb{R}^m, & J & \in \mathbb{R}^{m \times n}, & Y & \in \mathbb{R}^{m \times s}.
\end{align*}
\]

The matrices might depend on \( x \) and can be interpreted as partial derivatives. In detail, if \( z = (z_1, \ldots, z_s) \in \mathbb{R}^s \) denotes the arguments of all \( s \in \mathbb{N} \) absolute value functions that occur in the original code, then the matrices are

\[
\begin{align*}
  Z &= \left. \frac{\partial v}{\partial z_i} \right|_{z}, & L &= \left. \frac{\partial v}{\partial z_i} \right|_{z}, & J &= \left. \frac{\partial v}{\partial z_i} \right|_{z}, & Y &= \left. \frac{\partial v}{\partial z_i} \right|_{z}, & \Delta z &= \begin{bmatrix} \Delta z_1 \\ \Delta z_2 \\ \vdots \\ \Delta z_s \end{bmatrix}.
\end{align*}
\]

which consider only those parts of the mappings \( x \rightarrow z, |z| \rightarrow z, |z| \rightarrow y \), and \( x \rightarrow y \) that only involve smooth expressions. Both a simple evaluation of \( \Delta F \) for given \( x \) and direction \( \Delta x \) and the computation of the complete ANF can be done by using techniques from operator overloading and were already implemented in the AD package ADOL-C [3]. In this paper, we explain our initial development effort for computing the entries of \( Z, L, J, \) and \( Y \) and the vectors \( a \) and \( b \) by making simple modifications to the runtime system of the two source transformation tools OpenAD [4] and Tapenade [3]. The method will be demonstrated and validated for some simple test problems.

*Corresponding author, Argonne National Laboratory, tbosse@anl.gov
†Argonne National Laboratory, snarayan@mcs.anl.gov
‡INRIA Sophia-Antipolis, laurent.hascoet@inria.fr
Propagating incremental directions To use the piecewise linear differentiation drivers in OpenAD and Tapenade, the user is required only to replace all calls of the abs, min, and max function in the original code by the stub methods

\[\text{gabs}(z,u)[u = \text{abs}(z)], \quad \text{gmin}(z_1,z_2,u)[u = \text{min}(z_1,z_2)], \quad \text{and} \quad \text{gmax}(z_1,z_2,u)[u = \text{max}(z_1,z_2)],\]

The stub methods for the nonsmooth parts are then automatically replaced by methods that implement the corresponding propagation rules for the piecewise linear differentiation without changing the original results of \(F\).

Computing the entries of the ANF by using OpenAD For \(n\) input variables, \(m\) output variables, and intermediate active variables, the forward mode of OpenAD uses an active type containing an array, \(d\), of size \(n\) to propagate directional derivatives according to the propagation rules (2). Thus, if the code is executed in the smooth case, each of the \(m\) output variables will contain a derivative array that represents one row of the Jacobian matrix \(J\), if the derivative array of each of the input variables is assigned to one of the \(n\) basis vectors.

For the nonsmooth case, the runtime library of OpenAD changes the sizes of the derivative array \(d\) to \(n+s\) as described in Fig. 1(a). It also defines a global array \(dz\) and \(du\) that each have for any occurring absolute value one vector of size \(n+s\). By default, both arrays \(dz\) and \(du\) are initialized such that they form the Euclidean standard basis of dimension \(n+s\). The propagation of derivatives ensures that the entries of \(J, Z, L, \text{and } Y\) are computed mechanically. The desired quantities are stored in the corresponding portions of the derivative arrays for the output variables \(dz\) and \(du\) as shown in Fig. 1(c). The hand-coded replacement for the absolute value that is required for the computation of the ANF is based on the propagation rules (2) and given in Fig. 1(b).

\[
\begin{align*}
\text{type active} \\
\text{sequence} \\
\text{real(w2f__8) :: } v \\
\text{real(w2f__8), dimension(n+s) :: } d = 0.0d0 \\
\text{end type} \\
\text{real(w2f__8), dimension(s, n+s) :: } dz = 0.0d0 \\
\text{real(w2f__8), dimension(s, n+s) :: } du = 0.0d0
\end{align*}
\]

(a) Modified runtime library

| subroutine gabs(z, u) | use oad_active implicit none type(active) :: z intent(inout) z intent(inout) u oad_abs_s_index = oad_abs_s_index + 1 du(oad_abs_s_index,1:n+s) = dz(1:n+s) u%v = abs(z%v) u%d(1:n+s) = dz(oad_abs_s_index,1:n+s) end subroutine |

(b) Custom replacement for differentiated gabs

Figure 1: OpenAD code examples for the computation of the ANF and the resulting storage layout for the output variables containing the partial derivatives \(Z, L, J, \text{and } Y\)

References


Algorithmic Differentiation in the Stan Math C++ Library

Bob Carpenter∗ and Matthew D. Hoffman† and Marcus Brubaker‡ and Daniel Lee§ and Michael Betancourt¶ and Sebastian Weber∥ and Rob Trangucci∗∗

June 2016

Stan Math is an open-source C++ library implementing forward- and reverse-mode algorithmic differentiation through function and operator overloading. The library provides differentiable functions for probability and statistics, matrix operators, linear algebra solvers, and integrators for ordinary differential equations. The math library was designed to support the Stan statistical inference package, which provides derivative-based algorithms for Markov chain Monte Carlo sampling, optimization, and variational inference.

Reverse Mode

Object Oriented. For reverse mode, Stan Math uses an extensible object-oriented design for the algorithmic differentiation variables making up the expression graph. The base class holds a double value and adjoint and may be used directly for independent variables. Subclasses represent the results of specific operators applied to operands (functions applied to arguments). The chain rule is propagated down to the arguments through a virtual function.

Lazy Evaluation. Subclasses must store pointers to the operands for which derivatives are required in order to propagate the chain rule. The chain rule virtual function is evaluated lazily, and typically only needs the operands and value in order to compute the partials. In some cases, the partials are precomputed and stored as well. Lazy evaluation saves memory and can also save considerable computation in cases like addition where rather than multiplying by a constant one value, only additions need be performed.

Memory Management. Memory for subclass instances and the chain-rule stack is managed through an arena which automatically resizes and can be cleared for reuse or recovered in constant time. Memory management details are encapsulated by overriding operator new for the base variable class, automatically allocating subclass instances in the arena and placing them on the chain-rule stack, which is also in the arena. Utility functions are provided to allocate arrays for variadic operations rather than using containers that directly access the shared heap.

Client API. The application programmer interface (API) to differentiable functions is provided in terms of a pointer-to-implementation class, which encapsulates all details of the base class. Instances are defined by assigning independent values (implemented by overloading assignment). The result of any operation involving a variable is also a variable, including the final result.

Forward and Mixed Modes

Templated Tangents. For forward mode, Stan Math uses the standard approach of storing a value and tangent and computing the tangent values directly within function overloads for the forward-mode variable type.

Mixed Mode. Higher-order derivatives are available by nesting reverse mode within forward mode, with further nesting for higher-order derivatives.

Functionals. Stan encapsulates its forward- and reverse-mode implementations with functionals that operate over templated function objects to provide gradients, gradient-vector products, Jacobians, Hessians, Hessian-vector products, and some specialized higher-order operations such as gradients of traces of matrix-Hessian products (required for Riemannian Hamiltonian Monte Carlo).

Vectorized Log Density Functions

The log likelihood function for linear regression is defined as follows for an $N$-vector $y$ of outcomes, an $N \times K$ data matrix $x$, regression coefficient $K$-vector $\beta$, and scalar error scale $\sigma$.

$$
\log \text{normal}(y|x, \beta, \sigma) = \sum_{n=1}^{N} \log \text{normal}(y_n|x_n, \beta, \sigma) = -N \log \sqrt{2\pi} - N \log \sigma - \frac{1}{\sigma^2} \sum_{n=1}^{N} (y_n - x_n \beta)^2 + \text{const}
$$

∗Corresponding Author, Columbia University, carp@alias-i.com
†Adobe Research
‡University of Toronto, Scarborough
§Columbia University
¶University of Warwick
∥Novartis
∗∗Stan Group
Expression Templates. Expression templates for vectors allow scalars, Eigen vectors, Eigen row vectors, and standard library vectors to all be wrapped and used interchangeably. Being able to access the underlying size of the container expression templates enables the normal log density function to calculate $-\log \sigma$ or $1/\sigma^2$ only once, saving both time and space relative to a naïve repeated application of the full normal log density for each $y_n$.

Constant Dropping. Constants terms, such as $-\log \sqrt{2\pi}$ above, are not needed for Monte Carlo simulations or optimization. Density functions provide a boolean template parameter, which when set to true, allows the function to efficiently drop such constants. The Boost template traits library is used to recognize operations involving only double values that can be dropped. Thus if $\sigma$ is a constant, as in many Bayesian priors or penalty functions for maximum likelihood estimation, the $-\log \sigma$ term is also dropped.

Polymorphic Partial Evaluation. The code for both reverse mode and forward mode is shared, building up an implementation-neutral representation of the partials of the result with respect to the inputs in the evaluation of the vectorized function. In forward mode, this is used to compute the tangent value of the result. In reverse mode, a single implementation node is allocated on the stack, dramatically cutting down on virtual function calls (equivalently, switches or function pointer resolutions) during derivative propagation.

Linear Algebra and Matrix Derivatives

The Eigen C++ matrix library is used to implement matrix arithmetic (e.g., addition and multiplication) and linear algebra (e.g., log determinants, matrix division, eigendecomposition, and cholesky factorization). Wherever possible, matrix derivatives [1] are used to enable calls to Eigen functions to be made with double arguments, and put back together into the appropriate implementations with explicitly stored partials. This not only avoids overpromotion, it is much faster and uses less memory than naïvely algorithmically differentiating through Eigen’s templated implementations.

Solving Systems of Ordinary Differential Equations

Stan provides functions to integrate systems of ordinary differential equations specified as functors in order to solve initial value problems and provide derivatives of the solutions with respect to parameters (i.e., sensitivities). For non-stiff systems, Stan uses the 4th/5th-order Runge-Kutta solver from Boost’s Odeint package, and for stiff systems, the backward differentiation formula solver from Sundials with sensitivities (CVODES). In both cases, Stan uses nested reverse-mode algorithmic differentiation of the system function with respect to its parameters for the sensitivities. For stiff systems, the Jacobian of the system function is also computed with nested reverse-mode. This allows all computations to be carried out with double values and also allows control of error in the sensitivity calculations.

Comparisons to Other Systems

Stan’s approach is closest to that of Sacado [1] in both forward and backward mode, although the memory management, object orientation, and lazy gradient propagation are new in Stan. Stan is faster than all other reverse-mode C++ systems we were able to install (Adept, CppAD, Adol-C, Sacado) across a range of problems, with speedups ranging from a factor of two to factors of sixteen, with even greater gains in areas of Stan’s specialization in matrix operations and probability functions. Sacado is a bit faster than Stan on small problems, presumably due to Stan’s additional overhead for nesting and variables that do not themselves invoke the chain rule. Also, Adept [2] is faster when large arithmetic expressions are involved, because it uses expression templates to statically checkpoint compiled code for derivatives of a single expression. Adept’s expression templates are compatible with Stan’s basic architecture, and thus could be added in the future. Stan does not support tape reuse, as is done by CppAD [3]. This is partly because Stan’s primary applications support differentiating iterative and branching algorithms, and thus cannot be pre-taped, and partly because taping is fast with Stan’s lazy approach to chain rule propagation.

Acknowledgements

Stan was funded in part by grants from the Alfred P. Sloan Foundation, the U.S. Office of Naval Research, the U.S. National Science Foundation, the U.S. Institute for Educational Sciences, and the U.S. Department of Energy, as well as by the companies Novartis and YouGov.

References


KKT-based integration of differential-algebraic equations with optimality criteria

Jonathan Hißer‡, Jens Deussen† and Uwe Naumann‡
July 12, 2016

1. Introduction

Consider the following under-determined differential-algebraic equation (DAE) initial value problem where $x \in \mathbb{R}^n_x$ are the differential, $y \in \mathbb{R}^n_y$ are the algebraic variables and $p \in \mathbb{R}^n_p$ is a parametrization:

$$
\begin{align*}
\dot{x} &= f(t,x,y,p), \quad t \in (t_0,t_f], \quad x(t_0,p) = x_0(p) \\
0 &= g_i(t,x,y,p), \quad i = 1, \ldots, n_g, \quad n_g < n_y.
\end{align*}
$$

(1)

To deal with incomplete information in modeling it is sometimes possible to state embedded optimality criteria that select a locally (energy) optimal solution from the space of possible solutions as follows:

$$
\begin{align*}
\dot{x} &= f(t,x,y,p), \quad t \in (t_0,t_f], \quad x(t_0,p) = x_0(p) \\
y &\in \operatorname{arg\,min}_{y \in \mathbb{R}^n_y} \, w(t,x,y,p) \\
\text{s.t.} \quad 0 &= g_i(t,x,y,p), \quad i = 1, \ldots, n_g \\
0 &\leq h_i(t,x,y,p), \quad i = 1, \ldots, n_h.
\end{align*}
$$

(2)

The algebraic equations from (1) are included as equality constraints. We call (2) a differential-algebraic equation with optimality criteria (DAEO).

DAEO formulations emerge for example in systems biology when modeling the behavior of biochemical networks [1, 2] and in the combination of mass transfer with thermodynamic equilibrium models in atmospheric chemistry [3]. Numerical methods for solving DAEO using for example interior point methods have previously been investigated [4]. The numerical treatment of DAEO problems is also subject of ongoing research within both a BioSC and a DFG project (see acknowledgements).

On the poster we will present a software infrastructure for DAEO integration and discrete sensitivity analysis with regard to the parameters $p$ using multiple nested levels of algorithmic differentiation (AD) [5, 6]. In the following sections we shortly summarize some technical aspects of our approach.

2. KKT-based formulation

Assuming local convexity of the embedded optimization problem we can rewrite the DAEO as a DAE that contains the Karush-Kuhn-Tucker (KKT) conditions for the embedded optimization problem. The time dependent Lagrangian is

$$L(t,x,y,p,\lambda,\mu) = w(t,x,y,p) - \lambda^T \cdot g(t,x,y,p) - \mu^T \cdot h(t,x,y,p)$$

where $\lambda \in \mathbb{R}^{n_y}$ and $\mu \in \mathbb{R}^{n_h}$ are the multipliers for the equality and the inequality constraints respectively. We get the following DAE

$$
\begin{align*}
\dot{x} &= f(t,x,y,p), \quad t \in (t_0,t_f], \quad x(t_0,p) = x_0(p) \\
0 &= g_i(t,x,y,p), \quad i = 1, \ldots, n_g \\
0 &= h_i(t,x,y,p), \quad i \in A(t,x,y,p) \\
0 &= \mu_i, \quad i \notin A(t,x,y,p) \\
0 &= \nabla_y L(t,x,y,p,\lambda,\mu)
\end{align*}
$$

(3)

where $A(t,x,y,p)$ is the set of active constraints at time $t$. Switches in the set of active constraints can result in discontinuities in $\dot{x}$ and integration of such a DAE hence requires appropriate handling of switching events including their correct time location.

‡Corresponding Author, RWTH Aachen University, huesser@stce.rwth-aachen.de
†RWTH Aachen University, deussen@stce.rwth-aachen.de
‡RWTH Aachen University, naumann@stce.rwth-aachen.de
3 Integration and sensitivity analysis

Given type generic C++ implementations of the functions $f, w, g, h$ we generate a type generic residual of (3) using a tangent model of the Lagrangian generated by the operator overloading AD tool dco/c++ [6]. Based on the residual we implement a fully type generic and overloadable integrator using an implicit backward Euler method. We explicitly handle the set of active constraints, finding consistent solutions and location of constraint switches hence integrating over discontinuities. The integrator again employs AD to compute the system Jacobian of three different types of nonlinear systems to be solved by Newton’s method.

The consistent set of active constraints $A(t, x, y, p)$ of the embedded optimization problem needs to be found based on an active set method at $t = 0$ and every constraint switch event location. We repeatedly solve only the KKT conditions for $y, \lambda$ and $\mu$ without a time step and detect violated inequality constraints or negative Lagrange multipliers until a consistent constraint set is found.

For each time step $k$ with step size $s_k$ at time $t_k = t_{k-1} + s_k$ we solve the complete DAE system for $x_k$, $y_k$, $\lambda_k$ and $\mu_k$ by substituting a backward difference

$$\dot{x}_k = \frac{x(t_k) - x(t_k - s_k)}{s_k} = \frac{x_k - x_{k-1}}{s_k}.$$ 

We initially assume that the set of active constraints doesn’t change from the previous time step but check for violated inequality constraints or negative Lagrange multipliers in which case we have a constraint switch.

The time step $s_k$ used for the correct location of a discontinuity event needs to be found in the case of a constraint switch. For example in case of a violated inequality constraint $0 > h_i(t_k, x_k, y_k, p_k)$ we again solve the complete DAE system with an added equation for the constraint such that $0 = h_i(t_{k-1} + s_k, x_k, y_k, p_k)$ and in addition to $x_k, y_k, \lambda_k$ and $\mu_k$ we now also solve for $s_k$, the required step size to reach exactly the time $t_k$ where the constraint switch occurs.

Because our integrator is fully type generic a discrete sensitivity analysis is possible using operator overloading AD tools. We plan to use our software infrastructure to solve optimal control problems using both adjoint and forward sensitivity models. Dynamic optimization of DAEO can be especially challenging because of possible nonsmoothness with regard to $p$ resulting from the inequality constraints [7] and needs to be tackled using appropriate algorithmic methods such as the piecewise linearization proposed by Griewank et al. [8, 9] or the lexicographic derivatives proposed for this purpose by Khan et al. [10].

4 Conclusion

We plan to present a software infrastructure for solving DAEO that makes use of multiple nested levels of operator overloading AD. We will provide results from numerical experiments using a model from atmospheric chemistry [3] comparing a KKT-based approach to an explicit solution of the embedded optimization problem in each time step. For both approaches we investigate potential numerical problems such as ill-conditioned residuals near constraint switches. If progress is made in the time frame until the conference we will also present first results with regard to potentially nonsmooth optimal control.

Acknowledgements

Our work was financially supported by the BioSC project “Incremental multi-scale and multi-disciplinary modeling of processes in bioeconomy” (BEProMod) funded by the Ministry of Innovation, Science and Research of the German State of North Rhine-Westphalia, as well as the DFG project “Aachen dynamic optimization environment” (ADE) funded by the German Research Foundation.

References


Evolving the Incremental $\lambda$ Calculus into a Model of Forward AD

Robert Kelly*  Barak A. Pearlmutter†  Jeffrey Mark Siskind‡

April 2016

Introduction

Formal transformations somehow resembling the usual derivative are surprisingly common in computer science, with two notable examples being derivatives of regular expressions [1] and derivatives of types [2, 3]. A newcomer to this list is the incremental $\lambda$-calculus, or ILC, a “theory of changes” that deploys a formal apparatus allowing the automatic generation of efficient update functions which perform incremental computation [4]. An example of this would be using the ILC derivative-like operator $D$ to alter a function $f : B \rightarrow B$, which performs some major reorganization on a database (of type $B$), into the update function $Df : B \rightarrow B \rightarrow B$. Here $\Delta B$ is the type of changes to $B$. So $Df$, given an initial database, maps a change to that input database to a change to the output database. This in principle, and as shown in their work also in practice, allows enormous savings when the change to the input is small compared to the size of the input itself. Resemblance to the standard derivative can be exhibited by a simple example

$$D (\lambda x . f (g x)) \rightsquigarrow (\lambda x x' . Df (g x) (Dg x x')) \quad (1a)$$

or

$$D (f \circ g) x \rightsquigarrow Df (g x) \circ Dg x \quad (1b)$$

which seems suspiciously similar to the familiar Calculus 101 chain rule.

The ILC is not only defined, but given a formal machine-understandable definition—accompanied by mechanically verifiable proofs of various properties, including in particular correctness of various sorts. Here, we show how the ILC can be mutated into propagating tangents, thus serving as a model of Forward Accumulation Mode Automatic Differentiation.

This mutation is done in several steps. These steps can also be applied to the proofs, resulting in machine-checked proofs of the correctness of this model of forward AD.

The Mutagenic Steps

There are two differences between the incremental $\lambda$ calculus and forward AD. First, changes rather than tangents are propagated. These changes are elements of change sets, and constitute finite (i.e., not infinitesimal) modifications. (For example, a change to a list might consist of swapping the first two elements, and a change to a number might consist of increasing its value by 5.) In numerics, these would be “differences” rather than “differentials”, and $\Delta$ rather than $\partial$. Second, the changes are passed as additional arguments instead of being bundled together with primal values. Passing changes in additional arguments makes great sense in the domain of incremental computation, where the whole point of the construction is to partially evaluate a function $Df : \alpha \rightarrow \Delta \alpha \rightarrow \Delta \beta$ with respect to $f$’s original input, yielding a mapping of changes to changes: $\Delta \alpha \rightarrow \Delta \beta$. But in the context of forward AD, we wish to propagate tangent values in parallel with primal values, which necessitates both bundling the “new” values with the original ones, and including the original output in the output of the transformed function.

We proceed to eliminate these two differences. This is done in two stages. First, considering only power series change sets to the base type $\mathbb{R}$. And second, uncurrying the outputs of the derivative operator and causing it to propagate change sets and primal values bundled together all the way through to its output. Truncating the power series changes them into Dual Numbers [7], yielding the familiar Forward AD. A commutative diagram of these steps

---

*Corresponding Author, Dept of Computer Science, National University of Ireland Maynooth, funded by the Irish Research Council, rob.kelly@cs.nuim.ie
†Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net
‡School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu

1 The approach detailed here stands in contrast to the Simply Typed $\lambda$-Calculus of Forward Automatic Differentiation [5]. Aside from some issues with confluence, that work folded together levels of the hierarchy by not distinguishing numeric basis functions which operate on $\mathbb{R}$ from those which are lifted to operate on Dual numbers, while here these are distinguished. Moreover, here we have a framework for machine-readable machine-verified proofs of various correctness and efficiency properties. This approach differs from the Differential Lambda-Calculus [6] in analogous ways: complexity, machine-checked proofs, and explicit segregation of levels of differentiation.
is shown in Figure 1. The original ILC is in the top left, with relevant changes indicated with transitions to new states or nodes. Each of these edges leads to a different combination of forward AD in the ILC. The power series and uncurry steps can be taken in either order, so the diagram should commute.

Let us describe these two steps in a bit more detail.

**Step One: Power Series**

To see how power series change sets are introduced, we note that the ILC allows change sets to be defined for any base type \( \tau \). These change sets need only obey a particular set of axioms, which in our context amounts to associativity of addition of real numbers. We constrain ourselves to consider only change sets to reals: the base type \( \tau \) is handled by the standard \( \lambda \)-calculus mechanisms for avoiding variable capture during \( \beta \)-substitution, e.g., \( \alpha \)-renaming.

We further define an operator \( \text{coeff} \) which takes a nonnegative integer index and a power series in \( \varepsilon \) wrapped in a \( \lambda \) expression, i.e., \( (\lambda \varepsilon . (\text{ps}_\varepsilon)) \), and yields the requested coefficient of the given power series:

\[
(\text{ps}_\varepsilon) ::= 0 \mid \varepsilon \ast (\text{ps}_\varepsilon) \\
(\text{ps}_\varepsilon) ::= \mathbb{R} \mid \mathbb{R} + (\text{ps}_\varepsilon) \\
\Delta \mathbb{R} \equiv (\text{ps}_\varepsilon)
\]

For a specific value of \( \varepsilon \) (possibly subject to conditions of convergence) this would take on a particular numeric value. We further define an operator \( \text{coeff} \) which takes a nonnegative integer index and a power series in \( \varepsilon \) wrapped in a \( \lambda \) expression, i.e., \( (\lambda \varepsilon . (\text{ps}_\varepsilon)) \), and yields the requested coefficient of the given power series.

\[
\text{coeff} \ 0 (\lambda \varepsilon . r) \rightarrow r \\
\text{coeff} \ 0 (\lambda \varepsilon . r + \varepsilon \ast e) \rightarrow r \\
\text{coeff} \ 0 (\lambda \varepsilon . \varepsilon \ast e) \rightarrow 0 \\
\text{coeff} \ i (\lambda \varepsilon . r + \varepsilon \ast e) \rightarrow \text{coeff} \ (i - 1) (\lambda \varepsilon . e) \\
\text{coeff} \ i (\lambda \varepsilon . \varepsilon \ast e) \rightarrow \text{coeff} \ (i - 1) (\lambda \varepsilon . e)
\]

For instance,

\[
\text{coeff} \ 2 (\lambda \varepsilon . 0.1 + \varepsilon \ast (0.2 + \varepsilon \ast (0.3 + \varepsilon \ast (0.4 + \varepsilon \ast (0.5 + \cdots)))))) \rightarrow 0.3
\]

Useful properties of such a change set are straightforward to establish: closure under the derivatives of the numeric basis functions, and dependence during such operators of coefficients only on coefficients of the same or lower order. The first property is necessary for consistency, while the second allows these power series to be truncated at \( \varepsilon^2 \), thus yielding the tangents of standard forward AD. With this machinery, we could define the familiar derivative \( \text{diff} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \), for instance \( \text{diff} \sin = \cos \), as

\[
\text{diff} \ f \ x \equiv \text{coeff} \ 1 (\lambda \varepsilon . (D f x (\varepsilon \ast 1)))
\]

By defining \( \text{coeff} \) to distribute over algebraic datatypes

\[
\text{coeff} \ i (\lambda \varepsilon . \text{Constructor} e_1 \cdots e_n) \rightarrow \text{Constructor} (\text{coeff} \ i (\lambda \varepsilon . e_1)) \cdots (\text{coeff} \ i (\lambda \varepsilon . e_n))
\]

and post-compose over functions

\[
\text{coeff} \ i (\lambda \varepsilon . (\lambda x . e)) \rightarrow (\lambda x . \text{coeff} \ i (\lambda \varepsilon . e))
\]

this machinery can find directional derivatives of functions with non-scalar output, including Church-encoded output.

In this formulation, the tagging necessary to distinguish distinct nested invocations of derivative-taking operators [8, 9] is handled by the standard \( \lambda \)-calculus mechanisms for avoiding variable capture during \( \beta \)-substitution, e.g., \( \alpha \)-renaming.

---

**Figure 1: Mutating the Incremental \( \lambda \)-Calculus into Forward AD.**
Step Two: Uncurrying and Bundling

The second step is uncurrying arguments, and bundling the output. We need to change the type of the derivative operator from

$$
\mathcal{D} : (t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow u) \rightarrow (t_1 \rightarrow \Delta t_1 \rightarrow t_2 \rightarrow \Delta t_2 \rightarrow \cdots \rightarrow t_n \rightarrow \Delta t_n \rightarrow \Delta u)
$$

(6)

to

$$
\hat{\mathcal{D}} : (t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow u) \rightarrow (Ft_1 \rightarrow Ft_2 \rightarrow \cdots \rightarrow Ft_n \rightarrow Fu)
$$

(7)

where $Ft$ is isomorphic to $t \times \Delta t$, a primal value bundled with its change set. If we define $F(t_1 \rightarrow t_2) = Ft_1 \rightarrow Ft_2$ then this yields a simpler type signature,

$$
\hat{\mathcal{D}} : t \rightarrow Ft
$$

(8)

The mechanics of this change are straightforward, requiring that the ILC reductions be modified to take the new shape. Note that, thus uncurried and carrying primal and change set values in tandem, the chain rules of Equation 1 are simplified: $\hat{\mathcal{D}} (f \circ g) \Rightarrow \hat{\mathcal{D}} f \circ \hat{\mathcal{D}} g$.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/I2637 and by NSF grant 1522954-IIS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

1 Introduction

Numerical libraries are often used by scientists while writing their codes. This allows them to write the code faster and concentrate on their research rather than spend time on implementing and testing numerical algorithms. Once adjoints of the resulting code are required, the routines from those numerical libraries that saved the development time while writing the simulation in first place, turns out to be one of the major issues. If the library does not ship corresponding adjoint routines, the user has to implement those on his own. Adjoint code should be developed by experts in that field, due to many possible problems that might occur. Writing a continuous adjoint version of the routines as described in [1] or [2] is not an easy assignment and requires a deeper insight into the algorithms. It also may simply be not applicable, for example if your original routine has not converged. Using Algorithmic Differentiation (AD) may fail because the source code of the routine is not available. Even if a source code is available applying AD to the numerical library routine can be a major issue. For example because the underlying code base is too big or the code structure of the library does not suite well with requirements of the AD tool of your choice. And after all there is still the problem of testing the code. In any case the library vendor is the natural instance for providing adjoint version of his library routines. As the vendor has the best knowledge and expertise about the used algorithms, has access to the source code and can do proper testing. Although there are many numerical libraries out there that help you to develop your forward model, almost none of them provide you significant support when it comes to develop an adjoint model.

The Numerical Algorithm Group (NAG) is a vendor of a big general purpose numerical library (NAG Numerical Library). The library routines are accessible from various languages such as Fortran, C, Java etc. Recently NAG started working on AD (first order adjoint) versions of their Fortran and C Library routines. The resulting AD version of the library is called NAG AD Library. AD tools dco/c++ and dco/Fortran serve as basis for the Adjoint NAG Library. dco/c++ and dco/Fortran ([3]) are both operator overloading based AD tools.

2 NAG AD Library

NAG AD Library contains the adjoint version of the NAG Library routines. To provide the adjoint version of the routines the C++ AD tool dco/c++ and its Fortran counterpart dco/Fortran are used. Both tools implement AD based on operator overloading. Although both tools support the computation of derivatives in forward and reverse mode of arbitrary order, the AD routines currently implement only first order reverse mode of AD. The AD version of the routines is generated by replacing all floating point variables in the code with the first order adjoint variables from dco/c++ or dco/Fortran. We refer to the resulting interface as association by address interface (the reason for the name is that the value and the adjoint part are distinguished through the address). The example of the interface is presented in Listing 1 and Listing 2. In Listing 1 we see the original Fortran interface of the NAG routine. Listing 2 demonstrates corresponding association by address interface. The association by address interface can only be used by users who have access to dco/c++ or dco/Fortran. Therefore for those who does not use these AD tools we provide the association by name interface. See Listing 3 for an example. The association by name interface internally is based on a driver that calls the association by address interface, extracts the desired adjoint projections and stores them in the corresponding adjoint variables. The adjoint variables are identified by name suffix _a1s. This is also the reason for the naming, as the value and adjoint information is distinguished by name.

*LuFG Informatik 12, RWTH-Aachen University, lotz@stce.rwth-aachen.de
†LuFG Informatik 12, RWTH-Aachen University leppkes@stce.rwth-aachen.de
‡Corresponding Author, NAG, viktor.mosenkis@nag.co.uk
§LuFG Informatik 12, RWTH-Aachen University, naumann@stce.rwth-aachen.de
Listing 1: Original NAG interface

Subroutine c05ayf(a, b, eps, eta, f, x, iuser, ruser, ifail)
Implicit None
Double Precision, Intent (In) :: a, b, eps, eta
Double Precision, Intent (Out) :: x
Integer, Intent (Inout) :: fail
Double Precision, Intent (Inout) :: ruser(*)
Integer, Intent (Inout) :: iuser(*)
Interface
Function f(x, iuser, ruser)
Implicit None
Double Precision :: f
Double Precision, Intent (In) :: x
Double Precision, Intent (Out) :: ruser(*)
Integer, Intent (Inout) :: iuser(*)
End Function f
End Interface
End Subroutine c05ayf

Listing 2: Association by address interface

Subroutine c05ayf_dco_als(a, b, eps, eta, f, x, iuser, ruser, ifail)
Implicit None
Double Precision, Intent (In) :: a, b, eps, eta
Double Precision, Intent (Out) :: x
Integer, Intent (Inout) :: fail
Double Precision, Intent (Inout) :: ruser(*)
Integer, Intent (Inout) :: iuser(*)
Interface
Function f(x, iuser, ruser)
Implicit None
Double Precision, Intent (In) :: a, b, eps, eta
Double Precision, Intent (Out) :: x
Integer, Intent (Inout) :: ruser(*)
Integer, Intent (Inout) :: iuser(*)
End Function f
End Interface
End Subroutine c05ayf_dco_als

Listing 3: Association by name interface

Subroutine c05ayf_dco_als(a, a_als, b, b_als, eps, eps_als, eta, &
eta_als, f_als, x, x_als, iuser_als, ruser_als, iuser, ruser, & ruser_als, ifail)
Implicit None
Double Precision, Intent (In) :: a, b, eps, eta
Double Precision, Intent (Inout) :: a_als, b_als, eps_als, eta_als
Double Precision, Intent (Out) :: x
Double Precision, Intent (Inout) :: x_als
Integer, Intent (Inout) :: ifail
Integer, Intent (Inout) :: iuser_als
Double Precision, Intent (Inout) :: ruser_als
Double Precision, Intent (Inout) :: ruser_als
Integer, Intent (Inout) :: iuser_als
Double Precision, Intent (Inout) :: ruser_als
Interface
Function f_als(ret_als, x, x_als, iuser, ruser, ruser_als) Result (ret)
Implicit None
Double Precision :: ret
Double Precision, Intent (In) :: x
Double Precision, Intent (Inout) :: ruser(*)
Integer, Intent (Inout) :: iuser(*)
End Function f
End Interface
End Subroutine c05ayf_dco_als

3 Conclusions and further work

On the poster we will present the existing interfaces for both C++ and Fortran and present the list of currently available routines. We will discuss in detail how the association by name interface works. We will also talk about advantages and disadvantages of both interfaces. For the future we plan to extend the number of available routines. We also plan to provide advanced versions of the routines. So far we simply apply the operator overloading tool the code base. Therefore the user does not have control over the amount of memory used to reverse the control flow. In the future we want to provide more robust routines where the user can control the memory usage. For example by using hybrid approach and use of symbolic adjoints of parts of the code (like linear equation solvers).

References


Mixed Integer Programming for Call Tree Reversal.
Johannes Lotz* and Uwe Naumann† and Sumit Mitra‡
June 2016

As a poster, we want to present recent advances with a mixed integer programming (MIP) formulation for the call tree reversal (CTR) problem. This includes the formulation itself as well as results for a realistic case study based on PETSc. A corresponding paper is accepted at CSC16 (SIAM Workshop on Combinatorial Scientific Computing 2016). CTR is a special case of checkpointing, which is a widely used technique to reverse the data flow of a computer program with only a limited amount of available persistent memory. In contrast to the general checkpointing approach, CTR restricts possible checkpoints to subprogram calls. For the case study, commercial solvers are used for solving MIP and results are compared to known heuristics.

We consider decompositions of numerical simulation programs $P$ implementing multivariate vector functions $F : \mathbb{R}^m \rightarrow \mathbb{R}^n$, $y = F(x)$, into single assignment code

$$v_j = \varphi_j(v_i)_{i < j} , \quad j = 1, \ldots, p + m ,$$

where $v_{i-n} = x_i$ for $i = 1, \ldots, n$, $y_k = v_{p+k}$ for $k = 1, \ldots, m$ and $i < j$ denotes a functional dependence of $v_j$ on $v_i$ as an argument of the elemental function $\varphi_j$ for $j = 1, \ldots, p$. DATA FLOW REVERSAL (DFR) refers to accessing the required values $v_j$ in reverse order, that is, for $j = p + m, \ldots, 1 - n$. The $v_j$ cannot be assumed to be persistent due to potential overwriting of memory locations used for their storage in $P$, e.g.

**double** $x=1; \textbf{while}(\text{abs}(x)>1e-3) \ x=x\sin(x);$ 

Hence, a *store-all* approach to DFR pushes (records) the $v_j$ for $j = 1 - n, \ldots, p + m$ onto a stack followed by last-in/first-out recovery. Alternatively, each $v_j$ can be recomputed from stored inputs when required, that is, for $j = p + m, \ldots, 1$. This *recompute-all* approach yields quadratic (in $p$) arithmetic cost (number of floating-point operations; flops) while minimizing the amount of persistent memory occupied by the stack ($= n$ locations (locs) as inputs cannot be recomputed). *Store-all* DFR yields a stack of size $p + n$ locs thus minimizing the arithmetic cost at the expense of maximum memory requirement.

Checkpointing methods seek to minimize the arithmetic cost through (near-)optimal compromises between storage and recomputation of the required values for a given upper bound ($\geq n$) on the persistent memory. The corresponding DAG REVERSAL problem is known to be NP-complete [1]. Certain relevant special cases can be solved efficiently; see [2, 3]. Conceptually, a checkpoint for DFR of a section $v_j$, $j = l, \ldots, u \leq l$, in Equation (1) comprises the values of all $v_k$, $k \leq l$, (inputs to the section) read when computing the $v_j$. The variant of CALL TREE REVERSAL (CTR) we consider restricts the set of permissible checkpoints to arguments of subprogram calls in $P$. CTR is known to be NP-complete [4]. Greedy heuristics were proposed in [5].

In the following, we introduce the MIP, starting with the definition of CTR. A *call tree* $T = (S, r, \chi)$ with vertices $S$ is a tree with root $r \in S$ induced by the execution of a program. Each vertex $s \in S$ represents a call of a subprogram and each directed edge connects caller and callee. The ordered set $\chi[s] \subset S$ contains the callees of $s$. The call order is from left to right and depth-first in graphical representations of $T$. An annotated call tree is obtained by attaching values $M'[s] \in \mathbb{R}$ and $m[s] \in \mathbb{R}^{n[s]+1}$ to each vertex $s \in S$, where $n[s] = |\chi[s]|$. $M'[s]$ is the size of the subprogram argument checkpoint and $m[s]$ contains the stack sizes corresponding to the stages of $s$ executed before ($m[s]_0$, in-between ($m[s]$, $i \in \{1, \ldots, n[s]-1\}$) and after ($m[s]_n[s]$) its subprogram calls. The required stack size for one subprogram call (also referred to as persistent memory requirement (PMR)) is denoted by $m[s] = \sum_{i=0}^{n[s]} m[s]_i \in \mathbb{R}$. CTR is to find an optimal reversal scheme $y = y[s]_{s \in S}$ for the call tree $T(S, r, \chi)$ for a given upper bound $M$ on the available persistent memory. $y[s] \in \{0,1\}$ defines for each subprogram call, if it needs to be checkpointed or not. Feasibility requires a PMR of less than or equal to $M$ locs. A feasible reversal scheme $y$ is optimal if there is no other feasible reversal scheme $y'$ with $c(y') < c(y)$. The cost function $c(y)$ possibly takes into account various aspects of the problem, but is usually chosen to be the runtime overhead introduced by the checkpointing scheme in comparison to one evaluation of the original program.

MIP is defined by equations for synthesized ($M'$) and inherited ($M'$) attributes representing the PMR at the given point. For example, $M'[s]$, represents the size of the stack immediately prior to recovery of the $i$-th stage of a
subprogram \(s\) in the reverse section. MIP gets the form

\[
M_f^s = m[s] + \sum_{\sigma \in \chi[s]} [(1 - y[\sigma]) \cdot M_f^{}[\sigma] + y[\sigma] \cdot M_c^{}[\sigma]]
\]

\[
M_r^r = M_f^{}[r] = M_r^{}[s] = M_r^{}[\kappa] - m[\kappa] + y[s] \left( M_f^{}[s] - M_c^{}[s] \right) \quad \forall s \in S \setminus \{r\} \text{ and } \kappa, i : s = \chi[i]_i
\]

and feasibility is enforced by \(M_r^s \leq M \quad \forall s \in S\). The simplest cost function can be chosen as \(c(y, \bar{c}) = \sum_{s \in S} y[s] \cdot \bar{c}[s]\), where checkpointing \(s\) introduces an overhead of \(\bar{c}[s]\), with \(\bar{c} = \bar{c}[s]_{s \in S}\).

The overall software framework used is visualized in the following figure.

We search for the optimal checkpointing scheme represented by the binary variable \(y \in \{0, 1\}^{|S|}\). To retrieve \(y\), we apply a source-to-source compiler (I) to the complete code base of the original program \(\mathbb{P}\). The resulting instrumentation of the source code is performed using Clang’s LibTooling capabilities\(^1\). The instrumented program \(\mathbb{P}'\) not only computes the function values, but it also dumps the required call tree information \(\mathcal{T}\) into a file (II). This step is followed by solving the MIP (III) with a Python-driven CMPL [6] implementation using different solvers as backends. Finally, the computed checkpointing scheme needs to be integrated into the reversed program \(\mathbb{P}'\) (IV), which is the subject of an ongoing development effort.

In comparison to the different heuristics, solving the MIP performs better, as can be seen in the following figure. It shows for varying upper bounds on the available persistent memory the run time overhead of the found solution.

LMI and LMD are two different greedy heuristics. The underlying code used here is a PETSc implementation of the Bratu equation for a mesh size of 250 \(\times\) 250.

The availability of MIP solutions for CTR enables evaluation of the quality of known heuristics. Moreover, different objective functions can be formulated to address various aspects of the underlying computational and development efforts. This increased level of flexibility facilitates further in-depth investigation of the CTR problem.

References


---

\(^1\)http://clang.llvm.org/docs/LibTooling.html
Validated Computation of the Local Truncation Error of Runge-Kutta Methods with Automatic Differentiation

Olivier Mullier*, Alexandre Chapoutot†, Julien Alexandre Dit Sandretto‡

April 11, 2016

1 Introduction

Validated numerical integration is an appealing approach to produce rigorous results on Initial Value Problem of Ordinary Differential Equations (ODE). An IVP-ODE is defined by

\[ \begin{align*}
\dot{y} &= f(t, y) \\
y(0) &\in \mathcal{Y}_0 \subseteq \mathbb{R}^n, \ t \in [0, t_{\text{end}}].
\end{align*} \tag{1} \]

The set \( \mathcal{Y}_0 \) of initial conditions is used to model some (bounded) uncertainties. For a given initial condition \( y_0 \), the solution when it exists is denoted \( y(t; y_0) \). The goal of validated (or rigorous) numerical integration methods is to compute the set of solution, or an over-approximation, of the solutions of (1), i.e., \( \{ y(t; y_0) : \forall y_0 \in \mathcal{Y}_0, \forall t \in [0, t_{\text{end}}] \} \).

Most of the rigorous techniques defined so far, since Ramon Moore’s seminal work [1], are based on Taylor series approach, see for example [2, 3, 4, 5] and the references therein. Nevertheless, it is unlikely that only one kind of methods is adapted to all various classes of ODE. So, more recent work [6, 7, 8, 9, 10] deals with the adaptation of Runge-Kutta methods to be validated methods in order to try to benefit the good properties such as A-stability. A generic \( s \)-step Runge Kutta method for IVP-ODE is defined by

\[ y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \] \tag{2}

with

\[ k_i = f(t_n + c_i h, y_n + \sum_{j=1}^{s} a_{ij} k_j), \quad i = 1, \ldots, s. \] \tag{3}

The real coefficients \( c_i, a_{ij} \) and \( b_i \) fully characterize a Runge-Kutta methods, see [11].

The challenge to make a Runge-Kutta validated is to bound the local truncation error (LTE), i.e., the distance between the true solution \( y(t_n; y_{n-1}) \) at time \( t_n \) with \( y_{n-1} \) as initial conditions and the numerical solution \( y_n \) starting from the same initial condition so to bound \( y(t_n; y_{n-1}) - y_n \).

Following the order condition, see [12], a Runge-Kutta method is of order \( p \) if the \( p \) first terms of the Taylor form associated to the numerical solution \( y_n \) are equal to the terms of the exact solution of (1) that is \( y(t; y_{n-1}) \) assuming the same initial condition. In this case the LTE corresponds to the difference of the two Taylor remainders. Now, the challenge is to compute these Taylor remainders.

In this paper, we propose a novel approach to bound the LTE based on the order condition which is usable for explicit and implicit Runge-Kutta methods. More precisely, our approach is an instance of the algorithm defined in [13] and applied in the context of validated numerical integration methods based on Runge-Kutta methods.

2 Bounding the local truncation error

One of the great ideas of John Butcher in [12] is to express on the same basis of elementary differentials the Taylor expansion of the exact solution of (1) and those of the numerical solution. Those elementary differential are made of sums of partial derivatives of \( f \) with respect to the components of \( y \). An other great idea of John Butcher in [12] is to relate these partial derivatives of order \( q \) to a combinatorial problem to enumerate all the trees \( \tau \) with exactly \( q \) nodes.

From the structure of a tree \( \tau \) one can map a particular partial derivative, see [12] for more details. In consequence, one has the three following theorems which are used to express the order condition of Runge-Kutta methods. In theorems 2.1 and 2.2 \( \tau \) is a rooted tree, \( F(\tau) \) is the elementary differential associated to \( \tau \), \( r(\tau) \) is the order of \( \tau \) (the number of vertices it contains), \( \gamma(\tau) \) is the density, \( \alpha(\tau) \) is the number of equivalent trees and \( \psi(\tau) \) the elementary weight of \( \tau \) based on the coefficients \( c_i, a_{ij} \) and \( b_i \) defining a Runge-Kutta method.

*Corresponding author, ENSTA ParisTech, Palaiseau France, olivier.mullier@polytechnique.edu
†ENSTA ParisTech, Palaiseau, France, alexandre.chapoutot@ensta-paristech.fr
‡ENSTA ParisTech, Palaiseau, France, julien.alexandre-dit-sandretto@ensta-paristech.fr
Theorem 2.1. The $q$-th derivative w.r.t. time of the exact solution is given by

$$y^{(q)} = \sum_{r(\tau)=q} \alpha(\tau) F(\tau)(y_0) .$$

Theorem 2.2. The $q$-th derivative w.r.t. time of the numerical solution is given by

$$y_1^{(q)} = \sum_{r(\tau)=q} \gamma(\tau) \phi(\tau) \alpha(\tau) F(\tau)(y_0) .$$

Theorem 2.3 (Order condition). A Runge-Kutta method has order $p$ iff

$$\phi(\tau) = \frac{1}{\gamma(\tau)} \forall \tau, r(\tau) \leq p .$$

In consequence of Theorem 2.3 the LTE of a Runge-Kutta method is defined by

$$y(t_j; t_{j-1}) - y_j = \frac{h^{p+1}}{(p+1)!} \sum_{r(\tau)=p+1} \alpha(\tau) \left[1 - \gamma(\tau) \psi(\tau)\right] F(\tau)(y(\xi)) \quad \xi \in [t_{j-1}, t_j] . \quad (4)$$

Based on (4), validated numerical Runge-Kutta methods were successfully developed in [14] mainly using a symbolic generation of the elementary differentials. The limitation of the symbolic computation approach is that the number of partial derivatives of $f$ increases exponentially with the order $q$ and the syntactic expression may increase also exponentially so this approach does not scale up. During our investigation to overcome these limitations the work [13] was published. It defines an algorithm to compute B-series, i.e., sums of elementary differentials, only as an abstract mathematical object without targeting a particular use. Our contribution is then to instantiate the algorithm defined in [13] to fit our purpose to make validated Runge-Kutta methods. This algorithm will scale up as it is based on automatic differentiation techniques, more precisely, the techniques defined in [15]. It is also based on a particular factorization of the elementary differential in order to reduce the computation of the same sub-expressions appearing in different elementary differentials.

Note that in (4), the value of $y(\xi)$ is bounded following classical approach in validated numerical integration methods. More precisely, a variant of the Picard operator, see [2], is used in combination with interval arithmetic. Hence, it is possible to bound the value of the (4) in order to make any explicit and implicit Runge-Kutta validated.

3 Conclusion

In this paper, the computation of a guaranteed outer approximation of the local truncation error of a Runge-Kutta method is presented. Combining the computation of the LTE with the classical two step integration method that is the use of the Picard-Lindelöf operator to enclose all the solutions on one step, and the computation of the approximated solution makes possible the validated simulation of an ordinary differential equation with any Runge-Kutta method (implicit or explicit). This new method benefits from the previous work on the computation of B-series from Bartha et alii.

References


Vibrato and Automatic Differentiation for High Order Derivatives and Sensitivities of Financial Options

Gilles Pagès * Olivier Pironneau † Guillaume Sall ‡

June 21, 2016

Abstract
This work deals with the computation of second or higher order greeks of financial securities. It combines two methods, Vibrato and automatic differentiation and compares with other methods. We show that this combined technique is faster than standard finite difference, more stable than automatic differentiation of second order derivatives and more general than Malliavin Calculus. We present a generic framework to compute any greeks and present several applications on different types of financial contracts: European and American options, multidimensional Basket Call and stochastic volatility models such as Heston’s model. We give also an algorithm to compute derivatives for the Longstaff-Schwartz Monte Carlo method for American options. We also extend automatic differentiation for second order derivatives of options with non-twice differentiable payoff.

Keywords: Financial securities, risk assessment, greeks, Monte-Carlo, automatic differentiation, vibrato

Due to BASEL III regulations, banks are requested to evaluate the sensitivities of their portfolios every day (risk assessment). Some of these portfolios are huge and sensitivities are time consuming to compute accurately. Faced with the problem of building a software for this task and distrust automatic differentiation for non-differentiable functions, we turned to an idea developed by Mike Giles called Vibrato.

Vibrato at core is a differentiation of a combination of likelihood ratio method and pathwise evaluation. In Giles [8], [9], it is shown that the computing time, stability and precision are enhanced compared with numerical differentiation of the full Monte Carlo path.

In many cases, double sensitivities, i.e. second derivatives with respect to parameters, are needed (e.g. gamma hedging).

Finite difference approximation of sensitivities is a very simple method but its precision is hard to control because it relies on the appropriate choice of the increment. Automatic differentiation of computer programs bypass the difficulty and its computing cost is similar to finite difference, if not cheaper. But in finance the payoff is never twice differentiable and so generalized derivatives have to be used requiring approximations of Dirac functions of which the precision is also doubtful.

The purpose of this works is to investigate the feasibility of Vibrato for second and higher derivatives. We will first compare Vibrato applied twice with the analytic differentiation of

*Laboratoire de Probabilités et Modèles Aléatoires, UMR 7599, UPMC, Case 188, 4 pl. de Jussieu, F-75252 Paris Cedex 5, France, gilles.pages@upmc.fr.
†Laboratoire Jacques Louis Lions, UMR 7598, Case 187, 4 pl. de Jussieu, F-75252 Paris Cedex 5, France, olivier.pironneau@upmc.fr.
‡(corresponding author) Laboratoire de Probabilités et Modèles Aléatoires, UMR 7599, UPMC, Case 188, 4 pl. de Jussieu, F-75252 Paris Cedex 5, France, guillaume.sall@upmc.fr.
Vibrato and show that it is equivalent; as the second is easier we propose the best compromise for second derivatives: Automatic Differentiation of Vibrato.

In [3], Capriotti has recently investigated the coupling of different mathematical methods – namely pathwise and likelihood ratio methods – with an Automatic differentiation technique for the computation of the second order greeks; here we follow the same idea but with Vibrato and also for the computation of higher order derivatives.

Automatic Differentiation (AD) of computer program as described by Griewank in [15], [16], Naumann in [22] and Hascoet in [18] can be used in direct or reverse mode. In direct mode the computing cost is similar to finite difference but with no roundoff errors on the results: the method is exact because every line of the computer program which implements the financial option is differentiated exactly. The computing cost of a first derivative is similar to running the program twice.

Unfortunately, for many financial products the first or the second sensitivities do not exist at some point, such is the case for the standard Digital option at \( x = K \); even the payoff of the a plain vanilla European option is not twice differentiable at \( x = K \), yet the Gamma is well defined due to the regularizing effect of the Brownian motion (or the heat kernel) which gives sense to the expectation of a Dirac as a pointwise value of a probability density; in short the end result is well defined but the intermediate steps of AD are not.

For security derivatives computed by a Monte Carlo method, the computation of their sensitivities with respect to a parameter is most easily approximated by finite difference (also known as the shock method) thus requiring the reevaluation of the security with an incremented parameter. There are two problems with this method: it is imprecise when generalized to higher order derivatives and expensive for multidimensional problems with multiple parameters. The \( n^{th} \) derivative of a security with \( p \) parameters requires \( (n + 1)p \) evaluations; furthermore the choice of the perturbation parameter is tricky.

From a semi-analytical standpoint the most natural way to compute a sensitivity is the pathwise method described in Glasserman [11] which amounts to compute the derivative of the payoff for each simulation path. Unfortunately, this technique happens to be inefficient for certain types of payoffs including some often used in quantitative finance like Digitals or Barrier options. For instance, as it is not possible to obtain the Delta of a Digital Call that way (the derivative of the expectation of a Digital payoff is not equal to the expectation of the derivative of the Digital payoff, which in fact does not exist as a function), the pathwise method cannot evaluate the Gamma of a Call option in a standard Black-Scholes model. The pathwise derivative estimation is also called infinitesimal perturbation and there is a extensive literature on this subject; see for example Ho et al. [19], in Suri et al. [26] and in L’Ecuyer [21]. A general framework for some applications to option pricing is given in Glasserman [10]. We show a comparison to the two other well known methods i.e. likelihood ration method and Malliavin.

References


An Approach to Computing Discrete Adjoints for MPI-Parallelized Models Applied to the Ice Sheet System Model

Gilberto Pérez∗, Eric Larour†, Jean Utke‡, Anton Bovin§, Mathieu Morlighem¶

July 2016

1 Introduction

Constant monitoring of polar ice sheets through remote sensing, in particular since the advent of altimeter, radar, and gravity sensors such as ICESat-1, CryoSat, and GRACE, has created a large amount of data that has yet to find its way through Ice Sheet Models (ISMs) and hindcast reconstructions of polar ice sheet evolution. In particular, as evidenced by the wide discrepancy between ISMs involved in the SeaRISE and Ice2Sea projects [1, 2] significant improvements in modeled projections of the mass balance of polar ice sheets and their contribution to future sea-level rise has not resulted from the increase in availability of data, but rather from improvements in the type of physics captured in forward models. One reason for this is the lack of data assimilation capabilities embedded in the current generation of state-of-the-art ISMs. In the past 10 years, great strides have been made in improving model initialization by using steady-state model inversions of basal friction [3, 4, ?], ice rheology [5, 6] and bedrock elevation [7] among others. However, these approaches aim at improving our knowledge of poorly constrained input parameters and boundary conditions as long as the ice-flow regime is captured in a steady-state configuration. These inversions rely on analytically derived adjoint states of forward stress-balance or mass-transport models, but do not extend to transient regimes of ice flow.

Applications to transient models and long temporal time series such as the ICESat/CryoSat continuous altimetry record from 2003 to present-day, have been much more rare, and to our knowledge, are limited to a few studies. The main issue here precluding widespread application of transient data assimilation lies in the difficulty of deriving temporal adjoints of transient models. Actually, in many cases, a manual derivation of the adjoint state of a forward model is not possible, especially where ice-flow physics are not differentiable. For example in thermal transient models, where the melting-point is a physical constraint. This numerical issue can be mitigated with approaches such as: 1) ensemble runs, as in [8], where model runs compatible with observations are selected; 2) methods similar to the flux-correction methods implemented in [9] where boundary conditions are corrected in order to match time series of observations; 3) quasi-static approaches, where snapshot inversions are carried out in time, as in [10] and 4) sampling methods, which are computationally very expensive (each sample at the cost of one forward run). Though this is not an exhaustive list of all available methods, the main advantage of adjoint driven inversions is that it relies on the exact sensitivity of a forward model to its inputs, hence ensuring a physically meaningful inversion.

In fact, understanding sensitivities in a forward ice-flow model, which is required to physically constrain a temporal inversion, requires computation of derivatives of model outputs to model inputs. If such derivatives are approximated by finite-difference schemes, they are subject to the tradeoff between approximation and truncation errors for the perturbation, which is aggravated for higher-order derivatives. If the derivatives are computed using algorithmic differentiation (AD) [11], also known as automatic differentiation, then one can attain derivatives with machine precision provided the underlying program implementation of the numerical model is amenable to the application of an AD tool. In particular, this approach does not depend on the type of physics relied upon, and it is transparent to the model equations, provided each step of the overall software is differentiable. Applying AD to large-scale frameworks such as ISSM [4] is a difficult proposition, but one which enables significant improvements in the way models can be initialized [12], hindcast validated [4], and calibrated [13] towards better projections. Traditional approaches relying on Source-to-Source transformation have been developed, but for frameworks such as ISSM, which are C++ oriented, and highly parallelized, this type of approach breaks down. Our goal here is to demonstrate how the so-called operator-overloading AD approach can be implemented and validated for a framework such as ISSM, and what developments were necessary to make this capability operational. Our approach is discussed in section 2 of this manuscript, with section 3 describing the method validation as well as applications with ISSM. We discuss and conclude in the last section the applicability of such an approach to other frameworks, and on the opportunities these new developments afford for Cryosphere Science and data assimilation of remote sensing data in particular.

∗University of California, Irvine, gilberto.perez@uci.edu
†Jet Propulsion Laboratory, eric.larour@jpl.nasa.gov
‡Allstate Insurance Company, utkej1@gmail.com
§The University of Chicago
¶University of California, Irvine, mathieu.morlighem@uci.edu
2 Conclusion

We developed a new adjoint capability in ISSM, based on the ADOL-C framework and the AdjoinableMPI library, which is to our knowledge the first time this type of approach has been systematically applied to a software framework of this size and complexity. Despite the difficulties encountered rewriting the software, the overloaded approach is totally transparent to the user, which is critical given the size of the larger Cryosphere Science community that is not familiar with the adjoint work at all, and for which classic approaches such as source-to-source transformation have proven to be overly cumbersome. The flexibility of this approach allows in particular for quick turn-around in developing adjoint models of new parameterizations which are not easily hand derived. This is a massive advantage in that it opens this approach to the wider community. This, given the large amount of remote sensing data currently being collected and under-utilized, could prove paramount if we are to hindcast validate projections of sea-level rise. Further work is of course required to bring in additional observations such as gravity sensors, or radar stratigraphy observations, which will involve development of new cost functions, and scalability in 3D. Though this is complex in that it requires integrated resiliency and adjoint checkpointing schemes for long running transient modeling scenarios, our approach has proven flexible, and should lead to a brand new set of data assimilation capabilities that have already been available to other Earth Science communities for a long time. Indeed, by allowing temporal data assimilation for a large number of sensors and models, such as demonstrated here with the use of altimetry and radar sensors for mass transport and stress balance models respectively, ISSM paves the way for wider integration between the modeling and observational Cryosphere community.

References


1. Motivation and introduction.

In the numerical solution of nonlinear systems the situation may arise that the latter do not satisfy the smoothness requirements for traditional Newton-type methods to be applicable, but still conform to the weaker condition of composite piecewise smoothness (see [4, P. 91]).

For this constellation, an AD-based approach is to successively (piecewise) linearize the corresponding nonlinear maps, using AD-generated tangents, at basepoints $x_1, x_2, \ldots$ that are obtained as exact solutions of the linearizations, starting at an initial guess $x_0$ (see, e.g., [1]). Leaving aside further considerations, such as criteria for convergence to an exact solution of the underlying nonsmooth function, clearly, a minimal prerequisite for such a procedure to function is that in each step the derived piecewise linear systems are solvable, which would be trivially guaranteed if they were surjective.

The first part of the presentation will deal with this problem:

- We will define a notion of continuity for the behavior of a linearization under perturbations of the basepoint of the linearization.
- The AD-based linearization procedures of composite piecewise smooth functions presented in [1] conform to this continuity-notion for almost all basepoints (in a neighborhood of said basepoint).
- If the linearization at the initial basepoint is coherently oriented in that its linear/affine selection functions all have the same nonzero determinant sign, then there is a neighborhood $U$ of $x_0$ s.t. all linearizations at a basepoint $x' \in U$ are surjective and have only nonsingular selection functions albeit not necessarily with the same determinant signs.
- A counterexample will show that this is not generally the case without the coherent orientation at $x_0$.

The second part of the presentation is a little more "down to earth". Griewank has shown that the linearizations above, as well as any other continuous piecewise affine map, can be efficiently represented in the so-called abs-normal form ([1] and [2]). In [2] different algorithms were derived to solve the corresponding piecewise linear systems. Convergence criteria were proven. One generalized Newton method had particularly nice convergence properties in the sense that it generally needed single digit numbers of iterations. But these iterations were expensive, as they consisted of a full matrix inversion.

2010 Mathematics Subject Classification. Primary 57Q99, 52B99; Secondary 90C05.

Key words and phrases. Piecewise linear map, Surjectivity.
We will derive techniques for cheap and stable system updates, as well techniques to make use of sparsity and structural properties such as symmetry or bisymmetry. For band matrices these techniques boost the performance by several orders of magnitude, as will be demonstrated by some example problems. Some new convergence results will be added almost certainly.

As the algorithmic part is rather standard, we will, in this abstract, focus on presenting an outline of the argumentation of the surjectivity result, which is clearly the more general and more important part.

2. Surjectivity

Denote by \( B_r[p] \) the closed ball with radius \( r \), centered at \( p \). We call a piecewise linearization, or briefly: a linearization, any well defined procedure by which a continuous piecewise linear or piecewise affine function \( F : \mathbb{R}^n \to \mathbb{R}^n \) is generated. A special class of piecewise linearizations are linearizations at a basepoint \( x_0 \).

These notions are deliberately vague, as we, e.g., do not necessarily have to linearize another function with \( F \). A piecewise linear/piecewise affine map trivially is its own linearization. A basepoint may be a point on a curve, as well as the matrix representation of \( F \). The roles they play only matter insofar as they have to conform to:

**Definition 2.1.** We call a linearization procedure locally Lipschitz continuous at a basepoint \( x_0 \), if there exist \( \epsilon, L \in \mathbb{R}_{>0} \) such that for all \( \tilde{x}_0 \in B_\epsilon[x_0] \) and all \( x \in \mathbb{R}^n \setminus \{0\} \) we have for some \( \| \cdot \| \):

\[
\| F(x) - \tilde{F}(x) \| \leq L \cdot \| x_0 - \tilde{x}_0 \| \cdot \| F(x) \|,
\]

where \( F \) and \( \tilde{F} \) denote the linearizations at \( x_0 \) and \( \tilde{x}_0 \), respectively.

I.e. the perturbed values lie in a proportional ball around the unperturbed.

Below, we state the result and sketch the argumentation for piecewise linear maps. For piecewise affine maps the result is slightly weaker w.r.t. the radius of the ball. It follows from a localization argument (at vertices of the perturbed linearization, where it is locally piecewise linear).

**Theorem 2.2.** Let \( F : \mathbb{R}^n \to \mathbb{R}^n (n \geq 2) \) be a coherently oriented piecewise linear map, obtained by a linearization procedure at a basepoint \( x_0 \) that is locally Lipschitz continuous in \( B_\epsilon[x_0] \). Furthermore, let \( \epsilon := \min(1, \epsilon) \), where \( L \) is the Lipschitz constant of the Linearization. Then, if \( F \) is coherently oriented, for all \( \tilde{x}_0 \in B_\epsilon[x_0] \) the corresponding linearization \( \tilde{F} \) is surjective with nonsingular selection functions.

- We have \( F^{-1}(0) = \{0\} \) from the coherent orientation. Definition 2.1. then immediately gives the nonsingularity in a neighborhood, since singular matrices collapse linear subspaces to zero, which is a deformation that clearly does not conform to the Lipschitz condition.
- This leaves nonsurjective maps with nonsingular selection functions as possible counterexample.
- Denote by \( N \) the preimage of zero under \( F \). Define a map \( g : \mathbb{R}^n \setminus N \to S^{n-1} \setminus N \) by \( g(x) := \frac{x}{\|x\|_2} \) and a map \( \tilde{F} : S^{n-1} \setminus N \to S^{n-1} \) by \( \tilde{F}(g) = \frac{F(g)}{\|F(g)\|_2} \).
  Clearly, \( \tilde{F} \) is still continuous and image/preimage points correspond to the closed equivalence classes that are the rays in \( \mathbb{R}^n \). Also, it is surjective.
if and only if $F$ is surjective. If $\bar{F}$ is not surjective and its image not connected, then $S^{n-1} \setminus N$ cannot be connected, due to the continuity of $\bar{F}$, which implies $S^{n-1} \cap N = \emptyset$.

- This further reduces the range of possible counterexamples to nonsurjective maps with only nonsingular selection functions s.t. $F(\mathbb{R}^n) \cap S^{n-1}$ is connected. I.e. it has holes. Now a topological argument shows that the image of $\bar{F}$ cannot be derived from the image of a coherently oriented map by a continuous transformation, i.e. they can only be transformed into one another by cutting and regluing.

- The nonsurjective maps with only nonsingular selection functions s.t. $F(\mathbb{R}^n) \cap S^{n-1}$ is connected are homotopic to surjective maps with only nonsingular selection functions: Just contract the ”holes” in the image. I.e. they are the counterexample mentioned in the introduction.

References


Department of Mathematics, Humboldt University of Berlin, Germany

E-mail address: radons@math.hu-berlin.de
Discrete adjoint OpenFoam for surface and volume sensitivities

Arindam Sen*, Markus Towara† and Uwe Naumann‡

August 2016

1 Introduction

Adjoint methods for gradient based optimization are widely used in CFD applications. Open source CFD tools such as OpenFOAM are used by a vast user base in both industry and academia for various applications. The discrete adjoint version of OpenFoam using Algorithmic Differentiation [1] is developed that helps to compute efficient and accurate (up to machine precision) sensitivities, i.e. gradients of specific objective functions such as drag, lift, pressure loss etc. with respect to desired design variables [2] [3]. This discrete adjoint version is obtained by overloading all the basic mathematical operations using a custom data type with the aid of operator overloading tool dco/c++ [4]. Compared to continuous adjoints, this framework is flexible and easily adaptable. Also, because of the highly objected-oriented nature of the source code (in C++), source transformation tools like Tapenade [5] are not suitable. A black box adjoint usually incurs an unaffordable memory footprint. To overcome this limitation, several standard improvement techniques like binomial checkpointing [6], symbolic differentiation of the embedded linear iterative solver [7], reverse accumulation for steady state problems [8] were implemented. This framework is also extended to compute higher order adjoints. This framework was then tested on small to medium scale problems for a range of applications like ducted flows, external aerodynamics and conjugate heat transfer with qualitative validation against the continuous adjoint implementation [9].

2 Discrete adjoint OpenFOAM

The optimization problem can be represented as $J : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $J(\alpha) \rightarrow \min$, where $n$ denotes the number of input variables (design space) and $m$ denotes the number of output variables (dimension of the objective function). For the purpose of topology or shape optimization, $m = 1$ or $O(1)$ and $n \gg m$. For such a problem, the cost of obtaining the derivative, $\frac{\partial J}{\partial \alpha}$ using finite differences is $n \ast \text{Cost}(J)$ whereas that using adjoint is $m \ast \text{Cost}(J)$, thus becoming the obvious method of preference.

The basic method for obtaining a black box adjoint of a decoupled incompressible solver in OpenFOAM called *simpleFoam* using dco/c++ has previously been discussed [2].

A black box adjoint has limited usefulness in the context of relevant CFD problems pertaining to high memory footprint. Extension of this framework has been achieved by application of standard improvement techniques such as checkpointing and symbolic differentiation of the embedded linear iterative solver to obtain significant performance improvement [3]. Additionally for steady state problems, the application of reverse accumulation allows us to adjoin the last non-linear iteration step repeatedly by changing the inputs once the forward evaluations have converged contractively to a fixed point. This method allows the adjoint solution to usually converge faster and does not need additional memory for storing the checkpoints.

This framework is consistently applicable for a variation of flow characteristics like laminar/turbulent, steady/unsteady, compressible/incompressible and hence has a major advantage over it’s continuous counterpart which typically involves tedious mathematical derivations for different flow physics.

3 Results

The framework described above is tested on small to medium scale CFD applications. Fig 1. shows the volume sensitivities of an airduct design domain (Courtesy: Volkswagen AG), i.e derivatives of pressure loss between the inlet and outlet with respect to the porosity term $\alpha$. The design domain consists of 5 Mio. volume cells and the primal solution is obtained using *simpleFoam* for a Reynolds number 800. Fig. 2 shows the surface sensitivities of the DrivAer vehicle, i.e derivatives of drag force on the body of the car with respect to the normal surface displacement of the mesh points. The primal is solved on a design domain of 5.4 million cells with a smooth underbody using incompressible RANS for a Reynolds number of 30000.

*Corresponding Author, Software and tools for Computational Engineering (STCE), RWTH Aachen, sen@stce.rwth-aachen.de
†STCE, RWTH Aachen, towara@stce.rwth-aachen.de
‡STCE, RWTH Aachen, naumann@stce.rwth-aachen.de
Figure 1: Positive volume sensitivities of airduct design domain. Blue color depicting region where optimization could potentially remove material.

Figure 2: Drag sensitivities of DrivAer. To reduce drag, Blue: push in (fluid to solid), Red: pull out (solid to fluid).

4 Acknowledgements

This work was carried out under the AboutFlow project funded by the EU through the FP7-PEOPLE-2012-ITN, Grant agreement number:317006.

References


A Nonsmooth Optimization Case Study

Jean Utke\(^1\) and Sabrina Fiege\(^2\) and Andrea Walther\(^2\) and Kshitij Kulshreshtha\(^2\)

April 11, 2016

1 Introduction

This study has two goals. It investigates if the currently provided implementation of nonsmooth optimization in the LiPsMin\(^1\) solver is capable of handling a real life application that has not been specifically selected to fit the algorithmic approach of LiPsMin. Naturally, the application already benefits from computing derivatives with algorithmic differentiation, previously computed with CppAD\(^2\) and now being refitted to ADOL-C\(^3\) for use with LiPsMin.

The case study also aims at providing additional insight into the optimization problem, in particular, if the assumption that the existing non-contiguously differentiable artifacts are indeed mostly benign and - as assumed - do not severely change the optimization result but perhaps degrade the performance of a solver that does make the assumption of smoothly differentiable objectives and constraints.

At the current stage the preliminary technical work is almost complete but the actual study of the solver behavior is still ongoing and therefore any final results will be produced in time for the conference but are not available yet for this abstract.

2 The Use Case

The use case stems from a class of problems in the insurance space. Because the details represent proprietary information that cannot be disclosed, the use case is best characterized as a general nonlinear optimization problem in the usual notation

\[ x^* = \arg\min_{x \in X} f(x) \]

such that

\[ l \leq \begin{cases} x \\ c(x) \end{cases} \leq u \]

where both \( f \) and \( c \) can be nonlinear functions but for many solvers are assumed to be at least twice continuously differentiable, for example for the \texttt{nag.opt.nlp} solver in the NAG library\(^4\) or L-BFGS-B\(^5\). In practice however, there are optional (depending on the concrete problem instance) nonsmooth artifacts that can occur in the computation of \( f \) and \( c \) of the general form

\[ \text{if } a \circ b \text{ then } z=e_1 \text{ else } z=e_2 \]

Here, \( \circ \) is one of the usual comparison operators such as \( < \) or \( \leq \). The expression \( e_1 \) and \( e_2 \) are, by user inspection, asserted to be equal at the tie \( a = b \) and \( a, b, e_1, e_2 \) are mutually side-effect free. This then allows in principle for a transformation into abs-normal form\(^6\) to make it amenable to the algorithm implemented by the LiPsMin solver.

In addition to the above it should be noted that both \( f \) and \( c \) involve model evaluation computations, that is, a model resulting from a data fitting step is reevaluated at new data \( D \). This means as part of the practical use case one has \( f(x, D) \) and \( c(x, D) \) where one can view \( D \) as a passive parameter.

3 Observed Behavior

For the problem class we consider 47 test cases that all vary \( f, c, D \), the dimension of \( x \) and may include several alternative settings for \( l \) and \( u \) to be explored. Some of the test cases represent theoretical, reduced problems while about half of the cases are realistic practical problems. In the initial incarnation the problem had been formulated as a Lagrangian with fixed penalty parameters and was solved with L-BFGS-B. Even though the internal representation

\(^1\)Corresponding Author, Allstate Insurance Company, jutke@allstate.com
\(^2\)Universität Paderborn, Germany, [sfiege, andrea.walther, kshitij]@math.uni-paderborn.de
has now moved on to a more appropriate solver from the NAG library, the previous Lagrangian formulation is still available within the framework and serves as a comparison point for LiPsMin.

From the utilitarian point of view of a practical use in a company, the actual iteration counts are not of a large concern as long as the time the solver takes is practically feasible within the given work flow. On the other hand, an ever growing dimension of $x$ (currently at $\mathcal{O}(100)$) along with exploring tradeoffs for different constraint bounds, even when parallelized, suggests that the compute time for a solver does matter. A similarly dubious utilitarian argument is that a solution attained by a solver is considered good - in the absence of a proven global minimum - as long as the solution appears reasonable in the context of other sources of uncertainty particularly with respect to the data $\mathcal{D}$ and the quality of the models embedded in $f$ and $c$. While considerable efforts are made to control the quality of the models and the data, a new effort is undertaken by this study on the solver side.

With the earlier L-BFGS-B solver it was already observed that the solution for a significant number of the test cases was unstable in that for similar minimal values $f(x_1) \sim f(x_2)$ the solutions $x_1$ and $x_2$ vary significantly and also that certain problems took significantly more iterations. The reasons for this behavior could include

- inappropriate penalization (with L-BFGS-B)
- many similar local minima
- data $\mathcal{D}$ causing $f$ to be mostly flat in certain directions

but clearly also the fact that there are nonsmooth artifacts in the computation. Attempts to provide additional information by analyzing the Eigenvalues and Eigenvectors of the projected Hessian at the solution point and an exploration of the domain with random starting points are part of the toolbox. While not replacing these, an attempt that specifically targets the nonsmoothness is a complementary approach.

4 Implementation

The environment for determining the shape of the problem, ingesting the data, and performing the computation of $f$ and $c$ is entirely written in Python relying heavily on NumPy arrays and operations. The speedup NumPy affords through relying on compiled libraries is crucial and the up-front tracing overhead when using an AD tool must be amortized in the total compute time.

Given the authors of this abstract, it should be said that the initial competition between the tools ADOL-C and CppAD along with their respective python interface layers had been decided based on particular test case in which sparse updates to a large 2D NumPy array led to an unfavorable interaction with the ADOL-C location management that severely impacted performance. In the meantime, a variety of changes to the location management logic has made the problem disappear but as a consequence CppAD had become the tool of choice. The refitting for LiPsMin/ADOL-C made it necessary to mirror certain functionality from CppAD.

4.1 Locating switches in the user code

The very prerequisite of both, bringing a computation into abs-normal form as well as plainly avoiding the overhead of repeated retracing sweeps is the ability to locate in the user code for $f$ and $c$ the relatively few active conditional expressions that actually do change value during the solver iterations. There are of course many conditional expressions that remain passive by type or are active by type but never change value or are difficult to transform into the tool specific conditional expression format because of side effects so that a wholesale transformation of all such conditions is not appropriate. The mechanism introduced into CppAD for this particular use case and now adapted to ADOL-C implies the following steps:

1. assume the trace is created by evaluating at point $x_0$
2. a forward sweep detects a change in the value of a conditional expression at evaluation point $x_k$ and throws an exception containing the particular operation number $o$ obtained by counting all the operations stepped through the forward sweep up to this point
3. the exception handling unwinds the call stack through the Python layer to the caller of the forward sweep that then catches the exception extracts $o$
4. a trace is restarted at point $x_0$, all operations written to the trace are counted until the count equals $o$ and another exception is thrown.
5. the exception handling unwinds the call stack all the way to Python and the user is able to see the stack frame directly related to the comparison operation in Python that will change value at point $x_k$ and therefore has to be transformed into an ADOL-C conditional assignment.

Using this facility the trace can be created avoiding the need to recreated it and passing gradient computations on to sweeps over the trace performed by compiled C/C++ code. Timing experiments with CppAD show that fortunately these sweeps are fast enough to amortize the initial slow trace and to be in the overall execution on par with handwritten (in Python) and partially Cython-compiled derivative code found in the original framework.
4.2 Conditional Assignment with Equality

Even though the argument against floating point equality comparison is well received, there are practical cases, e.g. for initialization, where conditional assignments for all operators (as supported in CppAD) are desired, while ADOL-C offered support only for conditions of the form $e_c > 0$ with some actively typed expression $e_c$. For full parity we extended ADOL-C conditional assignment by a variant permitting $e_c \geq 0$.

4.3 Conditional Assignments and Python Arrays

There is an implied interpretation for certain operators in which one operand is a scalar and the other one is a NumPy array. One the other hand, the conditional assignment Python interface for both CppAD and ADOL-C are defined in terms of active scalars. Therefore, coercions to both, the active type and also the shape matching that of the involved array operands have to be performed. Fortunately these coercions can be encapsulated in a Python wrapper for the conditional assignments so as to keep most of the Python code type and shape agnostic.

4.4 Conditional Assignments and the abs-normal form

Under the user assertion that the expressions $e_1$ and $e_2$ (see Sect[2]) are equal at the tie $a = b$ and we represent $e_c = a - b$ for $<$ and $e_c = b - a$ for $>$. Then we rewrite the branch as

$$ z = \frac{e_c + |e_c|}{2e_c} e_1 + \frac{e_c - |e_c|}{2e_c} e_2 \quad \text{for} \quad e_c \neq 0. $$

Clearly, this does not cover the tie points but fortunately at least with the iteration points taken by the current solvers we never hit the tie exactly. Whether the evaluations chosen by LiPsMin will hit a tie exactly remains to be seen.

4.5 Python for LiPsMin

In order to make the LiPsMin functionality accessible for the Python framework we implemented a simple Python layer for the solver interface based on Boost.Python because unlike other interface layers it comprehensively handles the passing of NumPy arrays.

5 Work to be completed for AD2016

Because LiPsMin initially did not support any constraints we explore the addition of box constraints on the optimization variables in order to have a reasonable comparison with L-BFGS-B assuming identical settings for the penalty factors. With the aforementioned prerequisites in place we aim at presenting at AD2016 the following additional aspects for this case study:

- iteration counts and timings in comparison with L-BFGS-B, counts for intermediate nonsmooth variables
- analysis of success/failure cases
- conclusions for the practical usability of the solver for the use case and recommendations for future development.

References


