The Art of Differentiating Computer Programs

An Introduction to Algorithmic Differentiation

SOFTWARE • ENVIRONMENTS • TOOLS

The SIAM series on Software, Environments, and Tools focuses on the practical implementation of computational methods and the high performance aspects of scientific computation by emphasizing in-demand software, computing environments, and tools for computing. Software technology development issues such as current status, applications and algorithms, mathematical software, software tools, languages and compilers, computing environments, and visualization are presented.

Software, Environments, and Tools

Uwe Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*

C. T. Kelley, *Implicit Filtering*

Jeremy Kepner and John Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*

Jeremy Kepner, *Parallel MATLAB for Multicore and Multinode Computers*

Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*

Gérard Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*

Bo Einarsson, editor, *Accuracy and Reliability in Scientific Computing*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval, Second Edition*

Craig C. Douglas, Gundolf Haase, and Ulrich Langer, *A Tutorial on Elliptic PDE Solvers and Their Parallelization*

Louis Komzsik, *The Lanczos Method: Evolution and Application*

Bard Ermentrout, *Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students*

V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Waśniewski, and P. Yalamov, *LAPACK95 Users' Guide*

Stefan Goedecker and Adolfy Hoisie, *Performance Optimization of Numerically Intensive Codes*

Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*

Lloyd N. Trefethen, *Spectral Methods in MATLAB*

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition*

Michael W. Berry and Murray Browne, *Understanding Search Engines: Mathematical Modeling and Text Retrieval*

Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*

R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*

Randolph E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 8.0*

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*

Greg Astfalk, editor, *Applications on Advanced Architecture Computers*

Roger W. Hockney, *The Science of Computer Benchmarking*

Françoise Chaitin-Chatelin and Valérie Frayssé, *Lectures on Finite Precision Computations*

The Art of Differentiating Computer Programs

An Introduction to Algorithmic Differentiation

Uwe Naumann
RWTH Aachen University
Aachen, Germany

siam
Society for Industrial and Applied Mathematics
Philadelphia

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data
Naumann, Uwe, 1969-
The art of differentiating computer programs : an introduction to algorithmic differentiation/ Uwe Naumann.
p. cm. – (Software, environments, and tools)
Includes bibliographical references and index.
ISBN 978-1-611972-06-1
1. Computer programs. 2. Automatic differentiations. 3. Sensitivity theory (Mathematics) I. Title.
QA76.76.A98N38 2011
003′.3–dc23 2011032262

Partial royalties from the sale of this book are placed in a fund to help students attend SIAM meetings and other SIAM-related activities. This fund is administered by SIAM, and qualified individuals are encouraged to write directly to SIAM for guidelines.

To Ines, Pia, and Antonia

Contents

Preface

"How sensitive are the values of the outputs of my computer program with respect to changes in the values of the inputs? How sensitive are these first-order sensitivities with respect to changes in the values of the inputs? How sensitive are the second-order sensitivities with respect to changes in the values of the inputs? ..."

Computational scientists, engineers, and economists as well as quantitative analysts in computational finance tend to ask these questions on a regular basis. They write computer programs in order to simulate diverse real-world phenomena. The underlying mathematical models often depend on a possibly large number of (typically unknown or uncertain) parameters. Values for the corresponding inputs of the numerical simulation programs can, for example, be the result of (typically error-prone) observations and measurements. If very small perturbations in these uncertain values yield large changes in the values of the outputs, then the feasibility of the entire simulation becomes questionable. Nobody should make decisions based on such highly uncertain data.

Quantitative information about the extent of this uncertainty is crucial. First- and higher-order sensitivities of outputs of numerical simulation programs with respect to their inputs (also first and higher derivatives) form the basis for various approximations of uncertainty. They are also crucial ingredients of a large number of numerical algorithms ranging from the solution of (systems of) nonlinear equations to optimization under constraints given as (systems of) partial differential equations. This book describes a set of techniques for modifying the semantics of numerical simulation programs such that the desired first and higher derivatives can be computed accurately and efficiently. Computer programs implement algorithms. Consequently, the subject is known as Algorithmic (also Automatic) Differentiation (AD).

AD provides two fundamental modes. In *forward mode*, a *tangent-linear* version of the original program is built. The sensitivities of all outputs of the program with respect to its inputs can be computed at a computational cost that is proportional to the number of inputs. The computational complexity is similar to that of finite difference approximation. At the same time, the desired derivatives are computed with machine accuracy. Truncation is avoided.

Reverse mode yields an *adjoint* program that can be used to perform the same task at a computational cost that is proportional to the number of outputs. For example, in large-scale nonlinear optimization a scalar objective that is returned by the given computer program can depend on a very large number of input parameters. The adjoint program allows for the computation of the gradient (the first-order sensitivities of the objective with respect to all parameters) at a small constant multiple $\mathcal{R}$ (typically between 3 and 30) of the cost of running the original program. It outperforms gradient accumulation routines that are based

on finite differences or on tangent-linear code as soon as the size of the gradient exceeds $\mathcal{R}$. The ratio $\mathcal{R}$ plays a very prominent role in the evaluation of the quality of derivative code. It will reappear several times in this book.

The generation of tangent-linear and adjoint code is the main topic of this introduction to *The Art of Differentiating Computer Programs* by AD. Repeated applications of forward and reverse modes yield higher-order tangent-linear and adjoint code. Two ways of implementing AD are presented. Derivative code compilers take a source transformation approach in order to realize the semantic modification. Alternatively, run time support libraries can be developed that use operator and function overloading based on a redefined floating-point data type to propagate tangent-linear as well as adjoint sensitivities. Note that

AD differentiates what you implement![1]

Many successful applications of AD are described in the proceedings of, five international conferences [10, 11, 13, 18, 19]. The standard book on the subject by Griewank and Walther [36] covers a wide range of basic, as well as advanced, topics in AD. Our focus is different. We aim to present a textbook style introduction to AD for undergraduate and graduate students as well as for practitioners in computational science, engineering, economics, and finance. The material was developed to support courses on "Computational Differentiation" and "Derivatives Code Compilers" for students of Computational Engineering Science, Mathematics, and Computer Science at RWTH Aachen University. Project-style exercises come with detailed hints on possible solutions. All software is provided as open source. In particular, we present a fully functional derivative code compiler (`dcc`) for a (very) limited subset of C/C++. It can be used to generate tangent-linear and adjoint code of arbitrary order by reapplication to its own output. Our run time support library `dco` provides a better language coverage at the expense of less efficient derivative code. It uses operator and function overloading in C++. Both tools form the basis for the ongoing development of production versions that are actively used in a number of collaborative projects among scientists and engineers from various application areas.

Except for relatively simple cases, the differentiation of computer programs is not automatic despite the existence of many reasonably mature AD software packages.[2] To reveal their full power, AD solutions need to be integrated into existing numerical simulation software. Targeted application of AD tools and intervention by educated users is crucial. We expect AD to be become truly "automatic" at some time in the (distant) future. In particular, the automatic generation of optimal (in terms of robustness and efficiency) adjoint versions of large-scale simulation code is one of the great open challenges in the field of High-Performance Scientific Computing. With this book, we hope to contribute to a better understanding of AD by a wider range of potential users of this technology. Combine it with the book of Griewank and Walther [36] for a comprehensive introduction to the state of the art in the field.

There are several reasonable paths through this book that depend on your specific interests. Chapter 1 motivates the use of differentiated computer programs in the context of methods for the solution of systems of nonlinear equations and for nonlinear programming. The drawbacks of closed-form symbolic differentiation and finite difference approximations are discussed, and the superiority of adjoint over tangent-linear code is shown if the

[1] Which occasionally differs from what you *think* you implement!

[2] See `www.autodiff.org`.

number of inputs exceeds the number of outputs significantly. The generation of tangent-linear and adjoint code by forward and reverse mode AD is the subject of Chapter 2. If you are a potential user of first-order AD exclusively, then you may proceed immediately to the relevant sections of Chapter 5, covering the use of `dcc` for the generation of first derivative code. Otherwise, read Chapter 3 to find out more about the generation of second- or higher-order tangent-linear and adjoint code. The remaining sections in Chapter 5 illustrate the use of `dcc` for the partial automation of the corresponding source transformation. Prospective developers of derivative code compilers should not skip Chapter 4. There, we relate well-known material from compiler construction to the task of differentiating computer programs. The scanner and parser generators `flex` and `bison` are used to build a compiler front-end that is suitable for both single- and multipass compilation of derivative code. Further relevant material, including hints on the solutions for all exercises, is collected in the Appendix.

The supplementary website for this book, http://www.siam.org/se22, contains sources of all software discussed in the book, further exercises and comments on their solutions (growing over the coming years), links to further sites on AD, and errata.

In practice, the programming language that is used for the implementation of the original program accounts for many of the problems to be addressed by users of AD technology. Each language deserves to be covered by a separate book. The given computing infrastructure (hardware, native compilers, concurrency/parallelism, external libraries, handling data, i/o, etc.) and software policies (level of robustness and safety, version management) may complicate things even further. Nevertheless, AD is actively used in many large projects, each of them posing specific challenges. The collection of these issues and their structured presentation in the form of a book can probably only be achieved by a group of AD practitioners and is clearly beyond the scope of this introduction.

Let us conclude these opening remarks with comments on the book's title, which might sound vaguely familiar. While its scope is obviously much narrower than that of the classic by Knuth [45], the application of AD to computer programs still deserves to be called an "art." Educated users are crucial prerequisites for robust and efficient AD solutions in the context of large-scale numerical simulation programs. "In AD details really do matter."[3] With this book, we hope to set the stage for many more "artists" to enter this exciting field.

Uwe Naumann
July 2011

[3] Quote from one of the anonymous referees.

Acknowledgments

I could probably fill a few pages acknowledging the exceptional role of my family in my (professional) life including the evolution of this book. You know what I am talking about.

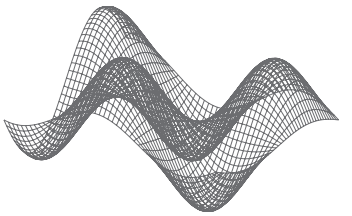
My wife Ines, who is by far the better artist, has helped me with the cover art. My own attempts never resulted in a drawing that adequately reflects the intrinsic joy of differentiating computer programs. I did contribute the code fragment, though.

I am grateful to Markus Beckers, Michael Förster, Boris Gendler, Johannes Lotz, Andrew Lyons, Viktor Mosenkis, Jan Riehme, Niloofar Safiran, Michel Schanen, Ebadollah Varnik, and Claudia Yastremiz for (repeatedly) proofreading the manuscript. Any remaining shortcomings should be blamed on them.

Three anonymous referees provided valuable feedback on various versions of the manuscript.

Last, but not least, I would like thank my former Ph.D. supervisor Andreas Griewank for seeding my interest in AD. As a pioneer in this field, he has always been keen on promoting AD technology by using various techniques. One of them is music.

Optimality[4]



Music: Think of Fool's Garden's "Lemon Tree"
Lyrics: Uwe Naumann
Vocals: Be my guest

I'm sittin' in front of the computer screen.
Newton's second iteration is what I've just seen.
It's not quite the progress that I would expect
from a code such as mine—no doubt it must be perfect!
Just the facts are not supportive, and I wonder ...

My linear solver is state of the art.
It does not get better wherever I start.
For differentiation is there anything else?
Perturbing the inputs—can't imagine this fails.
I pick a small Epsilon, and I wonder ...

I wonder how, but I still give it a try.
The next change in step size is bound to fly.
'Cause all I'd like to see is simply optimality.
Epsilon, in fact, appears to be rather small.
A factor of ten should improve it all.
'Cause all I'd like to see is nearly optimality.

A DAD ADADA DAD ADADA DADAD.

[4]Originally presented in Nice, France on April 8, 2010 at the *Algorithmic Differentiation, Optimization, and Beyond* meeting in honor of Andreas Griewank's 60th birthday.

A few hours later my talk's getting rude.
The sole thing descending seems to be my mood.
How can guessing the Hessian only take this much time?
N squared function runs appear to be the crime.
The facts support this thesis, and I wonder ...

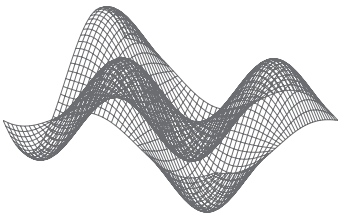
Isolation due to KKT.
Isolation—why not simply drop feasibility?

The guy next door's been sayin' again and again:
An adjoint Lagrangian might relieve my pain.
Though I don't quite believe him, I surrender.

I wonder how but I still give it a try.
Gradients and Hessians in the blink of an eye.
Still all I'd like to see is simply optimality.
Epsilon itself has finally disappeared.
Reverse mode AD works, no matter how weird,
and I'm about to see local optimality.

Yes, I wonder, I wonder ...

I wonder how but I still give it a try.
Gradient and Hessians in the blink of an eye.
Still all I'd like to see ...
I really need to see ...
now I can finally see my cherished optimality :-)



Chapter 1

Motivation and Introduction

The computation of derivatives plays a central role in many numerical algorithms. First- and higher-order sensitivities of selected outputs of numerical simulation programs with respect to certain inputs as well as projections of the corresponding derivative tensors may be required. Often the computational effort of these algorithms is dominated by the run time and the memory requirement of the computation of derivatives. Their accuracy may have a dramatic effect on both convergence behavior and run time of the underlying iterative numerical schemes. We illustrate these claims with simple case studies in Section 1.1, namely the solution of systems of nonlinear equations using the Newton method in Section 1.1.1 and basic first- and second-order nonlinear programming algorithms in Section 1.1.2. The use of derivatives with numerical libraries is demonstrated in the context of the NAG Numerical Library as a prominent representative for a number of similar commercial and noncommercial numerical software tools in Section 1.1.3.

This first chapter aims to set the stage for the following discussion of Algorithmic Differentiation (AD) for the accurate and efficient computation of first and higher derivatives. Traditionally, numerical differentiation has been performed manually, possibly supported by symbolic differentiation capabilities of modern computer algebra systems, or derivatives have been approximated by finite difference quotients. Neither approach turns out to be a serious competitor for AD. Manual differentiation is tedious and error-prone, while finite differences are often highly inefficient and potentially inaccurate. These two techniques are discussed briefly in Section 1.2 and Section 1.3, respectively.

1.1 Motivation: Derivatives for ...

Numerical simulation enables computational scientists and engineers to study the behavior of various kinds of real-world systems in ways that are impossible (or at least extremely difficult) in reality. The quality of the results depends largely on the quality of the underlying mathematical model $F : \mathbb{R}^n \to \mathbb{R}^m$. Computer programs are developed to simulate the functional dependence of one or more objectives $\mathbf{y} \in \mathbb{R}^m$ on a potentially very large number of parameters $\mathbf{x} \in \mathbb{R}^n$. For a given set of input parameters, the corresponding values of the objectives can be obtained by a single run of the simulation program as $\mathbf{y} = F(\mathbf{x})$. This

simulation of the studied real-world system can be extremely useful. However, it leaves various questions unanswered.

One of the simpler open questions is about sensitivities of the objective with respect to the input parameters with the goal of quantifying the change in the objective values for slight (infinitesimal) changes in the parameters. Suppose that the values of the parameters are defined through measurements within the simulated system (for example, an ocean or the atmosphere). The accuracy of such measurements is less important for small sensitivities as inaccuracies will not translate into significant variations of the objectives. Large sensitivities, however, indicate critical parameters whose inaccurate measurement may yield dramatically different results. More accurate measuring devices are likely to be more costly. Even worse, an adequate measuring strategy may be infeasible due to excessive run time or other hard constraints. Mathematical remodeling may turn out to be the only solution.

Sensitivity analysis is one of many areas requiring the *Jacobian* matrix of F,

$$\nabla F = \nabla F(\mathbf{x}) \equiv \left(\frac{\partial y_j}{\partial x_i} \right)_{i=0,\ldots,n-1}^{j=0,\ldots,m-1},$$

whose rows contain the sensitivities of the outputs y_j, $j = 0,\ldots,m-1$, of the numerical simulation $\mathbf{y} = F(\mathbf{x})$ with respect to the input parameters x_i, $i = 0,\ldots,n-1$. Higher derivative tensors including the *Hessian* of F,

$$\nabla^2 F = \nabla^2 F(\mathbf{x}) \equiv \left(\frac{\partial^2 y_j}{\partial x_i \partial x_k} \right)_{i,k=0,\ldots,n-1}^{j=0,\ldots,m-1},$$

are used in corresponding higher-order methods. This book is based on C/C++ as the underlying programming language; hence, vectors are indexed starting from zero instead of one. In the following, *highlighted terminology* is used without definition. Formal explanations are given in the subsequent chapters.

1.1.1 ... Systems of Nonlinear Equations

The solution of systems of nonlinear equations $F(\mathbf{x}) = 0$, where $F : \mathbb{R}^n \to \mathbb{R}^n$, is a fundamental requirement of many numerical algorithms, ranging from nonlinear programming via the numerical solution of nonlinear partial differential equations (PDEs) to PDE-constrained optimization. Variants of the Newton algorithm are highly popular in this context.

The basic version of the Newton algorithm (see Algorithm 1.1) solves $F(\mathbf{x}) = 0$ iteratively for $k = 0, 1, \ldots$ and for a given starting point $\mathbf{x}^0$ as

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \left(\nabla F(\mathbf{x}^k) \right)^{-1} \cdot F(\mathbf{x}^k).$$

The Newton step $d\mathbf{x}^k \equiv -\left(\nabla F(\mathbf{x}^k) \right)^{-1} \cdot F(\mathbf{x}^k)$ is obtained as the solution of the Newton system of linear equations

$$\nabla F(\mathbf{x}^k) \cdot d\mathbf{x}^k = -F(\mathbf{x}^k)$$

at each iteration. A good starting value $\mathbf{x}^0$ is crucial for reaching the desired convergence behavior. Convergence is typically defined by some norm of the residual undercutting a

Algorithm 1.1 Newton algorithm for solving the nonlinear system $F(\mathbf{x}) = 0$.

In:

→ implementation of the residual $\mathbf{y}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$F : \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{y} = F(\mathbf{x})$

→ implementation of the Jacobian $A \equiv \nabla F(\mathbf{x})$ of the residual at the current point $\mathbf{x}$:
$F' : \mathbb{R}^n \to \mathbb{R}^{n \times n}$, $A = F'(\mathbf{x})$

→ solver for computing the Newton step $d\mathbf{x} \in \mathbb{R}^n$ as the solution of the linear Newton system $A \cdot d\mathbf{x} = -\mathbf{y}$:
$s : \mathbb{R}^n \times \mathbb{R}^{n \times n} \to \mathbb{R}^n$, $d\mathbf{x} = s(\mathbf{y}, A)$

→ starting point: $\mathbf{x} \in \mathbb{R}^n$

→ upper bound on the norm of the residual $\|F(\mathbf{x})\|$ at the approximate solution: $\epsilon \in \mathbb{R}$

Out:

← approximate solution of the nonlinear system $F(\mathbf{x}) = 0$: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

```
1: y = F(x)
2: while ||y|| > ε do
3:     A = F'(x)
4:     dx = s(y, A)
5:     x ← x + dx
6:     y = F(x)
7: end while
```

given bound. Refer to [44] for further details on the Newton algorithm. A basic version without error handling is shown in Algorithm 1.1. Convergence is assumed.

The computational cost of Algorithm 1.1 is dominated by the accumulation of the Jacobian $A \equiv \nabla F$ in line 3 and by the solution of the Newton system in line 4. The quality of the Newton step $d\mathbf{x}$ depends on the accuracy of the Jacobian ∇F. Traditionally, ∇F is approximated using finite difference quotients as shown in Algorithm 1.2, where $\mathbf{e}^i$ denotes the ith Cartesian basis vector in $\mathbb{R}^n$, that is,

$$\mathbf{e}^i = \left(e^i_j\right)_{j=0,\ldots,n-1} \equiv \begin{cases} 1 & i = j, \\ 0 & \text{otherwise.} \end{cases}$$

The value of the residual is computed at no extra cost when using *forward* or *backward* finite differences, to be discussed in further detail in Section 1.3.

In Algorithm 1.2, a single evaluation of the residual at the current point $\mathbf{x}$ in line 1 is succeeded by n evaluations at perturbed points in line 4. The components of $\mathbf{x}$ are perturbed individually in line 3. The columns of the Jacobian are approximated separately in lines 5–7, yielding a computational cost of $O(n) \cdot Cost(F)$, where $Cost(F)$ denotes the cost of a single

Algorithm 1.2 Jacobian accumulation by (forward) finite differences in the context of the Newton algorithm for solving the nonlinear system $F(\mathbf{x}) = 0$.

In:

→ implementation of the residual $\mathbf{y} = (y_k)_{k=0,\ldots,n-1}$ at the current point $\mathbf{x} \in \mathbb{R}^n$: $F : \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{y} = F(\mathbf{x})$

→ current point: $\mathbf{x} \in \mathbb{R}^n$

→ perturbation: $\delta \in \mathbb{R}$

Out:

← residual at the current point: $\mathbf{y} = F(\mathbf{x}) \in \mathbb{R}^n$

← approximate Jacobian of the residual at the current point: $A = (a_{k,i})_{k,i=0,\ldots,n-1} \approx \nabla F(\mathbf{x}) \in \mathbb{R}^{n \times n}$

Algorithm:

1: $\mathbf{y} = F(\mathbf{x})$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad \tilde{\mathbf{x}} \leftarrow \mathbf{x} + \delta \cdot \mathbf{e}^i$
4: $\quad \tilde{\mathbf{y}} = F(\tilde{\mathbf{x}})$
5: $\quad$ **for** $k = 0$ to $n-1$ **do**
6: $\quad\quad a_{k,i} \leftarrow (\tilde{y}_k - y_k)/\delta$
7: $\quad$ **end for**
8: **end for**

evaluation of the residual. Refer to Section 1.3 for further details on finite differences as well as on alternative approaches to their implementation. Potential sparsity of the Jacobian should be exploited to reduce the computational effort as discussed in Section 2.1.3.

The inherent inaccuracy of the approximation of the Jacobian by finite differences may have a negative impact on the convergence of the Newton algorithm. Exact (up to machine accuracy) Jacobians can be computed by the *tangent-linear mode* of AD as described in Section 2.1. The corresponding part of the Newton algorithm is replaced by Algorithm 1.3. Columns of the Jacobian are computed in line 3 after setting $\mathbf{x}^{(1)}$ equal to the corresponding Cartesian basis vector in line 2. The value of the residual $\mathbf{y}$ is computed in line 3 by the given implementation of the tangent-linear residual $F^{(1)}$ at almost no extra cost. Details on the construction of $F^{(1)}$ are discussed in Chapter 2. The superscript $^{(1)}$ is used to denote first-order tangent-linear versions of functions and variables. This notation will be found advantageous for generalization in the context of higher derivatives in Chapter 3.

Example 1.1 For $\mathbf{y} = F(\mathbf{x})$ defined as

$$y_0 = 4 \cdot x_0 \cdot (x_0^2 + x_1^2),$$
$$y_1 = 4 \cdot x_1 \cdot (x_0^2 + x_1^2)$$

Algorithm 1.3 Jacobian accumulation by tangent-linear mode AD in the context of the Newton algorithm for solving the nonlinear system $F(\mathbf{x}) = 0$.

In:

→ implementation of the tangent-linear residual $F^{(1)}$ for computing the residual $\mathbf{y} = (y_k)_{k=0,\ldots,n-1} \equiv F(\mathbf{x})$ and its directional derivative $\mathbf{y}^{(1)} = (y^{(1)})_{k=0,\ldots,n-1} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in the tangent-linear direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n$, $\left(\mathbf{y}, \mathbf{y}^{(1)}\right) = F^{(1)}\left(\mathbf{x}, \mathbf{x}^{(1)}\right)$

→ current point: $\mathbf{x} \in \mathbb{R}^n$

Out:

← residual at the current point: $\mathbf{y} = F(\mathbf{x}) \in \mathbb{R}^n$

← Jacobian of the residual at the current point:
$A = (a_{k,i})_{k,i=0,\ldots,n-1} = \nabla F(\mathbf{x}) \in \mathbb{R}^{n \times n}$

Algorithm:

```
1: for i = 0 to n − 1 do
2:    x^(1) ← e^i
3:    (y, y^(1)) = F^(1)(x, x^(1))
4:    for k = 0 to n − 1 do
5:       a_{k,i} ← y_k^(1)
6:    end for
7: end for
```

and starting from $\mathbf{x}^T = (1,1)$, a total of 21 Newton iterations are performed by the code in Section C.1.2 to drive the norm of the residual $\mathbf{y}$ below 10^{-9}:

1	$x_0 = 0.666667$	$x_1 = 0.666667$	$\|\mathbf{y}\| = 11.3137$
2	$x_0 = 0.444444$	$x_1 = 0.444444$	$\|\mathbf{y}\| = 3.35221$
3	$x_0 = 0.296296$	$x_1 = 0.296296$	$\|\mathbf{y}\| = 0.993247$
...			
20	$x_0 = 0.000300729$	$x_1 = 0.000300729$	$\|\mathbf{y}\| = 1.03849e-09$
21	$x_0 = 0.000200486$	$x_1 = 0.000200486$	$\|\mathbf{y}\| = 3.07701e-10$ ∎

The solution of the linear Newton system by a direct method (such as Gaussian LU factorization or Cholesky LL^T factorization if the Jacobian is symmetric positive definite as in the previous example) is an $O\left(n^3\right)$ algorithm. Hence, the overall cost of the Newton method is dominated by the solution of the linear system in addition to the accumulation of the Jacobian. The computational complexity of the direct linear solver can be decreased by exploiting possible sparsity of the Jacobian [24].

Alternatively, iterative solvers can be used to approximate the Newton step. Matrix-free implementations of Krylov subspace methods avoid the accumulation of the full Jacobian. Consider, for example, the Conjugate Gradient (CG) algorithm [39] in Algorithm 1.4

Algorithm 1.4 Matrix-free CG algorithm for computing the Newton step in the context of the Newton algorithm for solving the nonlinear system $F(\mathbf{x}) = 0$.

In:

→ implementation of the tangent-linear residual $F^{(1)}$ for computing the residual $\mathbf{y} \equiv F(\mathbf{x})$ and its directional derivative $\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in the tangent-linear direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n$, $(\mathbf{y}, \mathbf{y}^{(1)}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$

→ starting point for the Newton step: $d\mathbf{x} \equiv \mathbf{x}^{(1)} \in \mathbb{R}^n$

→ upper bound on the norm of the residual $\| - \mathbf{y} - \nabla F(\mathbf{x}) \cdot d\mathbf{x}\|$ at the approximate solution for the Newton step: $\epsilon \in \mathbb{R}$

Out:

← approximate solution for the Newton step: $d\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: $\mathbf{x}^{(1)} \leftarrow d\mathbf{x}$
2: $(\mathbf{y}, \mathbf{y}^{(1)}) \leftarrow F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
3: $\mathbf{p} \leftarrow -\mathbf{y} - \mathbf{y}^{(1)}$
4: $\mathbf{r} \leftarrow \mathbf{p}$
5: **while** $\mathbf{r} \geq \epsilon$ **do**
6: $\quad \mathbf{x}^{(1)} \leftarrow \mathbf{p}$
7: $\quad (\mathbf{y}, \mathbf{y}^{(1)}) \leftarrow F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
8: $\quad \alpha \leftarrow \mathbf{r}^T \cdot \mathbf{r} \;/\; (\mathbf{p}^T \cdot \mathbf{y}^{(1)})$
9: $\quad d\mathbf{x} \leftarrow d\mathbf{x} + \alpha \cdot \mathbf{p}$
10: $\quad \mathbf{r}_{\text{prev}} \leftarrow \mathbf{r}$
11: $\quad \mathbf{r} \leftarrow \mathbf{r} - \alpha \cdot \mathbf{y}^{(1)}$
12: $\quad \beta \leftarrow \mathbf{r}^T \cdot \mathbf{r} \;/\; (\mathbf{r}_{\text{prev}}^T \cdot \mathbf{r}_{\text{prev}})$
13: $\quad \mathbf{p} \leftarrow \mathbf{r} + \beta \cdot \mathbf{p}$
14: **end while**

for symmetric positive definite systems. It aims to drive during each Newton iteration the norm of the residual $-\mathbf{y} - \nabla F \cdot d\mathbf{x}$ toward zero. Note that only the function value (line 3) and projections of the Jacobian (Jacobian-vector products in lines 3, 8, and 11) are required. These directional derivatives are delivered efficiently by a single run of the implementation of the tangent-linear residual $F^{(1)}$ in lines 2 and 7, respectively. The exact solution is obtained in infinite precision arithmetic after a total of n steps. Approximations of the solution in floating-point arithmetic can often be obtained much sooner provided that a suitable *preconditioner* is available [58]. For notational simplicity, Algorithm 1.4 assumes that no preconditioner is required.

Example 1.2 Solutions of systems of nonlinear equations play an important role in various fields of Computational Science and Engineering. They result, for example, from the

discretization of nonlinear PDEs used to model many real-world phenomena. We consider a very simple example for illustration.

The two-dimensional Solid Fuel Ignition (SFI) problem (also known as the Bratu problem) from the MINPACK-2 test problem collection [5] simulates a thermal reaction process in a rigid material. It is given by the elliptic PDE

$$\Delta y - \lambda \cdot e^y = 0, \tag{1.1}$$

where $y = y(x_0, x_1)$ is computed over some bounded domain $\Omega \subsetneq \mathbb{R}^2$ with boundary $\Gamma \subsetneq \mathbb{R}^2$ and Dirichlet boundary conditions $y(x_0, x_1) = g(x_0, x_1)$ for $(x_0, x_1) \in \Gamma$. For simplicity, we focus on the unit square $\Omega = [0, 1]^2$ and we set

$$g(x_0, x_1) = \begin{cases} 1 & \text{if } x_0 = 1, \\ 0 & \text{otherwise.} \end{cases}$$

We use finite differences as a basic discretization method. Its aim is to replace the differential

$$\Delta y \equiv \frac{\partial^2 y}{\partial x_0^2} + \frac{\partial^2 y}{\partial x_1^2}$$

with a set of algebraic equations, thus transforming (1.1) into a system of nonlinear equations that can be solved by Algorithm 1.1.

Let Ω be discretized using central finite differences with step size $h = 1/s$ in both the x_0 and x_1 directions. The second derivative with respect to x_0 at some point (x_0^i, x_1^j) (for example, (x_0^2, x_1^2) in Figure 1.1 where $s = 4$) is approximated based on the finite difference approximation of the first derivative with respect to x_0 at points $a = (x_0^i - h/2, x_1^j)$ and $b = (x_0^i + h/2, x_1^j)$. Similarly, the second derivative with respect to x_1 at the same point

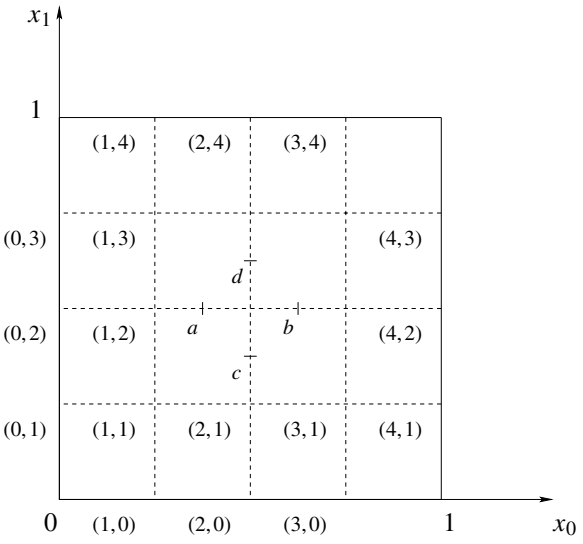


Figure 1.1. *Finite difference discretization of the SFI equation.*

is approximated based on the finite difference approximation of the first derivative with respect to x_1 at points $c = (x_0^i, x_1^j - h/2)$ and $d = (x_0^i, x_1^j + h/2)$. As the result of

$$\frac{\partial y(x_0,x_1)}{\partial x_0}(a) \approx \frac{y_{i,j} - y_{i-1,j}}{h}, \quad \frac{\partial y(x_0,x_1)}{\partial x_0}(b) \approx \frac{y_{i+1,j} - y_{i,j}}{h},$$

and

$$\frac{\partial^2 y(x_0,x_1)}{\partial x_0^2}(x_0^i, x_1^j) \approx \frac{\frac{\partial y(x_0,x_1)}{\partial x_0}(b) - \frac{\partial y(x_0,x_1)}{\partial x_0}(a)}{h},$$

we have

$$\frac{\partial^2 y(x_0,x_1)}{\partial x_0^2}(x_0^i, x_1^j) \approx \frac{y_{i+1,j} - 2 \cdot y_{i,j} + y_{i-1,j}}{h^2}.$$

Similarly,

$$\frac{\partial^2 y(x_0,x_1)}{\partial x_1^2}(x_0^i, x_1^j) \approx \frac{y_{i,j+1} - 2 \cdot y_{i,j} + y_{i,j-1}}{h^2}$$

follows from

$$\frac{\partial^2 y(x_0,x_1)}{\partial x_1^2}(x_0^i, x_1^j) \approx \frac{\frac{\partial y(x_0,x_1)}{\partial x_1}(d) - \frac{\partial y(x_0,x_1)}{\partial x_1}(c)}{h}$$

with

$$\frac{\partial y(x_0,x_1)}{\partial x_1}(c) \approx \frac{y_{i,j} - y_{i,j-1}}{h}, \quad \frac{\partial y(x_0,x_1)}{\partial x_1}(d) \approx \frac{y_{i,j+1} - y_{i,j}}{h}.$$

Consequently, the system of nonlinear equations to be solved becomes

$$-4 \cdot y_{i,j} + y_{i+1,j} + y_{i-1,j} + y_{i,j+1} + y_{i,j-1} = h^2 \cdot \lambda \cdot e^{y_{i,j}}$$

for $i, j = 1, \ldots, s-1$. Discretization of the boundary conditions yields $y_{s,j} = 1$ and $y_{i,0} = y_{0,j} = y_{i,s} = 0$ for $i, j = 1, \ldots, s-1$. A possible implementation is the following:

```
void f(int s, double **y, double l, double** r) {
  for (int i=1;i<s;i++)
    for (int j=1;j<s;j++)
      r[i][j]=y[i+1][j]+y[i-1][j]+y[i][j+1]+y[i][j-1]
              -4*y[i][j]-l/(s*s)*exp(y[i][j]);
}
```

Both y and r cover the entire discretized unit square, that is, $y \in \mathbb{R}^{(s+1)\times(s+1)}$ as well as $r \in \mathbb{R}^{(s+1)\times(s+1)}$. Both derivatives of boundary values as well as derivatives with respect to boundary values turn out to be equal to zero. For $s = 3$, we get

$$\begin{aligned}
-4 \cdot y_{1,1} + y_{2,1} + y_{0,1} + y_{1,2} + y_{1,0} &= h^2 \cdot \lambda \cdot e^{y_{1,1}} \\
-4 \cdot y_{1,2} + y_{2,2} + y_{0,2} + y_{1,3} + y_{1,1} &= h^2 \cdot \lambda \cdot e^{y_{1,2}} \\
-4 \cdot y_{2,1} + y_{3,1} + y_{1,1} + y_{2,2} + y_{2,0} &= h^2 \cdot \lambda \cdot e^{y_{2,1}} \\
-4 \cdot y_{2,2} + y_{3,2} + y_{1,2} + y_{2,3} + y_{2,1} &= h^2 \cdot \lambda \cdot e^{y_{2,2}}
\end{aligned}$$

Table 1.1. *Run time statistics for the SFI problem. The solution is computed by the standard Newton algorithm and by a matrix-free implementation of the Newton-CG algorithm. For different resolutions of the mesh (defined by the step size s) and for varying levels of accuracies of the Newton iteration (defined by ϵ), we list the run time of the Newton algorithm t, the number of Newton iterations i, and the number of Jacobian-vector products computed by calling the tangent-linear routine t1_f.*

		Newton			Newton-CG		
s	ϵ	t	i	t1_f	t	i	t1_f
25	10^{-5}	6,0	8	4.608	0,0	8	281
30	10^{-5}	24,6	8	6.720	0,0	8	333
35	10^{-5}	71,4	8	9.248	0,0	8	384
25	10^{-10}	6,8	9	5.184	0,0	9	528
30	10^{-10}	27,1	9	7.569	0,0	9	629
35	10^{-10}	79,6	9	10.404	0,1	9	724
300	10^{-5}	-	-	-	31,0	8	2.963
300	10^{-8}	-	-	-	44,4	9	4.905
300	10^{-10}	-	-	-	53,3	9	5.842

and hence the following system of nonlinear equations:

$$\begin{aligned} -4 \cdot y_{1,1} + y_{2,1} + y_{1,2} - h^2 \cdot \lambda \cdot e^{y_{1,1}} &= 0 \\ -4 \cdot y_{1,2} + y_{2,2} + y_{1,1} - h^2 \cdot \lambda \cdot e^{y_{1,2}} &= 0 \\ -4 \cdot y_{2,1} + y_{1,1} + y_{2,2} + 1 - h^2 \cdot \lambda \cdot e^{y_{2,1}} &= 0 \\ -4 \cdot y_{2,2} + y_{1,2} + y_{2,1} + 1 - h^2 \cdot \lambda \cdot e^{y_{2,2}} &= 0. \end{aligned}$$

The SFI problem will be used within various exercises throughout this book. Its solution using Algorithm 1.1 is discussed in Section 1.4.2.

The Jacobian of the residual used in Section 1.4.2 turns out to be symmetric positive definite. Hence, Algorithm 1.4 can be used for the solution of the linear Newton system. Run time statistics for Algorithm 1.1 using a direct linear solver as well as for a matrix-free implementation based on Algorithm 1.4 are shown in Table 1.1. We start from $y(x_0, x_1) = 10$ for $(x_0, x_1) \in \Omega \setminus \Gamma$ and $\lambda = 0.5$. The CG solver is converged to the same accuracy ϵ as the enclosing Newton iteration. Its substantial superiority over direct solvers for the given problem is mostly due to the matrix-free implementation and the missing preconditioning. Moreover, sparsity of the Jacobian is not exploited within the direct solver. Exploitation of sparsity speeds up the Jacobian accumulation by performing fewer evaluations of the tangent-linear model and reduces both the memory requirement and the run time of the solution of the linear Newton system when using a direct method. Refer to [24] for details on sparse direct solvers. Existing software includes MUMPS [3], PARDISO [55], SuperLU [23], and UMFPACK [22]. ∎

1.1.2 ... Nonlinear Programming

The following example has been designed to compare the performance of finite difference approximation of first and second derivatives with that of derivative code that computes

exact values in the context of basic unconstrained optimization algorithms. Adjoint code exceeds the efficiency of finite differences by a factor at the order of n. In many cases, this factor makes the difference between derivative-based methods being applicable to large-scale optimization problems or not.

Consider the nonlinear programming problem

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}),$$

where, for example, the objective function

$$f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2\right)^2 \tag{1.2}$$

is implemented as follows:

```
void f(int n, double *x, double &y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}
```

For educational reasons, we intentionally avoid the use of the more compact C++ notation `y+=x[i]*x[i]`. The clean separation of left- and right-hand sides of assignments will prove advantageous in Chapters 2 and 3.

The function in (1.2) has a global minimum at $\mathbf{x} = 0$. We use this problem to illustrate issues that arise in derivative-based optimization methods. Our objective is not to cover the state of the art in nonlinear optimization; refer to [50] for a survey of such techniques.

We apply basic line search methods to the given implementation of the objective. Such methods compute iterates

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \cdot B_k^{-1} \cdot \nabla f(\mathbf{x}^k) \tag{1.3}$$

for some suitable starting value $\mathbf{x}^0 = (x_i^0)_{i=0,\ldots,n-1}$ and with a step length $\alpha_k > 0$, where $\nabla f(\mathbf{x}^k)$ denotes the *gradient* (the transposed single-row Jacobian) of f at the current iterate. Simple first- (B_k is equal to the identity I_n in $\mathbb{R}^n$) and second-order (B_k is equal to the Hessian $\nabla^2 f(\mathbf{x}^k)$ of f at point $\mathbf{x}^k$) methods are discussed below. We aim to find a local minimizer by starting at $x_i^0 = 1$ for $i = 0,\ldots,n-1$.

Steepest Descent Algorithm

In the simplest case, (1.3) becomes

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \cdot \nabla f(\mathbf{x}^k).$$

The step length $\alpha_k > 0$ is, for example, chosen by recursive bisection on α_k starting from $\alpha_k = 1$ $(0.5, 0.25, \ldots)$ and such that a decrease in the objective value is ensured. This simple method is known as the *steepest descent algorithm*. It is stated formally in Algorithm 1.5, where it is assumed that a suitable α can always be found. Refer to [50] for details on exceptions.

Algorithm 1.5 Steepest descent algorithm for solving the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of the objective $y \in \mathbb{R}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$f : \mathbb{R}^n \to \mathbb{R}$, $y = f(\mathbf{x})$

→ implementation of f' for computing the objective $y \equiv f(\mathbf{x})$ and its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$ at the current point $\mathbf{x}$:
$f' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n$, $(y, \mathbf{g}) = f'(\mathbf{x})$

→ starting point: $\mathbf{x} \in \mathbb{R}^n$

→ upper bound on gradient norm $\|\mathbf{g}\|$ at the approximate minimal point: $\epsilon \in \mathbb{R}$

Out:

← approximate minimal value of the objective: $y \in \mathbb{R}$

← approximate minimal point: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

```
1:  repeat
2:     (y, g) = f'(x)
3:     if ||g|| > ε then
4:        α ← 1
5:        ỹ ← y
6:        while ỹ ≥ y do
7:           x̃ ← x − α · g
8:           ỹ = f(x̃)
9:           α ← α/2
10:       end while
11:       x ← x̃
12:    end if
13: until ||g|| ≤ ε
```

For a given implementation of f, the only nontrivial ingredient of Algorithm 1.5 is the computation of the gradient in line 2. For the given simple example, hand-coding of

$$\nabla f(\mathbf{x}) = \left(4 \cdot x_i \cdot \sum_{j=0}^{n-1} x_j^2 \right)_{i=0,\ldots,n-1}$$

is certainly an option. This situation will change for more complex objectives implemented as computer programs with many thousand lines of source code. Typically, the efficiency of handwritten derivative code is regarded as close to optimal. While this is a reasonable assumption in many cases, it still depends very much on the author of the derivative code.

Algorithm 1.6 Gradient approximation by (forward) finite differences in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

$\rightarrow$ implementation of the objective function: $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $y = f(\mathbf{x})$

$\rightarrow$ current point: $\mathbf{x} \in \mathbb{R}^n$

$\rightarrow$ perturbation: $\delta \in \mathbb{R}$

Out:

$\leftarrow$ objective at the current point: $y = f(\mathbf{x}) \in \mathbb{R}$

$\leftarrow$ approximate gradient of the objective at the current point: $\mathbf{g} \equiv (g_i)_{i=0,\ldots,n-1} \approx \nabla f(\mathbf{x}) \in \mathbb{R}^n$

Algorithm:

1: $y = f(\mathbf{x})$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad \tilde{\mathbf{x}} \leftarrow \mathbf{x} + \delta \cdot \mathbf{e}^i$
4: $\quad \tilde{y} = f(\tilde{\mathbf{x}})$
5: $\quad g_i \leftarrow (\tilde{y} - y)/\delta$
6: **end for**

Moreover, hand-coding may be infeasible within the time frame allocated to the project. Debugging is likely to occupy the bigger part of the development time. Hence, we aim to build up a set of rules that will allow us to automate the generation of derivative code to the greatest possible extent. All of these rules can be applied manually. Our ultimate goal, however, is the development of corresponding software tools in order to make this process less tedious and less error-prone.

As in Section 1.1.1, the gradient can be approximated using finite difference quotients (see Algorithm 1.6), provided that the computational complexity of $O(n) \cdot Cost(f)$ remains feasible. Refer to Section 1.3 for further details on finite differences. This approach has two major disadvantages. First, the approximation may be poor. Second, a minimum of $n+1$ function evaluations are required. If, for example, a single function evaluation takes one minute on the given computer architecture, and if $n = 10^6$ (corresponding, for example, to a temperature distribution in a very coarse-grain discretization of a global three-dimensional atmospheric model), then a single evaluation of the gradient would take almost two years. Serious climate simulation would not be possible.

The computational complexity is not decreased when using tangent-linear AD as outlined in Algorithm 1.7. Nevertheless, the improved accuracy of the computed gradient may lead to faster convergence of the steepest descent algorithm.

Large-scale and long-term climate simulations are performed by many researchers worldwide. A single function evaluation is likely to run for much longer than one minute,

Algorithm 1.7 Gradient accumulation by tangent-linear mode AD in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of the tangent-linear objective function $f^{(1)}$ for computing the objective $y \equiv f(\mathbf{x})$ and its directional derivative $y^{(1)} \equiv \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in the tangent-linear direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$f^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$, $(y, y^{(1)}) = f^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$

→ current point: $\mathbf{x} \in \mathbb{R}^n$

Out:

← objective at the current point: $y = f(\mathbf{x}) \in \mathbb{R}$

← gradient of the objective at the current point:
$\mathbf{g} \equiv (g_i)_{i=0,\ldots,n-1} = \nabla f(\mathbf{x}) \in \mathbb{R}^n$

Algorithm:

1: **for** $i = 0$ to $n - 1$ **do**
2: $\quad \mathbf{x}^{(1)} \leftarrow \mathbf{e}^i$
3: $\quad (y, y^{(1)}) = f^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
4: $\quad g_i \leftarrow y^{(1)}$
5: **end for**

even on the latest high-performance computer architectures. It may take hours or days to perform climate simulations at physically meaningful spatial discretization levels and over relevant time intervals. Typically only a few runs are feasible. The solution to this problem comes in the form of various flavors of so-called *adjoint* methods. In particular adjoint AD allows us to generate for a given implementation of f an adjoint program that **computes the gradient ∇f at a computational cost of $O(1) \cdot$ Cost(f)**. As opposed to finite differences and tangent-linear AD, adjoint AD thus makes the computational cost independent of n. It enables large-scale sensitivity analysis as well as high-dimensional nonlinear optimization and uncertainty quantification for practically relevant problems in science and engineering. This observation is worth highlighting even at this early stage, and it serves as motivation for the better part of the remaining chapters in this book.

Algorithm 1.8 illustrates the use of an adjoint code for f. The adjoint objective is called only once (in line 2). We use the subscript (1) to denote adjoint functions and variables. The advantages of this notation will become obvious in Chapter 3, in the context of higher-order adjoints.

Table 1.2 summarizes the impact of the various differentiation methods on the run time of the steepest descent algorithm when applied to our example problem in (1.2). These results were obtained on a standard Linux PC running the GNU C++ compiler with optimization level 3, which will henceforth be referred to as the **reference platform**. The numbers illustrate the superiority of adjoint over both tangent-linear code and finite difference

Algorithm 1.8 Gradient accumulation by adjoint mode AD in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of the adjoint objective function $f_{(1)}$ for computing the objective $y \equiv f(\mathbf{x})$ and the product $\mathbf{x}_{(1)} \equiv y_{(1)} \cdot \nabla f(\mathbf{x})$ of its gradient at the current point $\mathbf{x} \in \mathbb{R}^n$ with a factor $y_{(1)} \in \mathbb{R}$:
$f_{(1)} : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}^n$, $(y, \mathbf{x}_{(1)}) = f_{(1)}(\mathbf{x}, y_{(1)})$

→ current point: $\mathbf{x} \in \mathbb{R}^n$

Out:

← objective at the current point: $y = f(\mathbf{x}) \in \mathbb{R}$

← gradient of the objective at the current point: $\mathbf{g} = \nabla f(\mathbf{x}) \in \mathbb{R}^n$

Algorithm:

1: $y_{(1)} \leftarrow 1$
2: $(\mathbf{y}, \mathbf{x}_{(1)}) = f_{(1)}(\mathbf{x}, y_{(1)})$
3: $\mathbf{g} \leftarrow \mathbf{x}_{(1)}$

Table 1.2. *Run time of the steepest descent algorithm (in seconds and starting from* $\mathbf{x} = 1$*). The gradient (of size n) is approximated by central finite differences (FD; see Section* 1.3*) or computed with machine accuracy by a tangent-linear code (TLC; see Section* 2.1*) or an adjoint code (ADC; see Section* 2.2*). The tangent-linear and adjoint codes are generated automatically by the derivative code compiler (*`dcc`*) (see Chapter* 5*).*

n	FD	TLC	ADC
100	13	8	< 1
200	47	28	1
300	104	63	2
400	184	113	2.5
500	284	173	3
1000	1129	689	6

approximations in terms of the overall computational effort that is dominated by the cost of the gradient evaluation. Convergence of the steepest descent algorithm is defined as the L_2-norm of the gradient falling below 10^{-9}. The steepest descent algorithm is expected to perform a large number of iterations (with potentially very small step sizes) to reach this high level of accuracy. Similar numbers of iterations (over $3 \cdot 10^5$) are performed independently of the method used for the evaluation of the gradient. As expected, the step size α_k is reduced to values below 10^{-4} to reach convergence while ensuring strict descent in the objective function value.

Newton Algorithm

Second-order methods based on the Newton algorithm promise faster convergence in the neighborhood of the minimum by taking into account second derivative information. We consider the Newton algorithm discussed in Section 1.1.1 extended by a local line search to determine α_k for $k = 0, 1, \ldots$ in (1.3) as

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \cdot \left(\nabla^2 f(\mathbf{x}^k)\right)^{-1} \cdot \nabla f(\mathbf{x}^k).$$

As in Section 1.1.1, the Newton method is applied to find a stationary point of f by solving the nonlinear system $\nabla f = 0$. The Newton step

$$d\mathbf{x}^k \equiv -\left(\nabla^2 f(\mathbf{x}^k)\right)^{-1} \cdot \nabla f(\mathbf{x}^k)$$

is obtained as the solution of the linear Newton system

$$\nabla^2 f(\mathbf{x}^k) \cdot d\mathbf{x}^k = -\nabla f(\mathbf{x}^k)$$

at each iteration. If $\mathbf{x}^k$ is far from a solution, then sufficient descent in the residual can be obtained using a local line search to determine α_k such that the L_2-norm of the residual at the next iterate is minimized, that is, the scalar nonlinear optimization problem

$$\min_{\alpha_k} ||f(\mathbf{x}^k - \alpha_k \cdot d\mathbf{x}^k))||_2$$

needs to be solved. The first and, potentially also required, second derivatives of the objective with respect to α_k can be computed efficiently using the methods discussed in this book. Alternatively, a simple recursive bisection algorithm similar to that used in the steepest descent method can help to improve the robustness of the Newton method.

A formal description of the Newton algorithm for unconstrained nonlinear optimization is given in Algorithm 1.9. Again, convergence is assumed. The computational cost is dominated by the accumulation in lines 1, 3, and 14 of gradient and Hessian and by the solution of the linear Newton system in line 4. Both the gradient and the Hessian should be accurate in order to ensure the expected convergence behavior. Approximation by finite differences may not be good enough due to an inaccurate Hessian in particular.

An algorithmic view of the (second-order) finite difference method for approximating the Hessian is given in Algorithm 1.10. Refer to Section 1.3 for details on first- and second-order finite differences as well as for a description of alternative approaches to their implementation. The shortcomings of finite difference approximation become even more apparent in the second-order case. The inaccuracy is likely to become more significant due to the limitations of floating-point arithmetic. Moreover, $O(n^2)$ function evaluations are required for the approximation of the Hessian.

The first problem can be overcome by applying tangent-linear AD to Algorithm 1.7, yielding Algorithm 1.11. Each of the n calls of the *second-order tangent-linear* function $G^{(1)}$, where $G \equiv f'$ is defined as in Algorithm 1.7, involves n calls of the tangent-linear code. The overall computational complexity of the Hessian accumulation adds up to $O(n^2) \cdot Cost(f)$. Both the gradient and the Hessian are obtained with machine accuracy. Symmetry of the Hessian is exploited in neither Algorithm 1.10 nor Algorithm 1.11.

Algorithm 1.9 Newton algorithm for solving the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of the objective $y \in \mathbb{R}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$f : \mathbb{R}^n \to \mathbb{R}$, $y = f(\mathbf{x})$

→ implementation of the differentiated objective function f' for computing the objective $y \equiv f(\mathbf{x})$ and its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$ at the current point $\mathbf{x}$:
$f' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n$, $(y, \mathbf{g}) = f'(\mathbf{x})$

→ implementation of the differentiated objective function f'' for computing the objective y, its gradient $\mathbf{g}$, and its Hessian $H \equiv \nabla^2 f(\mathbf{x})$ at the current point $\mathbf{x}$:
$f'' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n\times n}$, $(y, \mathbf{g}, H) = f''(\mathbf{x})$

→ solver to determine the Newton step $d\mathbf{x} \in \mathbb{R}^n$ as the solution of linear Newton system $H \cdot d\mathbf{x} = -\mathbf{g}$:
$s : \mathbb{R}^n \times \mathbb{R}^{n\times n} \to \mathbb{R}^n$, $d\mathbf{x} = s(\mathbf{g}, H)$

→ starting point: $\mathbf{x} \in \mathbb{R}^n$

→ upper bound on the gradient norm $\|\mathbf{g}\|$ at the approximate solution: $\epsilon \in \mathbb{R}$

Out:

← approximate minimal value: $y \in \mathbb{R}$

← approximate minimal point: $\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: $(y, \mathbf{g}) = f'(\mathbf{x})$
2: **while** $\|\mathbf{g}\| > \epsilon$ **do**
3: $\quad (y, \mathbf{g}, H) = f''(\mathbf{x})$
4: $\quad d\mathbf{x} = s(\mathbf{g}, H)$
5: $\quad \alpha \leftarrow 1$
6: $\quad \tilde{y} \leftarrow y$
7: $\quad \tilde{\mathbf{x}} \leftarrow \mathbf{x}$
8: $\quad$ **while** $\tilde{y} \geq y$ **do**
9: $\quad\quad \tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} - \alpha \cdot d\mathbf{x}$
10: $\quad\quad \tilde{y} = f(\tilde{\mathbf{x}})$
11: $\quad\quad \alpha \leftarrow \alpha/2$
12: $\quad$ **end while**
13: $\quad \mathbf{x} \leftarrow \tilde{\mathbf{x}}$
14: $\quad (y, \mathbf{g}) = f'(\mathbf{x})$
15: **end while**

Algorithm 1.10 Hessian approximation by (forward) finite differences in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of f' for computing the objective $y \equiv f(\mathbf{x})$ and its approximate gradient $\mathbf{g} = (g_j)_{j=0,\ldots,n-1} \approx \nabla f(\mathbf{x})$ at the current point $\mathbf{x} \in \mathbb{R}^n$ as defined in Algorithm 1.6: $f' : \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}^n$, $(y, \mathbf{g}) = f'(\mathbf{x})$

→ current point: $\mathbf{x} \in \mathbb{R}^n$

→ perturbation: $\delta \in \mathbb{R}$

Out:

← objective function value at the current point: $y = f(\mathbf{x})$

← approximate gradient of the objective at the current point: $\mathbf{g} \approx \nabla f(\mathbf{x}) \in \mathbb{R}^n$

← approximate Hessian of the objective at the current point: $H = (h_{j,i})_{j,i=0,\ldots,n-1} \approx \nabla^2 f(\mathbf{x}) \in \mathbb{R}^{n\times n}$

Algorithm:

1: $(y, \mathbf{g}) = f'(\mathbf{x})$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad \tilde{\mathbf{x}} \leftarrow \mathbf{x} + \delta \cdot \mathbf{e}^i$
4: $\quad (\tilde{y}, \tilde{\mathbf{g}}) = f'(\tilde{\mathbf{x}})$
5: $\quad$ **for** $j = 0$ to $n-1$ **do**
6: $\quad\quad h_{j,i} \leftarrow (\tilde{g}_j - g_j)/\delta$
7: $\quad$ **end for**
8: **end for**

Substantial savings in the computational cost result from performing the gradient computation in adjoint mode. The savings are due to each of the n calls of the *second-order adjoint* function $G^{(1)}$, where $G \equiv f'$ is defined as in Algorithm 1.8, now involving merely a single call of the adjoint code. The overall computational complexity becomes $O(n) \cdot Cost(f)$ instead of $O(n^2) \cdot Cost(f)$.

The tangent-linear version of an adjoint code is referred to as *second-order adjoint* code. It can be used to compute both the gradient as $w \cdot \nabla f(\mathbf{x})$ as well as the projections of the Hessian in direction $\mathbf{v} \in \mathbb{R}^n$ as $w \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{v}$ by setting $w = 1$. A *second-order tangent-linear* code (tangent-linear version of the tangent-linear code) can be used to compute single entries of the Hessian as $\mathbf{u}^T \cdot \nabla^2 f(\mathbf{x}) \cdot \mathbf{w}$ by letting $\mathbf{u}$ and $\mathbf{w}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$. Consequently, the computational complexity of Hessian approximation using finite differences or the second-order tangent-linear model is $O(n^2) \cdot Cost(f)$. Second-order adjoint code delivers the Hessian at a computational cost of $O(n) \cdot Cost(f)$. Savings at the order of n are likely to make the difference between second-order methods being applicable or not. Refer to Table 1.3 for numerical results that support these findings. Note that further combinations of tangent-linear and adjoint AD are possible when computing second derivatives. Refer to Chapter 3 for details.

Algorithm 1.11 Hessian accumulation by second-order AD in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

$\rightarrow$ implementation of the tangent-linear version $G^{(1)}$ of the differentiated objective function $G \equiv f'$ defined in Algorithm 1.7 (yielding second-order tangent-linear mode AD) or Algorithm 1.8 (yielding second-order adjoint mode AD) for computing the objective $y \equiv f(\mathbf{x})$, its directional derivative $y^{(1)} \equiv \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in the tangent-linear direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$, its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$, and its second directional derivative $\mathbf{g}^{(1)} = (g_j^{(1)})_{j=0,\ldots,n-1} \equiv \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in direction $\mathbf{x}^{(1)}$ at the current point $\mathbf{x} \in \mathbb{R}^n$: $G^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n$, $(y, y^{(1)}, \mathbf{g}, \mathbf{g}^{(1)}) = G^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$

$\rightarrow$ current point: $\mathbf{x} \in \mathbb{R}^n$

Out:

$\leftarrow$ objective function value at the current point: $y = f(\mathbf{x})$

$\leftarrow$ gradient at the current point: $\mathbf{g} = \nabla f(\mathbf{x}) \in \mathbb{R}^n$

$\leftarrow$ Hessian at the current point: $H = (h_{j,i})_{j,i=0,\ldots,n-1} = \nabla^2 f(\mathbf{x}) \in \mathbb{R}^{n\times n}$

Algorithm:

1: **for** $i = 0$ to $n-1$ **do**
2: $\quad \mathbf{x}^{(1)} \leftarrow \mathbf{e}^i$
3: $\quad (y, y^{(1)}, \mathbf{g}, \mathbf{g}^{(1)}) = G^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
4: $\quad$ **for** $j = 0$ to $n-1$ **do**
5: $\quad\quad h_{j,i} \leftarrow g_j^{(1)}$
6: $\quad$ **end for**
7: **end for**

As already observed in Section 1.1.1, the solution of the linear Newton system by a direct method yields a computational complexity at the order of $O(n^3)$. Hence, the overall cost of the Newton method applied to the given implementation of (1.2) is dominated by the solution of the linear system in columns SOFD, SOTLM, and SOADM of Table 1.3. Exploitation of possible sparsity of the Hessian can reduce the computational complexity as discussed in Chapter 3. The run time of the Hessian accumulation is higher when using the second-order tangent-linear model (column SOTLM) or, similarly, a second-order finite difference approximation (column SOFD). Use of the second-order adjoint model reduces the computational cost (column SOADM). The Hessian may become indefinite very quickly when using finite difference approximation.

Matrix-free implementations of the Conjugate Gradients solver avoid the accumulation of the full Hessian. Note that in Algorithm 1.12 only the gradient $\mathbf{g}$ (line 3) and projections of the Hessian $\mathbf{g}^{(1)}$ (lines 3, 8, and 11) are required. Both are delivered efficiently by a single run of $G^{(1)}$ (in line 2 and 7, respectively). Again, preconditioning has been omitted for the sake of notational simplicity. If a second-order adjoint code is used

Table 1.3. *Run time of the Newton algorithm (in seconds and starting from* $\mathbf{x} = 1$*). The gradient and Hessian of the given implementation of* (1.2) *are approximated by second-order central finite differences (SOFD; see Section* 1.3*) or computed with machine accuracy by a second-order tangent-linear (SOTLC; see Section* 3.2*) or adjoint (SOADC; see Section* 3.3*) code. The Newton system is solved using a Cholesky factorization that dominates both the run time and the memory requirement for increasing n due to the relatively low cost of the function evaluation itself. The last column shows the run times for a matrix-free implementation of a Newton–Krylov algorithm that uses the CG algorithm to approximate the Newton step based on the second-order adjoint model. As expected, the algorithm scales well beyond the problem sizes that could be handled by the other three approaches. A run time of more than* 1 *second is observed only for* $n \geq 10^5$.

n	SOFD	SOTLC	SOADC	SOADC (CG)
100	< 1	< 1	< 1	< 1
200	2	1	< 1	< 1
300	7	3	1	< 1
400	17	9	4	< 1
500	36	21	10	< 1
1000	365	231	138	< 1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
10^5	$> 10^4$	$> 10^4$	$> 10^4$	1

(see Algorithm 1.11), then **the computational complexity of evaluating the gradient and a Hessian-vector product is $O(1) \cdot \mathbf{Cost}(f)$**. We take this result as further motivation for an in-depth look into the generation of first- and higher-order adjoint code in the following chapters.

Nonlinear Programming with Constraints

Practically relevant optimization problems are most likely subject to constraints, which are often nonlinear. For example, the solution may be required to satisfy a set of nonlinear PDEs as in many data assimilation problems in the atmospheric sciences. Discretization of the PDEs yields a system of nonlinear algebraic equations to be solved by the solution of the optimization problem.

The core of many algorithms for constrained optimization is the solution of the equality-constrained problem

$$\min\ o(\mathbf{x}) \quad \text{subject to } c(\mathbf{x}) = 0,$$

where both the objective $o : \mathbb{R}^n \to \mathbb{R}$ and the constraints $c : \mathbb{R}^n \to \mathbb{R}^m$ are assumed to be twice continuously differentiable within the domain Ω. The first-order *Karush–Kuhn–Tucker* (KKT) conditions yield the system

$$\begin{bmatrix} \nabla o(\mathbf{x}) - (\nabla c(\mathbf{x}))^T \cdot \Lambda \\ c(\mathbf{x}) \end{bmatrix} = 0$$

of $n+m$ nonlinear equations in $n+m$ unknowns $\mathbf{x} \in \mathbb{R}^n$ and $\Lambda \in \mathbb{R}^m$. The Newton algorithm can be used to solve the KKT system subject to the following conditions: The Jacobian of

Algorithm 1.12 CG algorithm for computing the Newton step in the context of the unconstrained nonlinear programming problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$.

In:

→ implementation of the tangent-linear version $G^{(1)}$ of the differentiated objective function $G \equiv f'$ defined in Algorithm 1.7 (yielding second-order tangent-linear mode AD) or Algorithm 1.8 (yielding a potentially matrix-free implementation based on second-order adjoint mode AD) for computing the objective $y \equiv f(\mathbf{x})$, its directional derivative $y^{(1)} \equiv \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in the tangent-linear direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$, its gradient $\mathbf{g} \equiv \nabla f(\mathbf{x})$, and its second directional derivative $\mathbf{g}^{(1)} = (g_j^{(1)})_{j=0,\ldots,n-1} \equiv \nabla^2 f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ in direction $\mathbf{x}^{(1)}$ at the current point $\mathbf{x} \in \mathbb{R}^n$:
$G^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n$, $(y, y^{(1)}, \mathbf{g}, \mathbf{g}^{(1)}) = G^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$

→ starting point: $d\mathbf{x} \in \mathbb{R}^n$

→ upper bound on the norm of the residual $\| -\mathbf{g} - H \cdot d\mathbf{x}\|$, where $H \equiv \nabla^2 f(\mathbf{x})$, at the approximate solution for the Newton step: $\epsilon \in \mathbb{R}$

Out:

← approximate solution for the Newton step: $d\mathbf{x} \in \mathbb{R}^n$

Algorithm:

1: $\mathbf{x}^{(1)} \leftarrow d\mathbf{x}$
2: $(y, y^{(1)}, \mathbf{g}, \mathbf{g}^{(1)}) = G^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
3: $\mathbf{p} \leftarrow -\mathbf{g} - \mathbf{g}^{(1)}$
4: $\mathbf{r} \leftarrow \mathbf{p}$
5: **while** $\mathbf{r} \geq \epsilon$ **do**
6: $\quad \mathbf{x}^{(1)} \leftarrow \mathbf{p}$
7: $\quad (y, y^{(1)}, \mathbf{g}, \mathbf{g}^{(1)}) = G^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$
8: $\quad \alpha \leftarrow \mathbf{r}^T \cdot \mathbf{r} \;/\; (\mathbf{p}^T \cdot \mathbf{g}^{(1)})$
9: $\quad d\mathbf{x} \leftarrow d\mathbf{x} + \alpha \cdot \mathbf{p}$
10: $\quad \mathbf{r}_{\text{prev}} \leftarrow \mathbf{r}$
11: $\quad \mathbf{r} \leftarrow \mathbf{r} - \alpha \cdot \mathbf{g}^{(1)}$
12: $\quad \beta \leftarrow \mathbf{r}^T \cdot \mathbf{r} \;/\; (\mathbf{r}_{\text{prev}}^T \cdot \mathbf{r}_{\text{prev}})$
13: $\quad \mathbf{p} \leftarrow \mathbf{r} + \beta \cdot \mathbf{p}$
14: **end while**

the constraints $\nabla c(\mathbf{x})$ needs to have full row rank; the Hessian

$$\nabla^2 \mathcal{L}(\mathbf{x}, \Lambda) \equiv \frac{\partial^2 \mathcal{L}}{\partial \mathbf{x}^2}(\mathbf{x}, \Lambda)$$

of the *Lagrangian* $\mathcal{L}(\mathbf{x}, \Lambda) = o(\mathbf{x}) - \Lambda^T \cdot c(\mathbf{x})$ with respect to $\mathbf{x}$ needs to be positive definite on the tangent space of the constraints, that is, $\mathbf{v}^T \cdot \nabla^2 \mathcal{L}(\mathbf{x}, \Lambda) \cdot \mathbf{v} > 0$ for all $\mathbf{v} \neq 0$ for which $\nabla c(\mathbf{x}) \cdot \mathbf{v} = 0$.

The iteration proceeds as

$$\begin{bmatrix} \mathbf{x}_{k+1} \\ \Lambda_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_k \\ \Lambda_k \end{bmatrix} + \begin{bmatrix} d\mathbf{x}_k \\ d\Lambda_k \end{bmatrix},$$

where the Newton step is computed as the solution of the linear system

$$\begin{bmatrix} \nabla^2 \mathcal{L}(\mathbf{x}_k, \Lambda_k) & -(\nabla c(\mathbf{x}_k))^T \\ \nabla c(\mathbf{x}_k) & 0 \end{bmatrix} \cdot \begin{bmatrix} d\mathbf{x}_k \\ d\Lambda_k \end{bmatrix} = \begin{bmatrix} (\nabla c(\mathbf{x}_k))^T \cdot \Lambda_k - \nabla o(\mathbf{x}_k) \\ -c(\mathbf{x}_k) \end{bmatrix}.$$

Note that

$$\nabla^2 \mathcal{L}(\mathbf{x}, \Lambda) = \nabla^2 o(\mathbf{x}) - \langle \Lambda, \nabla^2 c(\mathbf{x}) \rangle,$$

where the notation for the projection $\langle \Lambda, \nabla^2 c(\mathbf{x}) \rangle$ of the 3-tensor $\nabla^2 c(\mathbf{x}) \in \mathbb{R}^{m \times n \times n}$ in direction $\Lambda \in \mathbb{R}^m$ is formally introduced in Chapter 3.

Many modern algorithms for constrained nonlinear optimization are based on the solution of the KKT system. See, for example, [50] for a comprehensive survey. Our focus is on the efficient provision of the required derivatives.

If a direct linear solver is used, then the following derivatives need to be computed:

- $\nabla o(\mathbf{x})$ and $\nabla^2 o(\mathbf{x})$ at $O(n) \cdot Cost(o)$ using a second-order adjoint model of o;
- $\nabla c(\mathbf{x})$ and $\langle \Lambda, \nabla^2 c(\mathbf{x}) \rangle$ at $O(n) \cdot Cost(c)$ using a second-order adjoint model of c;
- $\langle \Lambda, \nabla c \rangle$ at $O(1) \cdot Cost(c)$ using the adjoint model of c.

A matrix-free implementation of a Newton–Krylov algorithm requires the following derivatives:

- $\nabla o(\mathbf{x})$ and $\langle \nabla^2 o(\mathbf{x}), v \rangle$, where $v \in \mathbb{R}^n$. Both can be computed at the cost of $O(1) \cdot Cost(o)$ using a second-order adjoint model of o;
- $\langle w, \nabla c(\mathbf{x}) \rangle$, $\langle \nabla c(\mathbf{x}), v \rangle$, and $\langle \Lambda, \nabla^2 c(\mathbf{x}), v \rangle$, where $v \in \mathbb{R}^n$ and $w \in \mathbb{R}^m$. The first- and second-order adjoint projections can be computed at the cost of $O(1) \cdot Cost(c)$ using a second-order adjoint model of c. A tangent-linear model of c permits the evaluation of Jacobian-vector products at the same relative computational cost.

Refer to Section 3.1 for formal definitions of the projection operator $\langle .,. \rangle$.

A detailed discussion of constrained nonlinear optimization is beyond the scope of this book. Various software packages have been developed to solve this type of problem. Some packages use AD techniques or can be coupled with code generated by AD. Examples include AMPL [26], IPOPT [59], and KNITRO [14]. Both IPOPT and KNITRO can be accessed via the Network-Enabled Optimization Server (NEOS[5]) maintained by Argonne National Laboratory's Mathematics and Computer Science Division. NEOS uses a variety of AD tools. Refer to the NEOS website for further information.

A case study for the use of AD in the context of constrained nonlinear programming is presented in [41]. Moreover, we discuss various combinatorial issues related to AD and to the use of sparse direct linear solvers. Our derivative code compiler `dcc` is combined with IPOPT [59] and PARDISO [55] to solve an inverse medium problem.

[5] `neos.mcs.anl.gov`

1.1.3 ... Numerical Libraries

The NAG C Library is a highly comprehensive collection of mathematical and statistical algorithms for computational scientists and engineers working with the programming languages C and C++. We use it as a case study for various—commercial as well as noncommercial—collections of derivative-based numerical algorithms. Their APIs (Application Programming Interfaces) are often very similar.

Systems of Nonlinear Equations

Function c05ubc of the NAG C Library computes a solution of a system of nonlinear equations $F(\mathbf{x}) = 0$, $F : \mathbb{R}^n \to \mathbb{R}^n$ by a modification of the Powell hybrid method [53]. It is based on the MINPACK routine HYBRJ1 [47]. The user must provide code for the accumulation of the Jacobian $\mathrm{A} \equiv \nabla F(\mathbf{x})$ as part of a function

```
void j_f(Integer n, const double x[],
                    double F[], double A[], ...);
```

The library provides a custom integer data type `Integer`. Algorithm 1.2 or—preferably—Algorithm 1.3 can be used to approximate the Jacobian or to accumulate its entries with machine accuracy at a computational cost of $O(n) \cdot Cost(F)$, respectively.

A similar approach can be taken for the integration of stiff systems of ordinary differential equations

$$\frac{\partial \mathbf{x}}{\partial t} = F(t, \mathbf{x})$$

using various NAG C Library routines. The API is similar to the above. The accumulation of the Jacobian of $F(t, \mathbf{x})$ with respect to $\mathbf{x}$ is analogous.

Unconstrained Nonlinear Optimization

The e04dgc section of the library deals with the minimization of an unconstrained nonlinear function $f : \mathbb{R}^n \to \mathbb{R}$, $y = f(\mathbf{x})$, where n is assumed to be very large. It uses a preconditioned, limited memory quasi-Newton CG method and is based upon algorithm PLMA as described in [33]. The user must provide code for the accumulation of the gradient $\mathrm{g} \equiv \nabla f(\mathbf{x})$ as part of a function

```
void g_f(Integer n, const double x[],
                    double *f, double g[], ...);
```

Both Algorithm 1.2 and Algorithm 1.3 can be used to approximate the gradient or to accumulate its entries with machine accuracy at a computational cost of $O(n) \cdot Cost(f)$, respectively. A better choice is Algorithm 1.8, which delivers the gradient with machine accuracy at a computational cost of $O(1) \cdot Cost(f)$.

Bound-Constrained Nonlinear Optimization

The e04lbc section of the library provides a modified Newton algorithm for finding unconstrained or bound-constrained minima of twice continuously differentiable nonlinear functions $f : \mathbb{R}^n \to \mathbb{R}$, $y = f(\mathbf{x})$ [32]. The user needs to provide code for the accumulation

of the gradient $g \equiv \nabla f(\mathbf{x})$ and for the computation of the function value $*y$ as part of a function

```
void g_f(Integer n, const double x[],
                    double *y, double g[], ...);
```

Moreover, code is required to accumulate the Hessian $H \equiv \nabla^2 f(\mathbf{x})$ inside of

```
void H_(Integer n, const double x[], double H[], ... );
```

The gradient should be computed with machine accuracy by Algorithm 1.8 at a computational cost of $O(1) \cdot Cost(f)$. For the Hessian, we can choose second-order finite differences, the second-order tangent-linear model, or the second-order adjoint model. While the first two run at a computational cost of $O(n^2) \cdot Cost(f)$, the second-order adjoint code delivers the Hessian with machine accuracy at a computational cost of only $O(n) \cdot Cost(f)$.

1.2 Manual Differentiation

Closed-form symbolic as well as algorithmic differentiation are based on two key ingredients: First, expressions for partial derivatives of the various arithmetic operations and intrinsic functions provided by programming languages are well known. Second, the chain rule of differential calculus holds.

Theorem 1.3 (Chain Rule of Differential Calculus). *Let*

$$\mathbf{y} = F(\mathbf{x}) = G_1(G_0(\mathbf{x}))$$

such that $G_0 : \mathbb{R}^n \to \mathbb{R}^k$, $\mathbf{z} = G_0(\mathbf{x})$ *is differentiable at* $\mathbf{x}$ *and* G_1; $\mathbb{R}^k \to \mathbb{R}^m$, $\mathbf{y} = G_1(\mathbf{z})$ *is differentiable at* $\mathbf{z}$. *Then* F *is differentiable at* $\mathbf{x}$ *and*

$$\frac{\partial F}{\partial \mathbf{x}} = \frac{\partial G_1}{\partial \mathbf{z}} \cdot \frac{\partial G_0}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}.$$

Proof. See, for example, [4]. □

Example 1.4 Let $\mathbf{y} = F(\mathbf{x}) = G_1(G_0(\mathbf{x}))$ such that $z = G_0(x_0, x_1) = x_0 \cdot x_1$ and

$$G_1(z) = \begin{pmatrix} \sin(z) \\ \cos(z) \end{pmatrix}.$$

Then,

$$\begin{aligned} \frac{\partial F}{\partial \mathbf{x}} &= \begin{pmatrix} \cos(z) \\ -\sin(z) \end{pmatrix} \cdot \begin{pmatrix} x_1 & x_0 \end{pmatrix} = \begin{pmatrix} \cos(z) \cdot x_1 & \cos(z) \cdot x_0 \\ -\sin(z) \cdot x_1 & -\sin(z) \cdot x_0 \end{pmatrix} \\ &= \begin{pmatrix} \cos(x_0 \cdot x_1) \cdot x_1 & \cos(x_0 \cdot x_1) \cdot x_0 \\ -\sin(x_0 \cdot x_1) \cdot x_1 & -\sin(x_0 \cdot x_1) \cdot x_0 \end{pmatrix}. \end{aligned}$$

■

The fundamental assumption we make is that at run time a computer program can be regarded as a sequence of assignments with arithmetic operations or intrinsic functions on their right-hand sides. The flow of control does not represent a serious problem as it is resolved uniquely for any given set of inputs.

Definition 1.5. *The given implementation of F as a* (numerical) program *is assumed to decompose into a* single assignment code *(SAC) at every point of interest as follows:*

$$\begin{aligned} &For\ j = n, \ldots, n+p+m-1, \\ &\quad v_j = \varphi_j(v_i)_{i \prec j}, \end{aligned} \tag{1.4}$$

where $i \prec j$ denotes a direct dependence of v_j on v_i. The transitive closure of this relation is denoted by $\prec^+$. The result of each elemental function φ_j is assigned to a unique auxiliary variable v_j. The n independent inputs *$x_i = v_i$, for $i = 0, \ldots, n-1$, are mapped onto m* dependent outputs *$y_j = v_{n+p+j}$, for $j = 0, \ldots, m-1$. The values of p* intermediate variables *v_k are computed for $k = n, \ldots, n+p-1$.*

Example 1.6 The SAC for the previous example becomes

$$\begin{aligned} &v_0 = x_0;\ v_1 = x_1 \\ &v_2 = v_0 \cdot v_1 \\ &v_3 = \sin(v_2) \\ &v_4 = \cos(v_2) \\ &y_0 = v_3;\ y_1 = v_4. \end{aligned}$$

∎

The SAC induces a directed acyclic graph (DAG) $G = (V, E)$ with integer vertices $V = \{0, \ldots, n+p+m-1\}$ and edges $E = \{(i, j) | i \prec j\}$. The vertices are sorted topologically with respect to variable dependence, that is, $\forall i, j \in V : (i, j) \in E \Rightarrow i < j$.

The elemental functions φ_j are assumed to possess jointly continuous partial derivatives with respect to their arguments. Association of the local partial derivatives with their corresponding edges in the DAG yields a *linearized DAG*.

Example 1.7 The linearized DAG for the function F in Example 1.4 is shown in Figure 1.2. ∎

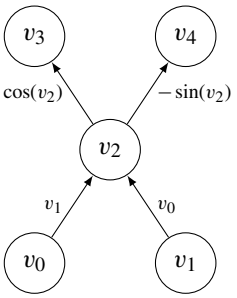


Figure 1.2. *Linearized DAG of F in Example* 1.4.

Let $A = (a_{i,j}) \equiv \nabla F(\mathbf{x})$. As an immediate consequence of the chain rule, the individual entries of the Jacobian can be computed as

$$a_{i,j} = \sum_{\pi \in [i \to n+p+j]} \prod_{(k,l) \in \pi} c_{l,k} \tag{1.5}$$

where

$$c_{l,k} \equiv \frac{\partial \varphi_l}{\partial v_k}(v_q)_{q \prec l}$$

and $[i \to n+p+j]$ denotes the set of all paths that connect the independent vertex i with the dependent vertex $n+p+j$ [6].

Example 1.8 From Figure 1.2 we get immediately

$$\nabla F = \begin{pmatrix} \cos(v_2) \cdot v_1 & \cos(v_2) \cdot v_0 \\ -\sin(v_2) \cdot v_1 & -\sin(v_2) \cdot v_0 \end{pmatrix} = \begin{pmatrix} \cos(x_0 \cdot x_1) \cdot x_1 & \cos(x_0 \cdot x_1) \cdot x_0 \\ -\sin(x_0 \cdot x_1) \cdot x_1 & -\sin(x_0 \cdot x_1) \cdot x_0 \end{pmatrix}. \qquad \blacksquare$$

Linearized DAGs and the chain rule (for example, formulated as in (1.5)) can be useful tools for the manual differentiation of numerical simulation programs. Nonetheless, this process can be extremely tedious and highly error-prone.

When differentiating computer programs, one aims for a derivative code that covers the entire domain of the original code. Correct derivatives should be computed for any set of inputs for which the underlying function is defined. Manual differentiation is reasonably straightforward for straight-line code, that is, for sequences of assignments that are not interrupted by control flow statements or subprogram calls. In this case, the DAG is static, meaning that its structure remains unchanged for varying values of the inputs. Manual differentiation of computer programs becomes much more challenging under the presence of control flow.

Consider the given implementation of (1.2):

```
void f(int n, double *x, double &y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}
```

The structure of the DAG varies with the value of n indicated by the variable index i in Figure 1.3. A handwritten gradient code might look as follows:

```
1 void g_f(int n, double *x, double &y, double *g) {
2   y=0;
3   for (int i=0;i<n;i++) {
4     g[i]=2*x[i];
5     y=y+x[i]*x[i];
6   }
7   for (int i=0;i<n;i++) g[i]=g[i]*2*y;
8   y=y*y;
9 }
```

Local gradients of the sums in line 5 are built in line 4. Each of them needs to be multiplied in line 7 with the local partial derivative of the square operation in line 8 to obtain the

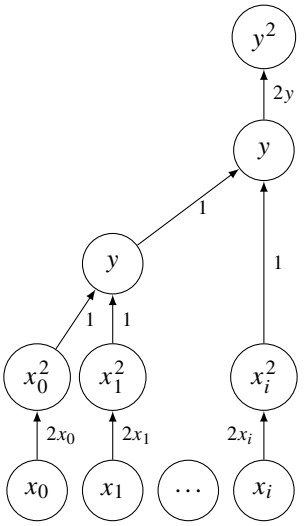


Figure 1.3. *Linearized DAG of the given implementation of* $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$.

final gradient. While this simple example is certainly manageable, it still demonstrates how painful this procedure is likely to become for complex code involving nontrivial control flow and many subprograms.

Repeating the above process for second derivatives computed by the differentiated gradient code yields the following handwritten Hessian code:

```
1 void h_g_f(int n, double *x, double &y, double *g, double **H){
2   y=0;
3   for (int i=0;i<n;i++) {
4     g[i]=2*x[i];
5     y=y+x[i]*x[i];
6   }
7   for (int i=0;i<n;i++)
8     for (int j=0;j<=i;j++) {
9       H[i][j]=4*x[j]*g[i];
10      if (i==j)
11        H[i][j]=H[i][j]+4*y;
12      else
13        H[j][i]=H[i][j];
14    }
15  for (int i=0;i<n;i++) g[i]=g[i]*2*y;
16  y=y*y;
17 }
```

This code is based on the linearized DAG of the gradient code shown in Figure 1.4. Again, the structure of this DAG depends on the value of n. The entries of the Hessian are assembled in lines 9, 11, and 13.

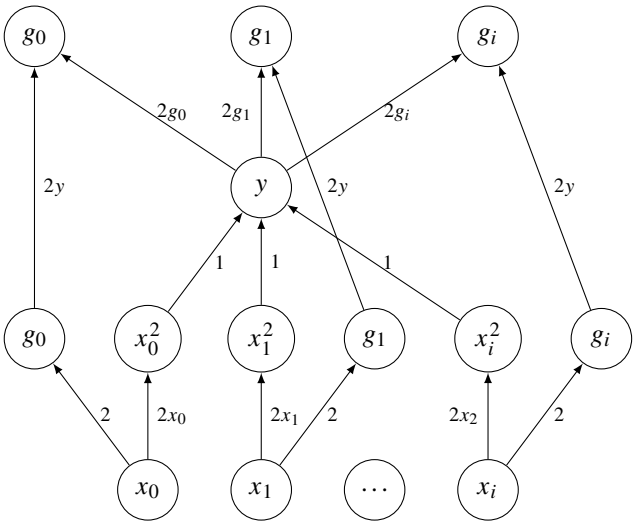


Figure 1.4. *Linearized DAG of the given implementation of the gradient of* $y = \left(\sum_{i=0}^{n-1} x_i^2\right)^2$.

Unfortunately, the symbolic differentiation capabilities of your favorite computer algebra system are unlikely to be of much help. They are not needed for the very simple local partial derivatives and are unable to cope with the control flow. Note that some computer algebra systems such as Maple [29] and Mathematica [61] have recently added AD capabilities to their list of functionalities.

1.3 Approximation of Derivatives

Despite its obvious drawbacks, finite differences can be a useful tool for debugging and potentially verifying derivative code. If the computed values and their approximation match, then the tested derivative code is probably correct for the given set of inputs. If they do not match, then this may be an indication of an error in the derivative code. However, it may as well be the finite difference approximation that turns out to be wrong. There is no easy "bullet-proof" check for correctness of derivatives. In the extreme case, you may just have to debug your derivative code line by line. Finite differences applied to carefully selected parts of the original code may provide good support.

Definition 1.9. *Let* $D \subseteq \mathbb{R}^n$ *be an open domain and let* $F : D \to \mathbb{R}^m$,

$$F = \begin{pmatrix} F_0 \\ \vdots \\ F_{m-1} \end{pmatrix},$$

be continuously differentiable on D. A forward finite difference *approximation of the ith column of the Jacobian*

$$\nabla F(\mathbf{x}) = \left(\frac{\partial F_j}{\partial x_i}(\mathbf{x})\right)_{i=0,\ldots,n-1}^{j=0,\ldots,m-1}$$

at point $\mathbf{x}$ *is computed as*

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) \equiv \begin{pmatrix} \frac{\partial F_0}{\partial x_i}(\mathbf{x}) \\ \vdots \\ \frac{\partial F_{m-1}}{\partial x_i}(\mathbf{x}) \end{pmatrix} \approx_1 \frac{F(\mathbf{x}+\mathbf{e}^i \cdot \delta) - F(\mathbf{x})}{\delta}, \tag{1.6}$$

*for $i = 0,\ldots,n-1$ and where the i*th Cartesian basis vector in $\mathbb{R}^n$ *is denoted by* $\mathbf{e}^i$.

A backward finite difference *approximation of the ith column of the same Jacobian is computed as*

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) \approx_1 \frac{F(\mathbf{x}) - F(\mathbf{x}-\mathbf{e}^i \cdot \delta)}{\delta}. \tag{1.7}$$

The first-order accuracy (denoted by $\approx_1$) of unidirectional finite differences follows immediately from the Taylor expansion of F at $\mathbf{x}$. For the sake of simplicity in the notation, we consider the univariate scalar case where $f : \mathbb{R} \to \mathbb{R}$. Without loss of generality, let f be infinitely often continuously differentiable at $x \in \mathbb{R}$. Then, the Taylor expansion of f at x is given by

$$\begin{aligned} f(x') = f(x) &+ \frac{\partial f}{\partial x}(x) \cdot (x'-x) \\ &+ \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x) \cdot (x'-x)^2 + \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x) \cdot (x'-x)^3 + \cdots . \end{aligned} \tag{1.8}$$

For $x' = x + \delta$ we get

$$f(x+\delta) = f(x) + \frac{\partial f}{\partial x}(x) \cdot \delta + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x) \cdot \delta^2 + \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x) \cdot \delta^3 + \cdots , \tag{1.9}$$

and similarly for $x' = x - \delta$

$$f(x-\delta) = f(x) - \frac{\partial f}{\partial x}(x) \cdot \delta + \frac{1}{2!} \cdot \frac{\partial^2 f}{\partial x^2}(x) \cdot \delta^2 - \frac{1}{3!} \cdot \frac{\partial^3 f}{\partial x^3}(x) \cdot \delta^3 + \cdots . \tag{1.10}$$

Truncation after the first derivative terms of (1.9) and (1.10) yields scalar univariate versions of (1.6) and (1.7), respectively. For $0 \ll \delta < 1$, the truncation error is dominated by the value of the δ^2 term, which implies that only accuracy up to the order of δ (equaling δ^1 and hence first-order accuracy) can be expected.

Postponing the truncation of the Taylor series will increase the order of accuracy. This fact is exploited by central finite differences.

Definition 1.10. *Let $F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m$ be given as in Definition* 1.9. *A* central finite difference *approximation of the ith column of the Jacobian ∇F at point* $\mathbf{x}$ *is computed as*

$$\frac{\partial F}{\partial x_i}(\mathbf{x}) \approx_2 \frac{F(\mathbf{x}+\mathbf{e}^i \cdot \delta) - F(\mathbf{x}-\mathbf{e}^i \cdot \delta)}{2 \cdot \delta}. \tag{1.11}$$

Second-order accuracy ($\approx_2$) follows immediately from (1.8). Subtraction of (1.10) from (1.9) yields

$$\begin{aligned} & f(x+\delta) - f(x-\delta) \\ & \quad = f(x) + \frac{\partial f}{\partial x}(x)\cdot\delta + \frac{1}{2!}\cdot\frac{\partial^2 f}{\partial x^2}(x)\cdot\delta^2 + \frac{1}{3!}\cdot\frac{\partial^3 f}{\partial x^3}(x)\cdot\delta^3 + \cdots \\ & \qquad - \left(f(x) - \frac{\partial f}{\partial x}(x)\cdot\delta + \frac{1}{2!}\cdot\frac{\partial^2 f}{\partial x^2}(x)\cdot\delta^2 - \frac{1}{3!}\cdot\frac{\partial^3 f}{\partial x^3}(x)\cdot\delta^3 + \cdots \right) \\ & \quad = 2\cdot\frac{\partial f}{\partial x}(x)\cdot\delta + \frac{2}{3!}\cdot\frac{\partial^3 f}{\partial x^3}(x)\cdot\delta^3 + \cdots . \end{aligned}$$

Truncation after the first derivative term yields the scalar univariate version of (1.11). For small values of δ, the truncation error is dominated by the value of the δ^3 term, which implies that only accuracy up to the order of δ^2 (second-order accuracy) can be expected.

Example 1.11 The gradient of the given implementation of (1.2) is accumulated by forward finite differences with perturbation $\delta \equiv \mathrm{h} = 10^{-9}$ as follows:

```
1  void g_f_ffd(int n, double* x, double& y, double *g) {
2    const double h=1e-9;
3    double y_ph;
4    double *x_ph=new double[n];;
5    for (int i=0;i<n;i++) x_ph[i]=x[i];
6    f(n,x,y);
7    for (int i=0;i<n;i++) {
8      x_ph[i]+=h;
9      f(n,x_ph,y_ph);
10     g[i]=(y_ph-y)/h;
11     x_ph[i]=x[i];
12   }
13   delete [] x_ph;
14 }
```

The driver routine for backward finite differences is obtained by subtracting (instead of adding) h in line 8 and by switching the operands in the subtraction in line 10.

Extension of the driver to central finite differences is straightforward:

```
1  void g_f_cfd(int n, double* x, double& y, double *g) {
2    const double h=5e-10;
3    double y_mh,y_ph;
4    double *x_mh=new double[n];;
5    double *x_ph=new double[n];;
6    for (int i=0;i<n;i++) x_ph[i]=x_mh[i]=x[i];
7    for (int i=0;i<n;i++) {
8      x_mh[i]-=h;
9      f(n,x_mh,y_mh);
10     x_ph[i]+=h;
11     f(n,x_ph,y_ph);
12     g[i]=(y_ph-y_mh)/(2*h);
13     x_ph[i]=x_mh[i]=x[i];
```

```
14    }
15    f(n,x,y);
16    delete [] x_ph;
17    delete [] x_mh;
18  }
```

The call of f at the original point x in line 15 ensures the return of the correct function value y. ∎

Directional derivatives can be approximated by forward finite differences as

$$\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \approx_1 \frac{F(\mathbf{x} + \delta \cdot \mathbf{x}^{(1)}) - F(\mathbf{x})}{\delta}.$$

The correctness of this statement follows immediately from the forward finite difference approximation of the Jacobian of $F(\mathbf{x} + s \cdot \mathbf{x}^{(1)})$ at point $s = 0$ as

$$\begin{aligned}\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} &\approx_1 \left.\frac{F(\mathbf{x} + (s+\delta) \cdot \mathbf{x}^{(1)}) - F(\mathbf{x} + s \cdot \mathbf{x}^{(1)})}{\delta}\right|_{s=0} \\ &= \frac{F(\mathbf{x} + \delta \cdot \mathbf{x}^{(1)}) - F(\mathbf{x})}{\delta}.\end{aligned}$$

Similarly, $\mathbf{y}^{(1)}$ can be approximated by backward finite differences

$$\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \approx_1 \frac{F(\mathbf{x}) - F(\mathbf{x} - \delta \cdot \mathbf{x}^{(1)})}{\delta}$$

or by central finite differences

$$\mathbf{y}^{(1)} \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \approx_2 \frac{F(\mathbf{x} + \delta \cdot \mathbf{x}^{(1)}) - F(\mathbf{x} - \delta \cdot \mathbf{x}^{(1)})}{2 \cdot \delta}.$$

The classical definition of derivatives as limits of finite difference quotients suggests that the quality of an approximation is improved by making the perturbation δ smaller. Unfortunately, this assumption is not valid in finite precision arithmetic such as implemented by today's computers.

The way in which numbers are represented on a computer is defined by the IEEE 754 standard [1]. Real numbers $x \in \mathbb{R}$ are represented with *base* β, *precision* t, and *exponent range* $[L, U]$ as

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{t-1}}{\beta^{t-1}} \right) \cdot \beta^e,$$

where $0 \le d_i \le \beta - 1$ for $i = 0, \ldots, t-1$, and $L \le e \le U$. The string of base-β digits $m = d_0 d_1 \ldots d_{t-1}$ is called the *mantissa* and e is called the *exponent.* A floating-point system is *normalized* if $d_0 \neq 0$ unless $x = 0$, that is, $1 \le m < \beta$. We assume $\beta = 2$, as this is case for almost any existing computer.

Example 1.12 Let $\beta = 2$, $t = 3$, and $[L,U] = [-1,1]$. The corresponding normalized floating-point number system contains the following 25 elements:

$$
\begin{aligned}
&0\\
&\pm 1.00_2 * 2^{-1} = \pm 0.5_{10}, \quad \pm 1.01_2 * 2^{-1} = \pm 0.625_{10}\\
&\pm 1.10_2 * 2^{-1} = \pm 0.75_{10}, \quad \pm 1.11_2 * 2^{-1} = \pm 0.875_{10}\\
&\pm 1.00_2 * 2^{0} = \pm 1_{10}, \quad \pm 1.01_2 * 2^{0} = \pm 1.25_{10}\\
&\pm 1.10_2 * 2^{0} = \pm 1.5_{10}, \quad \pm 1.11_2 * 2^{0} = \pm 1.75_{10}\\
&\pm 1.00_2 * 2^{1} = \pm 2_{10}, \quad \pm 1.01_2 * 2^{1} = \pm 2.5_{10}\\
&\pm 1.10_2 * 2^{1} = \pm 3_{10}, \quad \pm 1.11_2 * 2^{1} = \pm 3.5_{10}. \qquad \blacksquare
\end{aligned}
$$

The IEEE *single precision* floating-point number data type **float** uses 32 bits: 23 bits for its mantissa, 8 bits for the exponent, and one sign bit. In decimal representation, we get 6 significant digits with minimal and maximal absolute values of 1.17549e-38 and 3.40282e+38, respectively. The stored exponent is *biased* by adding $2^7 - 1 = 127$ to its actual signed value.

The *double precision* floating-point number data type **double** uses 64 bits: 52 bits for its mantissa, 11 bits for the exponent, and one sign bit. In decimal representation we get 15 significant digits with minimal and maximal absolute values of 2.22507e-308 and 1.79769e+308, respectively. The stored biased exponent is obtained by adding $2^{10} - 1 = 1023$ to its actual signed value. Higher precision floating-point types are defined accordingly.

If $x \in \mathbb{R}$ is not exactly representable in the given floating-point number system, then it must be approximated by a nearby floating-point number. This process is known as *rounding*. The default algorithm for rounding in binary floating-point arithmetic is *rounding to nearest*, where x is represented by the nearest floating-point number. Ties are resolved by choosing the floating-point number whose last stored digit is even, i.e., equal to zero in binary floating-point arithmetic.

Example 1.13 In the previously discussed ($\beta = 2, t = 3, [L,U] = [-1,1]$) floating-point number system, the decimal value 1.126 is represented as 1.25 when rounding to nearest. Tie breaking toward the trailing zero in the mantissa yields 1 for 1.125. $\blacksquare$

Let x and y be two floating-point numbers that agree in all but the last few digits. If we compute $z = x - y$, then z may only have a few digits of accuracy due to *cancellation*. Subsequent use of z in a calculation may impact the accuracy of the result negatively. Finite difference approximation of derivatives is a prime example for potentially catastrophic loss in accuracy due to cancellation and rounding.

Example 1.14 Consider the approximation of the first derivative of $y = f(x) = x$ in single precision IEEE floating-point arithmetic at $x = 10^6$ by the forward finite difference quotient

$$\nabla f(x) \approx \frac{f(x+\delta) - f(x)}{\delta}$$

with $\delta \equiv \mathrm{h} = 0.1$. Obviously, $\nabla f(x) = 1$ independent of x. Still, the code

```
...
float x=1e6, h=1e-1;
cout << "(f(x+h)-f(x))/h=" << (x+h-x)/h << endl;
...
```

returns 1.25. A bit-level look at this computation yields

$$\begin{aligned} f(x) = x &= 0\ 10010010\ 11101000010010000000000 \\ \delta &= 0\ 01111011\ 10011001100110011001101 \\ f(x+\delta) = x+\delta &= 0\ 10010010\ 11101000010010000000010 \end{aligned}$$

and hence

$$\begin{aligned} f(x+\delta) - f(x) &= 0\ 01111100\ 00000000000000000000000 \\ (f(x+\delta) - f(x))/\delta &= 0\ 01111111\ 01000000000000000000000. \end{aligned}$$

Thus $\delta = 0.1$ is rounded to its nearest representable neighbor

$$\begin{aligned} \delta \approx (1 &+ 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} \\ &+ 2^{-21} + 2^{-23}) \cdot 2^{(2^0+2^1+2^3+2^4+2^5+2^6-127)}. \end{aligned}$$

Moreover, we observe that

$$\begin{aligned} f(x) &= (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-13}) \cdot 2^{(2^1+2^4+2^7-127)} \\ &= 1.9073486328125 \cdot 2^{19} \approx 1{,}000{,}000 \end{aligned}$$

and

$$\begin{aligned} f(x+\delta) &= (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-13} + 2^{-22}) \cdot 2^{(2^1+2^4+2^7-127)} \\ &= 1.907348871231079101562 5 \cdot 2^{19} \approx 1{,}000{,}000.125. \end{aligned}$$

Even if the subtraction of $f(x)$ from $f(x+\delta)$ is performed internally with higher precision, the subsequent division by δ followed by rounding to single precision yields a significant loss in accuracy, as we have $(f(x+\delta) - f(x))/\delta = 1.25$ as opposed to the correct result 1. ∎

The impact of cancellation and rounding becomes even more dramatic if second derivatives are approximated using second-order finite differences.

Theorem 1.15. *Let $F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m$ be given as in Definition* 1.9. Central finite difference *approximations of the ith columns of the Hessians*

$$\nabla^2 F_k(\mathbf{x}) \equiv \left(\frac{F_k}{\partial x_i \partial x_j}(\mathbf{x}) \right)$$

at point $\mathbf{x}$ *are computed for $k = 0, \ldots, m-1$ and $i, j = 0, \ldots, n-1$ as*

$$\begin{aligned} \frac{\partial^2 F_k}{\partial x_i \partial x_j}(\mathbf{x}) \approx \Big[& F(\mathbf{x} + (\mathbf{e}^j + \mathbf{e}^i) \cdot \delta) - F(\mathbf{x} + (\mathbf{e}^j - \mathbf{e}^i) \cdot \delta) \\ & - F(\mathbf{x} + (\mathbf{e}^i - \mathbf{e}^j) \cdot \delta) + F(\mathbf{x} - (\mathbf{e}^j + \mathbf{e}^i) \cdot \delta) \Big] / (4 \cdot \delta^2). \end{aligned} \tag{1.12}$$

Proof. (1.12) follows immediately from

$$\begin{aligned}\frac{\partial^2 F_k}{\partial x_i \partial x_j}(\mathbf{x}) &\approx \frac{\frac{\partial F_k}{\partial x_i}(\mathbf{x}+\mathbf{e}^j\cdot\delta)-\frac{\partial F_k}{\partial x_i}(\mathbf{x}-\mathbf{e}^j\cdot\delta)}{2\cdot\delta}\\ &= \left[\frac{F(\mathbf{x}+\mathbf{e}^j\cdot\delta+\mathbf{e}^i\cdot\delta)-F(\mathbf{x}+\mathbf{e}^j\cdot\delta-\mathbf{e}^i\cdot\delta)}{2\cdot\delta}\right.\\ &\quad \left.-\frac{F(\mathbf{x}-\mathbf{e}^j\cdot\delta+\mathbf{e}^i\cdot\delta)-F(\mathbf{x}-\mathbf{e}^j\cdot\delta-\mathbf{e}^i\cdot\delta)}{2\cdot\delta}\right]/(2\cdot\delta) \qquad \square\end{aligned}$$

For $f:\mathbb{R}\to\mathbb{R}$, we get the well-known formula

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+\delta)-2\cdot f(x)+f(x-\delta)}{\delta^2} \quad .$$

Example 1.16 The Hessian of the given implementation f of Equation (1.2) can be accumulated by second-order central finite differences with $\delta \equiv \mathrm{h} = 10^{-6}$ as follows:

```
1  void h_cfd(int n, double* x, double** H) {
2    const double h=1e-6;
3    double yp1, yp2;
4    double* xp=new double[n];
5    for (int i=0;i<n;i++) xp[i]=x[i];
6    for (int i=0;i<n;i++) {
7      for (int j=0;j<=i;j++) {
8        xp[i]=x[i]; xp[j]=x[j];
9        xp[i]+=+h; xp[j]+=h;
10       f(n, xp, yp2);
11       xp[i]=x[i]; xp[j]=x[j];
12       xp[i]-=h; xp[j]+=h;
13       f(n, xp, yp1); yp2-=yp1;
14       xp[i]=x[i]; xp[j]=x[j];
15       xp[i]+=h; xp[j]-=h;
16       f(n, xp, yp1); yp2-=yp1;
17       xp[i]=x[i]; xp[j]=x[j];
18       xp[i]-=h; xp[j]-=h;
19       f(n, xp, yp1); yp2+=yp1;
20       H[i][j]=H[j][i]=yp2/(4*h*h);
21     }
22   }
23   delete [] xp;
24 }
```

Symmetry is exploited through restriction of the computation to the lower—resp., upper—triangular $n \times n$ submatrix of the Hessian in lines 7 and 20.

Alternatively, central finite differences can be applied to the given finite difference approximation g_f_cfd of the gradient.

```
1  void h_g_f_cfd(int n, double* x, double& y,
2                 double *g, double** H) {
3    const double h=1e-6;
4    double* x_mh=new double[n];
5    double* x_ph=new double[n];
6    double* g_mh=new double[n];
7    double* g_ph=new double[n];
8    for (int i=0;i<n;i++) x_mh[i]=x_ph[i]=x[i];
9    for (int i=0;i<n;i++) {
10     x_mh[i]-=h;
11     g_f_cfd(n,x_mh,y,g_mh);
12     x_ph[i]+=h;
13     g_f_cfd(n,x_ph,y,g_ph);
14     for (int j=i;j<n;j++)
15       H[i][j]=H[j][i]=(g_ph[j]-g_mh[j])/(2*h);
16     x_ph[i]=x_mh[i]=x[i];
17   }
18   g_f_cfd(n,x,y,g);
19   delete [] g_ph, g_mh, x_ph, x_mh;
20 }
```

Symmetry is exploited by restricting the evaluation of the difference quotient in line 15 to the lower right $(n-i)\times(n-i)$ submatrix of the Hessian. The savings are more substantial if second-order finite differences are applied directly to f. ∎

As our goal is certainly not the promotion of finite differences; we quickly move on to AD as the main subject of this book in the following chapters.

1.4 Exercises

1.4.1 Finite Differences and Floating-Point Arithmetic

Write a C++ program that converts single precision floating-point variables into their bit representation (see Section 1.3). Investigate the effects of cancellation and rounding on the finite difference approximation of first and second derivatives of a set of functions of your choice.

1.4.2 Derivatives for Systems of Nonlinear Equations

Apply Algorithm 1.1 to approximate a solution $\mathbf{y} = \mathbf{y}(\mathbf{x}_0, \mathbf{x}_1)$ of the discrete SFI problem introduced in Example 1.2.

1. Approximate the Jacobian of the residual $\mathbf{r} = F(\mathbf{y})$ by finite differences. Write exact derivative code based on (1.5) for comparison.

2. Use finite differences to approximate the product of the Jacobian with a vector within a matrix-free implementation of the Newton algorithm based on Algorithm 1.4.

3. Repeat the above for further problems from the MINPACK-2 test problem collection [5], for example, for the *Flow in a Channel* and *Elastic Plastic Torsion* problems.

1.4.3 Derivatives for Nonlinear Programming

Apply the steepest descent and Newton algorithms to an extended version of the Rosenbrock function [54], which is defined as

$$y = f(\mathbf{x}) \equiv \sum_{i=0}^{n-2} (1 - x_i)^2 + 10 \cdot (x_{i+1} - x_i^2)^2$$

for $n = 10, 100, 1000$ and for varying starting values of your choice. The function has a global minimum at $x_i = 1$, $i = 0, \ldots, n-1$, where $f(\mathbf{x}) = 0$. Approximate the required derivatives by finite differences. Observe the behavior (development of function values and L_2-norm of gradient; run time) of the algorithms for varying values of the perturbation size. Use (1.5) to derive (handwritten) exact derivatives for comparison.

Repeat the above for

$$y = f(\mathbf{x}) \equiv \sum_{i=0}^{n-4} (x_i + 10x_{i+1})^2 + 5(x_{i+2} - x_{i+3})^2 + (x_{i+1} - 2x_{i+2})^4 + 10(x_i - x_{i+3})^4$$

with minimum $f(\mathbf{x}) = 0$ at $x_i = 0$, $i = 0, \ldots, n-1$, and for

$$y = f(\mathbf{x}) \equiv \sum_{i=0}^{n-2} (x_i^2)^{x_{i+1}^2 + 1} + (x_{i+1}^2)^{x_i^2 + 1}$$

with minimum $f(\mathbf{x}) = 0$ at $x_i = 1$, $i = 0, \ldots, n-1$.

1.4.4 Derivatives for Numerical Libraries

Use manual differentiation and finite differences with your favorite solver for

1. Systems of nonlinear equations to find a numerical solution of the SFI problem introduced in Section 1.4.2; repeat for further MINPACK-2 test problems.

2. Nonlinear programming to minimize the Rosenbrock function; repeat for the other two test problems from Section 1.4.3.

Chapter 2

First Derivative Code

Chapter 2 aims to equip the reader with the fundamentals of first derivative code generated by AD. Two methods for implementing tangent-linear and adjoint models are considered: source transformation and operator overloading. The former is introduced as a technique for rewriting numerical simulation programs manually. With current AD technology still being far from "plug-and-play," we feel that users of AD tools should theoretically be capable to perform the underlying semantic transformations by hand. Otherwise it is unlikely that they will unleash the full power of these tools.

The automated generation of first derivative code is based on the knowledge about partial derivatives of the intrinsic functions and arithmetic operators offered by programming languages and on the chain rule of differential calculus—its associativity in particular. We focus on aspects of AD with immediate relevance to the derivative code compiler `dcc` that is presented in Chapter 5. For a comprehensive discussion of advanced issues in AD we refer the reader to [36]. A combination of both texts will be a very good starting point for anyone interested in AD and derivative code compiler technology.

Some notation is needed for the upcoming material.

Definition 2.1. *Let $D \subseteq \mathbb{R}^n$ be an open domain and let $f : D \to \mathbb{R}$ be continuously differentiable on D. The* partial derivative *of $y = f(\mathbf{x})$, $\mathbf{x} = (x_i)_{i=0,\ldots,n-1}$, at point $\mathbf{x}_0$ with respect to x_j is denoted as*

$$f_{x_j}(\mathbf{x}_0) \equiv \frac{\partial f}{\partial x_j}(\mathbf{x}_0).$$

The vector

$$\nabla f(\mathbf{x}_0) \equiv \begin{pmatrix} f_{x_0}(\mathbf{x}_0) \\ \vdots \\ f_{x_{n-1}}(\mathbf{x}_0) \end{pmatrix} \in \mathbb{R}^n$$

is called the gradient of f at point $\mathbf{x}_0$.

Example 2.2 The gradient of the Rosenbrock function

$$y = f(\mathbf{x}) = (1 - x_0)^2 + 100 \cdot (x_1 - x_0^2)^2$$

(see also Section 1.4.3) is a vector $\nabla f \in \mathbb{R}^2$ defined as

$$\nabla f = \nabla f(\mathbf{x}) \equiv \begin{pmatrix} 400 \cdot x_0^3 + 2 \cdot x_0 - 400 \cdot x_1 \cdot x_0 - 2 \\ 200 \cdot x_1 - 200 x_0^2 \end{pmatrix}.$$

It vanishes identically at $x = (1, 1)$ due to a local extremum. ∎

Definition 2.3. *Let $D \subseteq \mathbb{R}^n$ be an open domain and let*

$$F \equiv (F_i)_{i=0,\ldots,m-1} : D \to \mathbb{R}^m$$

be continuously differentiable on D. The matrix

$$\nabla F(\mathbf{x}_0) \equiv \begin{pmatrix} (\nabla F_0(\mathbf{x}_0))^T \\ \vdots \\ (\nabla F_{m-1}(\mathbf{x}_0))^T \end{pmatrix} \in \mathbb{R}^{m \times n}$$

containing the gradients of each of the m components F_i of F as rows is called the Jacobian of F at point $\mathbf{x}_0$.

Example 2.4 Consider the Jacobian $\nabla \mathbf{r} = \nabla \mathbf{r}(\mathbf{y}, \lambda) \in \mathbb{R}^{4 \times 4}$ of the discrete residual of the SFI problem introduced in Example 1.2. For $s = 3$ and $h = 1/s$ the Jacobian of

$$\begin{aligned} r_0 &= -4 \cdot y_{1,1} + y_{2,1} + y_{1,2} - h^2 \cdot \lambda \cdot e^{y_{1,1}} \\ r_1 &= -4 \cdot y_{1,2} + y_{2,2} + y_{1,1} - h^2 \cdot \lambda \cdot e^{y_{1,2}} \\ r_2 &= -4 \cdot y_{2,1} + y_{1,1} + y_{2,2} - h^2 \cdot \lambda \cdot e^{y_{2,1}} - 1 \\ r_3 &= -4 \cdot y_{2,2} + y_{1,2} + y_{2,1} - h^2 \cdot \lambda \cdot e^{y_{2,2}} - 1 \end{aligned}$$

with respect to $\mathbf{y}$ becomes

$$\nabla \mathbf{r} \equiv \begin{pmatrix} -h^2 \cdot \lambda \cdot e^{y_{1,1}} - 4 & 1 & 1 & 0 \\ 1 & -h^2 \cdot \lambda \cdot e^{y_{1,2}} - 4 & 0 & 1 \\ 1 & 0 & -h^2 \cdot \lambda \cdot e^{y_{2,1}} - 4 & 1 \\ 0 & 1 & 1 & -h^2 \cdot \lambda \cdot e^{y_{2,2}} - 4 \end{pmatrix}$$

yielding

$$\nabla \mathbf{r} = \begin{pmatrix} -4.15 & 1 & 1 & 0 \\ 1 & -4.15 & 0 & 1 \\ 1 & 0 & -4.15 & 1 \\ 0 & 1 & 1 & -4.15 \end{pmatrix}$$

for $y_{1,1} = y_{1,2} = y_{2,1} = y_{2,2} = 1$ and $\lambda = 0.5$. ∎

As before, we consider multivariate vector functions $F : \mathbb{R}^n \to \mathbb{R}^m$ mapping a vector of independent inputs $\mathbf{x} \in \mathbb{R}^n$ onto a vector of dependent outputs $\mathbf{y} \in \mathbb{R}^m$ as $\mathbf{y} = F(\mathbf{x})$. The function F is assumed to be continuously differentiable in a neighborhood of all arguments at which it is assumed to be evaluated. Hence, its Jacobian matrix $\nabla F \equiv \nabla F(\mathbf{x}) \in \mathbb{R}^{m \times n}$

exists at all of these points, and it contains the numerical values of the corresponding partial derivatives of the components of $\mathbf{y}$ with respect to the components of $\mathbf{x}$:

$$\nabla F(\mathbf{x}) = \left(\frac{\partial y_j}{\partial x_i}\right)_{i=0,\ldots,n-1}^{j=0,\ldots,m-1}.$$

This chapter forms the basis for the remaining material discussed in Chapters 3, 4, and 5. Tangent-linear and adjoint models of numerical simulation programs and their generation using forward and reverse mode AD are discussed in Sections 2.1 and 2.2, respectively. We focus on the (manual) implementation of tangent-linear and adjoint code and its semiautomatic generation by means of overloading of arithmetic operators and intrinsic functions. Compiler-based source transformation techniques are considered in Chapter 4. The bottleneck of adjoint code is the often excessive memory requirement which is proportional to the number of statements executed by the original code. Its reduction through trade-offs between storage and recomputation in the context of checkpointing schemes is the subject of Section 2.3.

2.1 Tangent-Linear Model

The purely mathematical formulation of AD is essentially straightforward. Most problems arise when implementing AD on a computer. The fact that one and the same program variable (memory location) can hold the values of several mathematical variables yields complications, particularly for adjoint code. Tangent-linear models and their implementations turn out to be considerably simpler. Unless stated otherwise, we assume that distinct variables $\mathbf{x}$ and $\mathbf{y}$ are stored in distinct memory locations.

Definition 2.5. *The Jacobian $\nabla F = \nabla F(\mathbf{x})$ induces a linear mapping $\nabla F : \mathbb{R}^n \to \mathbb{R}^m$ defined by*

$$\mathbf{x}^{(1)} \mapsto \nabla F \cdot \mathbf{x}^{(1)}.$$

The function $F^{(1)} : \mathbb{R}^{2\cdot n} \to \mathbb{R}^m$, defined as

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}, \tag{2.1}$$

is referred to as the tangent-linear model *of F.*

The directional derivative $\mathbf{y}^{(1)}$ can be regarded as the partial derivative of $\mathbf{y}$ with respect to an auxiliary scalar variable s, where

$$\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial s}.$$

By the chain rule, we get

$$\mathbf{y}^{(1)} \equiv \frac{\partial \mathbf{y}}{\partial s} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial s} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}.$$

Derivative code compilers such as `dcc` (see Chapter 5) transform a given implementation

```
void f(int n, int m, double* x, double* y)
```

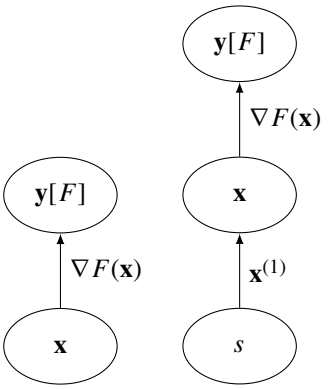


Figure 2.1. *High-level linearized DAG of* $\mathbf{y} = F(\mathbf{x})$ *(a) and its tangent-linear extension (b).*

of the function $\mathbf{y} = F(\mathbf{x})$ into tangent-linear code $(\mathbf{y}^{(1)}, \mathbf{y}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$ for computing both the function value and the directional derivative:

$$\begin{aligned} \mathbf{y}^{(1)} &= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}, \\ \mathbf{y} &= F(\mathbf{x}). \end{aligned}$$

The signature of the resulting tangent-linear subroutine is the following:

```
void t1_f(int n, int m, double* x, double* t1_x,
                        double* y, double* t1_y).
```

Superscripts of tangent-linear subroutine names and tangent-linear variable names are replaced with the prefix t1_, for example, $\mathbf{v}^{(1)} \equiv$ t1_v.

The entire Jacobian can be accumulated by letting $\mathbf{x}^{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$.

Linearized DAGs can be derived for multivariate vector functions $\mathbf{y} = F(\mathbf{x})$ at various levels of granularity depending on which parts of the computation are considered to be elemental. The most high-level view is shown in Figure 2.1 (a). Its tangent-linear extension represents the augmentation with the auxiliary variable s and the corresponding local partial derivative $\mathbf{x}^{(1)}$. It is shown in Figure 2.1 (b). The tangent-linear model follows immediately from the application of the chain rule on linearized DAGs (see (1.5)) to Figure 2.1 (b).

2.1.1 Tangent-Linear Code by Forward Mode AD

An early and intuitive entry to forward mode AD can be found in [60].

Forward mode AD is based on a conceptual augmentation of the SAC statements with the partial derivatives of each φ_j with respect to all its arguments $v_i, i \prec j$. According to the chain rule, directional derivatives $v_j^{(1)}$ of v_j in direction $\mathbf{x}^{(1)} = (v_0^{(1)}, \ldots, v_{n-1}^{(1)})$ are propagated in parallel with the function values as stated in the following theorem.

Theorem 2.6. *The tangent-linear model* $\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$ *of a program implementing* $\mathbf{y} = F(\mathbf{x})$, $F : \mathbb{R}^n \to \mathbb{R}^m$, *as in Definition* 1.5, *is evaluated for given inputs* $(v_0, \ldots, v_{n-1}) = \mathbf{x}$ *by the following recurrence:*

$$\begin{aligned} &For\ j = n, \ldots, n+p+m-1, \\ &\qquad v_j^{(1)} = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_i^{(1)}, \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j}. \end{aligned} \tag{2.2}$$

All SAC statements are preceded by local tangent-linear models as defined in (2.1). *The directional derivative of* $\mathbf{y}$ *with respect to* $\mathbf{x}$ *is returned as*

$$\mathbf{y}^{(1)} = (v_{n+p}^{(1)}, \ldots, v_{n+p+m-1}^{(1)}) \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}.$$

Proof. The following proof is motivated by a similar argument in [36]. An extended version

$$\mathcal{F} : \mathbb{R}^{n+p+m} \to \mathbb{R}^{n+p+m}$$

of F is considered whose elemental functions are mappings of the entire SAC memory space onto itself as

$$\mathbf{v}_{p+m} = \mathcal{F}(\mathbf{v}_0),$$

where $\mathbf{v}_0, \mathbf{v}_{p+m} \in \mathbb{R}^{n+p+m}$. The corresponding extended SAC becomes

$$\mathbf{v}_j = \Phi_j(\mathbf{v}_{j-1}) \quad \text{for } j = 1, \ldots, p+m,$$

where $\mathbf{v}_j \equiv (v_i^j)_{i=0,\ldots,n+p+m-1}$ and

$$v_k^j \equiv \begin{cases} \varphi_k(v_i^{j-1})_{i \prec k} & \text{if } k-n = j, \\ v_k^{j-1} & \text{otherwise} \end{cases}$$

for $k = n, \ldots, n+p+m-1$. The elemental functions φ_k are the same as in Definition 1.5. Initialization of $\mathbf{v}_0 = (x_0, \ldots, x_{n-1}, 0 \ldots, 0)$ yields

$$\mathbf{v}_{p+m} = (x_0, \ldots, x_{n-1}, v_n, \ldots, v_{n+p-1}, y_0, \ldots, y_{m-1}).$$

The two functions F and $\mathcal{F}$ are mathematically equivalent in the sense that they both compute $\mathbf{y}$ as a function of $\mathbf{x}$. The extended function keeps all the intermediate results explicitly.

We denote by $\Phi_{i+1}(\Phi_i(\mathbf{v}_{i-1}))$ the application of Φ_{i+1} to the result of Φ_i at point $\mathbf{v}_{i-1}$ or, equivalently, by $\Phi_{i+1} \circ \Phi_i(\mathbf{v}_{i-1})$. From

$$\mathcal{F}(\mathbf{v}_0) = \Phi_{p+m} \circ \Phi_{p+m-1} \circ \cdots \circ \Phi_1(\mathbf{v}_0),$$

it follows by the chain rule that

$$\nabla \mathcal{F}(\mathbf{v}_0) = \nabla \Phi_{p+m}(\mathbf{v}_{p+m-1}) \cdot \nabla \Phi_{p+m-1}(\mathbf{v}_{p+m-2}) \cdot \cdots \cdot \nabla \Phi_1(\mathbf{v}_0),$$

where

$$\nabla\Phi_k(\mathbf{v}_{k-1}) \equiv (\nabla\varphi_{ji}^k)_{i,j=0,\ldots,n+p+m-1}$$

and

$$\nabla\varphi_{ji}^k = \begin{cases} c_{ji} & \text{if } k-n=j \text{ and } i \prec j, \\ 1 & \text{if } k-n \neq j \text{ and } j=i, \\ 0 & \text{otherwise} \end{cases}$$

for $k = n,\ldots,n+p+m-1$. It follows that the product of the extended Jacobian $\nabla\mathcal{F}(\mathbf{v}_0)$ with a direction $\mathbf{v}_0^{(1)} \in \mathbb{R}^{n+p+m}$ can be computed as

$$\begin{aligned} \mathbf{v}_{p+m}^{(1)} &= \nabla\mathcal{F}(\mathbf{v}_0)\cdot\mathbf{v}_0^{(1)} \\ &= \nabla\Phi_{p+m}(\mathbf{v}_{p+m-1})\cdot\left(\nabla\Phi_{p+m-1}(\mathbf{v}_{p+m-2})\cdot\cdots\cdot\left(\nabla\Phi_1(\mathbf{v}_0)\cdot\mathbf{v}_0^{(1)}\right)\cdots\right). \end{aligned}$$

The forward mode of AD is obtained immediately from the last equation by considering single elements of the $\mathbf{v}_j^{(1)}$ for $j = 0,\ldots,p+m$. □

The choice of $\mathbf{x}$ uniquely determines the flow of control in the given implementation of F as well as in its tangent-linear version. Single assignment code can be generated for each assignment separately, making (2.2) applicable to arbitrary (intra- and interprocedural) flow of control. The correctness of this approach follows immediately from the chain rule. We formulate these observations as a set of rules, each of which is illustrated by an example. Thus we aim to provide a *cook book* that helps to produce tangent-linear code for arbitrary numerical programs. Advanced features provided by modern programming languages will require careful adaptation of these rules. This process is not expected to pose any conceptual difficulties.

Frequently, special care must be taken when implementing even seemingly simple mathematical ideas such as forward mode AD on a computer. Difficulties may arise from the conceptual difference between mathematical variables (e.g., the SAC variables in Theorem 2.6) and declared program variables that represent memory locations in the implementation. As previously mentioned, a single program variable may represent several mathematical variables by storing their values in the same memory location. Values of mathematical variables get lost due to overwriting. Program variables become invalid when leaving their scope. The associated memory can be reassigned by the operating system, thus overwriting its contents or making it inaccessible (depending on the programming language in use). While the implications turn out to be rather straightforward for forward mode AD, they will cause substantial trouble when implementing the only slightly more mathematically complicated reverse mode.

Tangent-Linear Code Generation Rule 1: Duplicate Active Data Segment

Tangent-linear code augments the original computation with the evaluation of directional derivatives. Hence any stored *active* value in the original program must be matched by a memory location storing the corresponding derivative. This statement applies to global as well as local and temporary SAC variables.

Definition 2.7. *We refer to a computed value as* active *if it depends on the value of at least one independent variable. Additionally, it must have an impact on the value of at least one*

dependent variable. A program variable is active if it cannot be proven to never hold an active value. Otherwise it is passive [38].

Signatures of subprograms must be extended with tangent-linear arguments for all active parameters. Their *intent* (input, output, or both) remains unchanged. For example, the tangent-linear version of

```
double g;

void f(double& x, double& y) {
  y=x+2*g;
}
```

becomes

```
double g, t1_g;

void t1_f(double& x, double& t1_x, double& y, double& t1_y) {
  double v0, t1_v0;
  double v1, t1_v1;
  double v2, t1_v2;
  double v3, t1_v3;

  t1_v0=t1_x; v0=x;
  t1_v1=t1_g; v1=g;
  t1_v2=2*t1_v1; v2=2*v1;
  t1_v3=t1_v0+t1_v2; v3=v0+v2;
  t1_y=t1_v3; y=v3;
}
```

Throughout this book we assume that subroutine arguments are passed by reference; this is indicated by the & character in C++. Arrays are assumed to be passed as pointers to their respective first entry. Issues arising from the fact that parameters are passed by value (e.g., in C/C++) or are marked as input-only or output-only (e.g., in Fortran) are beyond the scope of this book. Otherwise, special treatment becomes necessary in the context of adjoint code.

Copy propagation [2] simplifies the tangent-linear code to

```
void t1_f(double& x, double& t1_x, double& y, double& t1_y) {
  double v2, t1_v2;
  double v3, t1_v3;

  t1_v2=2*t1_g; v2=2*g;
  t1_v3=x+t1_v2; v3=x+v2;
  t1_y=t1_v3; y=v3;
}
```

or even further to

```
void t1_f(double& x, double& t1_x, double& y, double& t1_y) {
  t1_y=t1_x+2*t1_g;
  y=x+2*g;
}
```

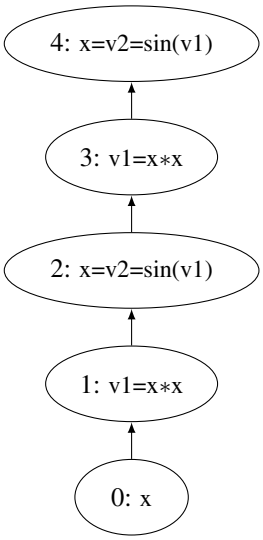


Figure 2.2. *Tangent-Linear Code Generation Rule* 2.

In the following we avoid copying the inputs $v_i = x_i$, for $i = 0,\ldots,n-1$, explicitly, thus generating only SAC variables v_j for $j = n,\ldots,n+p+m-1$. We always generate a SAC variable to hold the result of the right-hand side of each assignment, even for very simple cases such as $x = \sin(x)$, in order to break potential *aliasing* (reference to the same memory location) between variables on the right-hand side and the variable on the left-hand side of the assignment. This convention will simplify the formalization of rules for adjoint code generation in Section 2.2.1.

It is generally impossible to construct examples that isolate a specific issue addressed by the given derivative code rule. Most code fragments can be used to illustrate several transformation techniques.

Tangent-Linear Code Generation Rule 2: Assignment-Level SACs and TLMs

Intraprocedural control flow will almost certainly prevent us from building the *global* SAC of the entire program at compile time. For example, unrolling the **for**-loop in

```
void f(int n, double& x) {
  for (int i=0;i<n;i++) x=sin(x*x);
}
```

is impossible for an unknown n. The chain rule allows us to restrict the building of SAC to static code fragments such as individual assignments or sequences thereof, which are also referred to as *basic blocks*. The same program variable may represent multiple global SAC variables. For example, in the tangent-linear code

```
void t1_f(int n, double& x, double& t1_x) {
  double v1, t1_v3;
  double v2, t1_v2;
```

```
  for (int i=0;i<n;i++) {
    t1_v1=2*x*t1_x; v1=x*x;
    t1_v2=cos(v1)*t1_v1; v2=sin(v1);
    t1_x=t1_v2; x=v2;
  }
}
```

the memory location accessed through x=v2 may hold v_0, v_2, v_4, and so forth. Similarly, v_1, v_3, v_5,... are stored in v1. Refer to Figure 2.2 for graphical illustration.

Tangent-Linear Code Generation Rule 3: Interprocedural Tangent-Linear Code

Subroutine calls are simply replaced by calls of their tangent-linear versions. The correctness of this approach follows immediately from inlining the respective tangent-linear subroutine calls. Consider, for example, the following interprocedural version of the code used to illustrate Rule 2:

```
void g(double& x) {
  x=x*x;
}
void f(int n, double& x) {
  for (int i=0;i<n;i++) {
    g(x);
    x=sin(x);
  }
}
```

The square operation has been extracted into the subroutine g. In the tangent-linear code, the call to g is simply replaced by a call to its tangent-linear version t1_g:

```
void t1_g(double& x, double& t1_x) {
  t1_x=2*x*t1_x;
  x=x*x;
}
void t1_f(int n, double& x, double& t1_x) {
  for (int i=0;i<n;i++) {
    t1_g(x,t1_x);
    t1_x=cos(x)*t1_x;
    x=sin(x);
  }
}
```

Example 2.8 Consider the implementation of (1.2) given in Section 1.1.2. Figure 2.3 shows the corresponding linearized DAG; Figure 2.4 shows its tangent-linear extension for $n = 3$. A tangent-linear version of the code is the following:

```
void t1_f(int n, double* x, double* t1_x,
                 double& y, double& t1_y) {
  t1_y=0; y=0;
  for (int i=0;i<n;i++) {
```

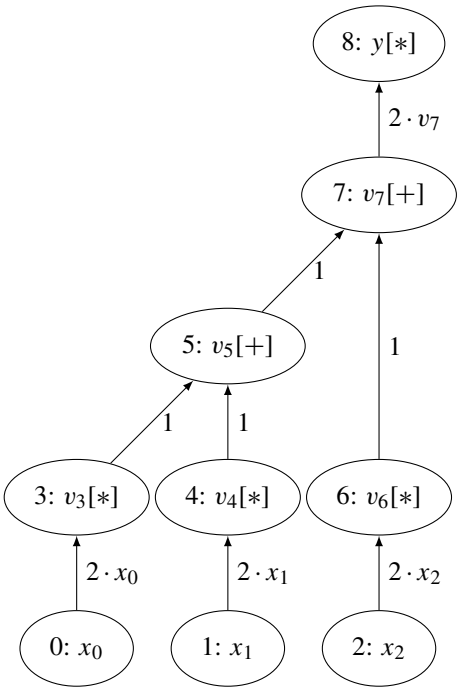


Figure 2.3. *Linearized DAG of* $y = (\sum_{i=0}^{n-1} x_i^2)^2$ *for* $n = 3$*; a single argument* v *of a multiplication denotes the square operation* $v \cdot v$*.*

```
        t1_y=t1_y+2*x[i]*t1_x[i];
        y=y+x[i]*x[i];
    }
    t1_y=2*y*t1_y;
    y=y*y;
}
```

The following driver accumulates the gradient entry-by-entry using the tangent-linear function t1_f.

```
void driver(int n, double* x, double* g) {
  double y;
  double* t1_x=new double[n];
  for (int i=0;i<n;i++) t1_x[i]=0;
  for (int i=0;i<n;i++) {
    t1_x[i]=1;
    t1_f(n, x, t1_x, y, g[i]);
    t1_x[i]=0;
  }
  delete [] t1_x;
}
```

A total of n calls to the tangent-linear routine is required to compute the full gradient. ∎

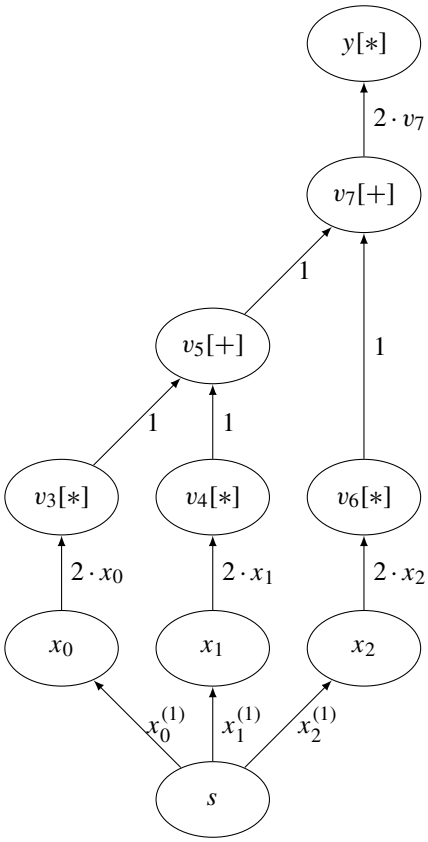


Figure 2.4. *Tangent-linear extension of the linearized DAG of the function* $y = (\sum_{i=0}^{n-1} x_i^2)^2$ *for* $n = 3$.

Table 2.1. *Run times for tangent-linear code (in seconds). n function evaluations are compared with n evaluations of the tangent-linear code required for a full gradient accumulation. The compiler optimizations are either switched off (*`g++ -O0`*) or the full set of optimizations is enabled (*`g++ -O3`*); refer to the* `g++` *manual pages for documentation on the different optimization levels. We observe a difference of a factor* $\mathcal{R}$ *of less than* 2 *when comparing the run time of a single run of the tangent-linear code with that of an original function evaluation. Full compiler optimization reduces* $\mathcal{R}$ *to about* 1.2 *as shown in the rightmost column.*

	`g++ -O0`			`g++ -O3`			$\mathcal{R}$
n	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t1_f	1.4	5.6	22.2	0.2	0.9	3.7	≈ 1.2

2.1.2 Tangent-Linear Code by Overloading

A very natural and convenient way to implement forward mode AD is by definition of an augmented data type containing $v^{(1)}$ in addition to v for all variables (program variables as well as auxiliary variables generated by the compiler). Directional derivatives are propagated by replacing all arithmetic operations and intrinsic functions with routines for computing both the value and the derivative. A simple type change of all *active* floating-point variables carrying nontrivial derivative information to the new augmented data type is often the only source code modification if the target programming language (such as C++) supports function and operator overloading.[6] For example, $u = v \cdot w$ becomes $(u = v \cdot w, u^{(1)} = v^{(1)} \cdot w + v \cdot w^{(1)})$ and $u = \sin(v)$ is modified into $(u = \sin(v), u^{(1)} = \cos(v) \cdot v^{(1)})$, where u, v, w are floating-point variables.

AD by overloading is implemented by our C++ library `dco` (derivative code by overloading). The source code of version 0.9 is listed in Appendix A. It serves as an illustration of the concepts discussed in this chapter as well as in Chapter 3. The production version 1.0 features a variety of advanced optimization techniques whose discussion is beyond the scope of this introductory text. Its performance exceeds that of version 0.9 significantly.

For tangent-linear scalar mode AD, a class dco_t1s_type (`dco`'s tangent-linear 1st-order scalar type) is defined with **double** precision members v (value) and t (tangent).

```
class dco_t1s_type {
  public :
    double v;
    double t;
    dco_t1s_type(const double&);
    dco_t1s_type();
    dco_t1s_type& operator=(const dco_t1s_type&);
};
```

A special constructor (dco_t1s_type(**const double**&)) converts passive into active variables at run time. The provided standard constructor simply initializes the value and derivative components to zero. The assignment operator returns a copy of the right-hand side unless it is aliased with the left-hand side of the assignment.

```
dco_t1s_type& dco_t1s_type::operator=(const dco_t1s_type& x) {
  if (this==&x) return *this;
  v=x.v; t=x.t;
  return *this;
}
```

Implementations of all relevant arithmetic operators and intrinsic functions are required, for example,

```
dco_t1s_type operator*(const dco_t1s_type& x1,
                       const dco_t1s_type& x2) {
  dco_t1s_type tmp;
  tmp.v=x1.v*x2.v;
  tmp.t=x1.t*x2.v+x1.v*x2.t;
```

[6] More substantial modifications may become necessary in languages that do not have full support for object-oriented programming.

```
    return tmp;
  }
```

and

```
  dco_t1s_type sin(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=sin(x.v);
    tmp.t=cos(x.v)*x.t;
    return tmp;
  }
```

Refer to Section A.1 for a more complete version of the source code. The driver program in Listing 2.1 uses the implementation of **class** dco_t1s_type to compute the gradient of (1.2) for n = 4 at the point $x_i = 1$ for $i = 0,\ldots,3$. Four evaluations of the tangent-linear routine

```
void f(dco_t1s_type *x, dco_t1s_type &y)
```

are performed with the derivative components of x initialized to the Cartesian basis vectors in $\mathbb{R}^4$.

Listing 2.1. *Driver for tangent-linear code by overloading.*

```
#include<iostream>
using namespace std;
#include "dco_t1s_type.hpp"

const int n=4;

void f(dco_t1s_type *x, dco_t1s_type &y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  y=y*y;
}

int main() {
  dco_t1s_type x[n], y;
  for (int i=0;i<n;i++) x[i]=1
  for (int i=0;i<n;i++) {
    x[i].t=1;
    f(x,y);
    x[i].t=0;
    cout << y.t << endl;
  }
  return 0;
}
```

Let **class** dco_t1s_type be defined in the C++ source files `dco_t1s_type.hpp` and `dco_t1s_type.cpp`, and let the driver program be stored as `main.cpp`. An executable is built by calling

```
$(CPPC) -c dco_t1s_type.cpp
$(CPPC) -c main.cpp
$(CPPL) -o main dco_t1s_type.o main.o
```

Table 2.2. *Run times for tangent-linear code by overloading (in seconds). n function evaluations are compared with n evaluations of the tangent-linear code required for a full gradient accumulation. With the full set of compiler optimizations enabled, we observe a factor $\mathcal{R}$ of less than* 10 *when comparing the run time of a single run of the tangent-linear code with that of an original function evaluation in the right-most column. The overloading solution turns out to be more than* 5 *times slower than the hand-written tangent-linear code due to less effective compiler optimization of the overloaded code.*

	g++ -O0			g++ -O3			$\mathcal{R}$
n	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t1_f	7.7	29.5	116.5	1.9	7.1	28.3	≈ 9.1

where `$(CPPC)` and `$(CPPL)` should be replaced by a C++ compiler and a corresponding linker, respectively (for example, `g++`). Run time measurements are reported in Table 2.2.

When computing several directional derivatives at the same time, it is favorable to evaluate the function and its local partial derivatives only once, followed by products of the latter with vectors of directional derivatives in correspondence with the chain rule. This *vector forward mode* of AD can be implemented by overloading all intrinsic functions and arithmetic operators for the user-defined data type

```
class dco_t1v_type {
  public :
    double v;
    double *t;
    ...
};
```

The t component becomes a vector whose size is set at run time and stored in the static variable dco_t1v_type :: t_length . All overloaded functions and arithmetic operators are modified accordingly, for example,

```
dco_t1v_type operator*(const dco_t1v_type& x1, const
    dco_t1v_type& x2) {
  dco_t1v_type tmp;
  tmp.v=x1.v*x2.v;
  for (int i=0;i<dco_t1v_type::t_length;i++)
    tmp.t[i]=x1.t[i]*x2.v+x1.v*x2.t[i];
  return tmp;
}
```

All constructors need to allocate the vector t; the destructor deallocates it. This dynamic allocation and deallocation is performed as part of each arithmetic operation or intrinsic function call, which results in a significant and potentially infeasible run-time overhead. Alternatively, one may chose to allocate t statically. A recompilation of dco_t1v_type may be necessary whenever the required size of t changes. As usual in AD, the implementation of an efficient library (here for vector forward mode) needs to take into account the characteristics of the given computing platform including hardware (CPU, memory hierarchy, i/o system) and system software (operating system, memory management, native compiler, run-time support libraries).

2.1.3 Seeding and Harvesting Tangent-Linear Code

Pioneering work on the exploitation of sparsity in Jacobians was presented in [21]. A comprehensive survey of partitioning and coloring techniques for exploiting sparsity in Jacobians and Hessians is given in [30]. The authors discuss both *direct* and *indirect* methods and give a large number of essential references. We focus on direct methods as a good starting point for the successful use of the potentially more efficient indirect methods.

The jth column of $\nabla F(\mathbf{x})$ can be approximated by (forward) finite differences as follows:

$$\nabla F_{*,j} \equiv \frac{\partial F}{\partial x_j} \approx \frac{F(\mathbf{x}+h\cdot\mathbf{e}^j)-F(\mathbf{x})}{h}. \tag{2.3}$$

In order to minimize the number of function evaluations required, we intend to avoid the computation of statically known entries. Without loss of generality, such entries are assumed to be zeros. Hence, knowledge of the sparsity pattern of the Jacobian is the key to all results in this section.

Potential savings in the run time of Jacobian accumulation result from the observation that the columns of the Jacobian can be partitioned into mutually disjoint subsets $\mathcal{I}_1,\ldots,\mathcal{I}_l$ of the column index set $\{0,\ldots,n-1\}$. Any two columns $\mathbf{u}\equiv\nabla F_{*,i}$ and $\mathbf{v}\equiv\nabla F_{*,j}$ that belong to the same subset $\mathcal{I}_h$, $h\in\{1,\ldots,l\}$, are assumed to be *structurally orthogonal*; that is, $\nexists k : u_k\neq 0\wedge v_k\neq 0$. Hence the desired entries of all columns in a given set $\mathcal{I}_j$ can be computed simultaneously as

$$\nabla F_{*,\mathcal{I}_j} \approx \frac{F\left(\mathbf{x}+h\cdot\sum_{i\in\mathcal{I}_j}\mathbf{e}^i\right)-F(x)}{h}.$$

As a relevant example, the authors of [21] discuss band matrices of band width w. Obviously, any two columns $\nabla F_{*,i}$ and $\nabla F_{*,j}$ with $|j-i|>w$ are structurally orthogonal. Moreover, a sequential greedy approach is proposed as a heuristic for determining a feasible (structurally orthogonal but not necessarily optimal in terms of a minimal value for the number of index sets l) partitioning of the columns. When considering the ith column, the remaining columns $\nabla F_{*,j}$, $i<j<n$, are tested for structural orthogonality with ∇F_i. If the test is successful, then j becomes part of the same partition. This procedure is run iteratively for increasing $i=0,1,\ldots$ until no more columns remain unassigned.

An obvious lower bound for l is the maximum number of nonzero elements in any single row in the Jacobian. Sequential partitioning reaches this lower bound for band matrices. However, as shown in the following example, it does not produce an optimal partitioning in all cases.

Example 2.9 Sequential partitioning applied to

$$\nabla F = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} \\ 0 & 0 & a_{1,2} & a_{1,3} \\ 0 & a_{2,1} & a_{2,2} & 0 \end{pmatrix} \tag{2.4}$$

results in $\mathcal{I}_1=\{0,1\}$, $\mathcal{I}_2=\{2\}$, and $\mathcal{I}_3=\{3\}$. A better solution is to partition as $\mathcal{I}_1=\{0,2\}$ and $\mathcal{I}_2=\{1,3\}$. ∎

The column partitioning problem applies both to finite difference approximation of the Jacobian and to Jacobian accumulation in tangent-linear mode AD. With the latter, we aim

to compute

$$B_t = A \cdot S_t, \tag{2.5}$$

where $A \equiv \nabla F(\mathbf{x})$ and $S_t \in \{0,1\}^{n \times l_t}$ such that $\forall a_{i,j} \neq 0 \; \exists b^t_{i,l} \in B_t : a_{i,j} = b^t_{i,l}$; that is, each nonzero entry $a_{i,j}$ of the Jacobian A must be present in B_t. The matrices S_t and B_t are referred to as the *seed matrix* and the *compressed Jacobian*, respectively. The number of columns in S_t is denoted by l_t (l_a will be used for adjoint seeding). The term *harvesting* refers to the retrieval of the uncompressed Jacobian matrix. Harvesting is performed by solving the system of simultaneous linear equations in (2.5). For direct methods, the solution is obtained by a simple substitution procedure. The combinatorial problem is to find a minimal l_t; the resulting coloring problems on various graph representations of the Jacobian are discussed in detail in [30]. The coloring problem is known to be NP-complete [28], which makes heuristics the preferred approach to the determination of a feasible, and hopefully close to optimal, partitioning of the columns of the Jacobian.

Example 2.10 Suppose that (2.4) results from a function $F : \mathbb{R}^4 \to \mathbb{R}^3$ implemented as

```
void f(int n, double *x, int m, double *y)
```

with a tangent-linear version

```
void t1_f(int n, double *x, double *t1_x,
          int m, double *y, double *t1_y)
```

generated by forward mode AD. The following example driver for computing the compressed Jacobian B_t uses the column partitioning $\mathcal{I}_1 = \{0,2\}$ and $\mathcal{I}_2 = \{1,3\}$.

```
int main() {
  double x[4]=...;
  double y[3], t1_y[3];
  {
    double t1_x[4]={1,0,1,0};
    t1_f(4,x,t1_x,3,y,t1_y); // columns 0 and 2
  }
  ...
  {
    double t1_x[4]={0,1,0,1};
    t1_f(4,x,t1_x,3,y,t1_y); // columns 1 and 3
  }
  ...
}
```

For the known sparsity pattern of ∇F and the resulting seed matrix S_t, all unknown nonzero entries $x_{j,i}$ are obtained by simple substitution from

$$\begin{pmatrix} x_{0,0} & 0 & 0 & x_{0,3} \\ 0 & 0 & x_{1,2} & x_{1,3} \\ 0 & x_{2,1} & x_{2,2} & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,3} \\ a_{1,2} & a_{1,3} \\ a_{2,2} & a_{2,1} \end{pmatrix}.$$

Obviously, no arithmetic operations are required to retrieve $x_{j,i} = a_{j,i}$ for $j = 0,1,2$ and $i = 0,1,2,3$. ∎

2.2 Adjoint Model

The computational complexity of the tangent-linear approach to computing the first derivative of a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ grows with the number of independent variables n. This number can become extremely large for many real-word applications. For example, n is often proportional to the number of grid points used for the discretization of some physical domain (ocean, atmosphere, surface of a car or airplane, etc.). The evaluation of ∇F in tangent-linear mode (just as with finite difference approximation) becomes infeasible if, for example, a single function evaluation takes only one minute and $n = 10^6$. Neither developers nor users of nonlinear optimization software are willing to spend almost two years (10^6 minutes) waiting for a single gradient evaluation. In the case where $m = 1$, an adjoint code can return the same gradient in less than 10 minutes. This speedup, which may involve a factor of one million or greater, is certainly reason enough to take a closer look at adjoint models.

In functional analysis, adjoint operators are typically defined on Hilbert spaces with a suitable inner product; see [25]. Our focus is on linear operators represented by derivative tensors of first and higher order on $\mathbb{R}^n$ (more precisely on IF^n where IF denotes the floating-point numbers representable on the given computer architecture). The adjoint of the Jacobian forms the basis of this approach. Operators induced by higher derivative tensors will, due to symmetry, turn out to be *self-adjoint*. Refer to Chapter 3 for details.

Definition 2.11. *The adjoint of the linear operator $\nabla F : \mathbb{R}^n \to \mathbb{R}^m$ is defined as $(\nabla F)^* : \mathbb{R}^m \to \mathbb{R}^n$ where*

$$\langle (\nabla F)^* \cdot \mathbf{y}_{(1)}, \mathbf{x}^{(1)} \rangle_{\mathbb{R}^n} = \langle \mathbf{y}_{(1)}, \nabla F \cdot \mathbf{x}^{(1)} \rangle_{\mathbb{R}^m}, \tag{2.6}$$

and where $\langle .,. \rangle_{\mathbb{R}^n}$ and $\langle .,. \rangle_{\mathbb{R}^m}$ denote the scalar products in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively.

Theorem 2.12. $(\nabla F)^* = (\nabla F)^T$.

Proof. Let $A \equiv \nabla F(\mathbf{x})$ and $A = (a_{j,i})$ with $j = 0,\ldots,m-1$ and $i = 0,\ldots,n-1$. Set $A^T = (a^T_{i,j})$.

$$\begin{aligned}
\langle (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)}, \mathbf{x}^{(1)} \rangle_{\mathbb{R}^n} &= \sum_{i=0}^{n-1} x_i^{(1)} \cdot \sum_{j=0}^{m-1} a^T_{i,j} \cdot y_{(1)j} \\
&= {}_{[D]} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_i^{(1)} \cdot a^T_{i,j} \cdot y_{(1)j} \\
&= {}_{[K+]} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} x_i^{(1)} \cdot a^T_{i,j} \cdot y_{(1)j} \\
&= {}_{[K\cdot]} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} y_{(1)j} \cdot a^T_{i,j} \cdot x_i^{(1)} \\
&= {}_{[D]} \sum_{j=0}^{m-1} y_{(1)j} \cdot \sum_{i=0}^{n-1} a^T_{i,j} \cdot x_i^{(1)}
\end{aligned}$$

$$= {}_{[(A^T)^T]} \sum_{j=0}^{m-1} y_{(1)j} \cdot \sum_{i=0}^{n-1} a_{j,i} \cdot x_i^{(1)}$$
$$= \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \rangle_{\mathbb{R}^m}.$$

We have used distributivity $[D]$, commutativity of addition $[K+]$ and multiplication $[K\cdot]$, and the fact that $(A^T)^T = A$. With (2.1) and (2.7) it follows that $\langle \mathbf{x}_{(1)}, \mathbf{x}^{(1)} \rangle_{\mathbb{R}^n} = \langle \mathbf{y}_{(1)}, \mathbf{y}^{(1)} \rangle_{\mathbb{R}^m}$. □

An immediate consequence of Definition 2.11 is the following: If the adjoint of the output $\mathbf{y}_{(1)}$ is chosen orthogonal to the directional derivative $\mathbf{y}^{(1)} = \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}$, then the adjoint of the input $\mathbf{x}_{(1)} = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$ is orthogonal to $\mathbf{x}^{(1)}$.

Definition 2.13. *The Jacobian $\nabla F = \nabla F(\mathbf{x})$ induces a linear mapping $\mathbb{R}^m \to \mathbb{R}^n$ defined by*

$$\mathbf{y}_{(1)} \mapsto \nabla F^T \cdot \mathbf{y}_{(1)}.$$

The function $F_{(1)} : \mathbb{R}^{n+m} \to \mathbb{R}^n$ defined as

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \tag{2.7}$$

is referred to as the adjoint model *of F.*

Adjoints are defined as partial derivatives of an auxiliary scalar variable t with respect to $\mathbf{y}$ and $\mathbf{x}$ where

$$\mathbf{y}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{y}}$$

and

$$\mathbf{x}_{(1)} \equiv \frac{\partial t}{\partial \mathbf{x}}.$$

By the chain rule, we get

$$\mathbf{x}_{(1)} \equiv \left(\frac{\partial t}{\partial \mathbf{x}}\right)^T = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \cdot \left(\frac{\partial t}{\partial \mathbf{y}}\right)^T = \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}.$$

A graphical illustration in the form of the *adjoint extension* of the linearized DAG for $\mathbf{y} = F(\mathbf{x})$ is shown in Figure 2.5. The adjoint extension of the linearized DAG of (1.2) is displayed in Figure 2.6.

The derivative code compiler `dcc` transforms the given implementation

```
void f(int n, int m, double* x, double* y)
```

of the function $\mathbf{y} = F(\mathbf{x})$ into *adjoint code* $(\mathbf{y}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}) = F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$, which computes

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}), \\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)}, \\ \mathbf{y}_{(1)} &= 0. \end{aligned}$$

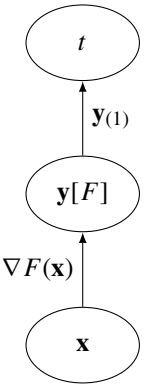


Figure 2.5. *Adjoint extension of the linearized DAG for* $\mathbf{y} = F(\mathbf{x})$.

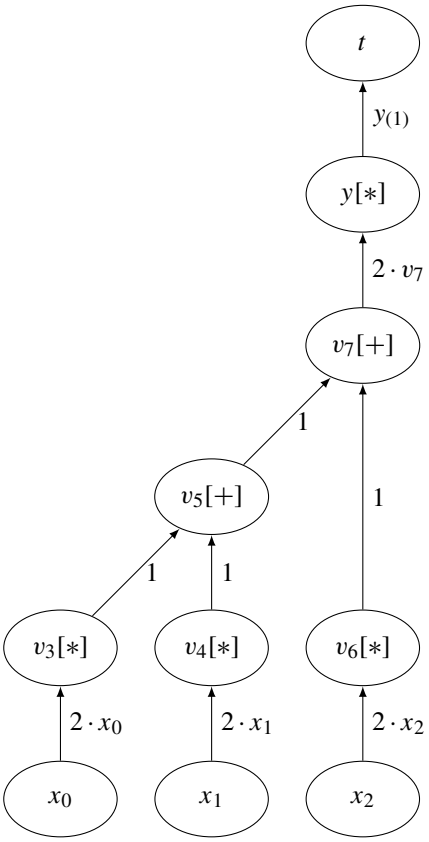


Figure 2.6. *Adjoint extension of the linearized DAG of the function* $y = (\sum_{i=0}^{n-1} x_i^2)^2$ *with* $n = 3$.

The signature of the resulting adjoint subroutine is the following:

```
void a1_f(int n, int m, double* x, double* a1_x,
                        double* y, double* a1_y).
```

Subscripts of adjoint subroutine names and adjoint variable names are replaced with the prefix a1_, such as $\mathbf{v}_{(1)} \equiv$ a1_v. The entire Jacobian is accumulated by letting $\mathbf{y}_{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^m$. There is no approximate model for adjoints as there is for directional derivatives in the form of finite differences.

2.2.1 Adjoint Code by Reverse Mode AD

Early work on reverse mode AD includes [56]. The author uses the simple function

$$y = f(\mathbf{x}) = \prod_{i=0}^{n-1} x_i$$

to illustrate the ability of the reverse mode to compute gradients cheaply (with a computational cost that exceeds by a small constant factor that of the pure function evaluation). This function is known as *Speelpenning's example* and is used extensively to illustrate the power of reverse mode AD. Equation (1.2) exhibits similar properties while being better suited for the discussion of unconstrained nonlinear optimization methods in Chapter 1.

Let us take a closer look at the structure of the adjoint code that is generated by reverse mode AD.

Theorem 2.14. *For given adjoints of the dependent variables, (nonincremental) reverse mode AD propagates adjoints backwards through the SAC as follows:*

$$\begin{aligned} &\text{For } j = n, \ldots, n+p+m-1 \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \\ &\text{for } i = n+p-1, \ldots, 0 \\ &\quad v_{(1)i} = \sum_{j:i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_{(1)j}. \end{aligned} \tag{2.8}$$

The $v_{(1)j}$ are assumed to be initialized to $y_{(1)j}$ for $j = n+p, \ldots, n+p+m-1$. A forward evaluation of the SAC is performed to compute all intermediate variables whose values are required in reverse order for the adjoint propagation. In the second part of (2.8) *the elemental functions in the SAC are processed in reverse order.*

Proof. In [36] the authors consider the same extended version

$$\mathcal{F} : \mathbb{R}^{n+p+m} \to \mathbb{R}^{n+p+m}$$

of F as in the proof of Theorem 2.6. From

$$\nabla \mathcal{F}(\mathbf{v}_0) = \nabla \Phi_{p+m}(\mathbf{v}_{p+m-1}) \cdot \nabla \Phi_{p+m-1}(\mathbf{v}_{p+m-2}) \cdot \cdots \cdot \nabla \Phi_1(\mathbf{v}_0)$$

we immediately have

$$\nabla \mathcal{F}(\mathbf{v}_0)^T = \nabla \Phi_1(\mathbf{v}_0)^T \cdot \nabla \Phi_2(\mathbf{v}_1)^T \cdot \cdots \cdot \nabla \Phi_{p+m}(\mathbf{v}_{p+m-1})^T,$$

and hence

$$\begin{aligned} \mathbf{v}_{(1)0} &= \nabla \mathcal{F}(\mathbf{v}_0)^T \cdot \mathbf{v}_{(1)p+m} \\ &= \nabla \Phi_1(\mathbf{v}_0)^T \cdot \nabla \Phi_2(\mathbf{v}_1)^T \cdot \cdots \cdot \nabla \Phi_{p+m}(\mathbf{v}_{p+m-1})^T \cdot \mathbf{v}_{(1)p+m}. \end{aligned}$$

Evaluation as

$$\mathbf{v}_{(1)0} = (\nabla \Phi_1(\mathbf{v}_0)^T \cdot (\nabla \Phi_2(\mathbf{v}_1)^T \cdot (\cdots (\nabla \Phi_{p+m}(\mathbf{v}_{p+m-1})^T \cdot \mathbf{v}_{(1)p+m}) \ldots))$$

yields (2.8). □

Our *cook book* for differentiating computer programs needs to be extended in order to make Theorem 2.14 applicable to real code. Again, advanced features of modern programming languages will require careful adaptation of these rules; their comprehension, as well as the ability to apply them to syntactically and semantically simpler code, is a crucial prerequisite for a successful generation of adjoint versions of real-world simulation programs.

Adjoint Code Generation Rule 1: Duplicate Active Data Segment

An adjoint code generator augments the original computation with the evaluation of adjoints for all SAC and program variables. Any stored active value in the original program, as well as any SAC variable, must be matched by a memory location to store the corresponding adjoint. Signatures of subprograms must be extended with adjoint arguments for all active parameters. Adjoint inputs become outputs and vice versa. For example, the adjoint version of

```
double g;

void f(double& x, double& y) {
  y=x+2*g;
}
```

becomes

```
double g, a1_g;

void a1_f(double& x, double& a1_x, double& y, double& a1_y) {
  double v2, a1_v2;
  double v3, a1_v3;

  // forward section
  v2=2*g; v3=x+v2; y=v3;
```

```
  // reverse section
  a1_v3=a1_y;
  a1_x=a1_v3; a1_v2=a1_v3;
  a1_g=2*a1_v2;
}
```

In the *forward section* of the adjoint code, the SAC computes all intermediate values that enter the computation of the local partial derivatives in (2.8). For the linear function given above the values of the SAC variables (v2, v3) are not required for the evaluation of the constant local partial derivatives. Hence, the construction of the SAC in the forward section is actually obsolete in this particular case.

Adjoints of all SAC and program variables (a1_v3, a1_v2, a1_x, a1_g) are computed as a function of the adjoint output a1_y in the *reverse section* of the adjoint code. Copy propagation combined with the observation from the previous paragraph yields an optimized version of this code as follows:

```
double g, a1_g;

void a1_f(double& x, double& a1_x, double& y, double& a1_y) {
  // forward section
  y=x+2*g;

  // reverse section
  a1_x=a1_y; a1_g=2*a1_y;
}
```

Adjoint Code Generation Rule 2: Increment and Reset Adjoint Program Variables

Consider the following implementation of a function $f : \mathbb{R}^2 \to \mathbb{R}^2$:

```
void f(double* x, double* y) {
  y[0]=sin(x[0]); y[1]=x[0]*x[1];
}
```

Its DAG is shown in Figure 2.7. The program variable x[0] appears on the right-hand side of the two assignments. In both cases, it represents the same mathematical variable (node

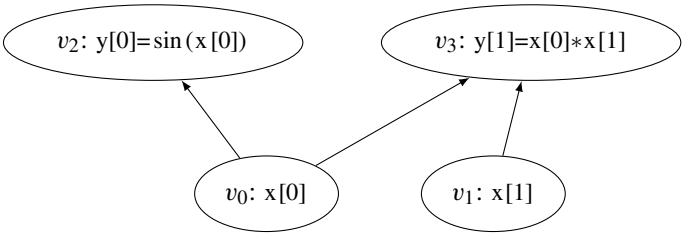


Figure 2.7. *Adjoint Code Generation Generation Rule* 2*: Increment adjoint program variables.*

in the DAG). Hence, the adjoint versions of both assignments contribute to the adjoint a1_x[0]. An implementation of the current definition of reverse mode AD may require access to information about two or more assignments when generating code for computing the adjoint of some program variable. In the current example, to generate

```
a1_x[0]=cos(x[0])*a1_y[0]+x[1]*a1_y[1];
```

we need to access the adjoints of both left-hand sides (a1_y[0], a1_y[1]) in addition to the arguments (x[0], x[1]) of the corresponding partial derivatives. Note that, in general, these assignments may not lie in close proximity in the code. Ideally, we would like to find a method that allows us to process the original assignments in strictly reverse order. That is, each adjoint assignment should *spread* its contributions to the adjoints of its right-hand side arguments instead of adjoint program variables having to *collect* them. The result is the following *incremental adjoint code*:

```
void a1_f(double* x, double* a1_x, double* y, double* a1_y) {
  double v2, a1_v2;
  double v3, a1_v3;
  // forward section
  v2=sin(x[0]);
  y[0]=v2;
  v3=x[0]*x[1];
  y[1]=v3;
  // reverse section
  a1_v3=a1_y[1]; a1_y[1]=0;
  a1_x[0]+=x[1]*a1_v3;
  a1_x[1]+=x[0]*a1_v3;
  a1_v2=a1_y[0]; a1_y[0]=0;
  a1_x[0]+=cos(x[0])*a1_v2;
}
```

The auxiliary variables (v2, v3) are each used exactly once. Consequently, their adjoints (a1_v2, a1_v3) are defined exactly once and hence do not need to be incremented. To avoid incrementation of invalid values, adjoints of program variables on left-hand sides of assignments need to be reset to zero after the corresponding adjoint assignments. Refer to the example used to explain Adjoint Code Generation Rule 4 for an illustration. Adjoint inputs (a1_x) are expected to be initialized by the calling routine. Further optimization of the adjoint code yields

```
void a1_f(double* x, double* a1_x, double* y, double* a1_y) {
  // forward section
  y[0]=sin(x[0]);
  y[1]=x[0]*x[1];
  // reverse section
  a1_x[0]+=x[1]*a1_y[1];
  a1_x[1]+=x[0]*a1_y[1]; a1_y[1]=0;
  a1_x[0]+=cos(x[0])*a1_y[0]; a1_y[1]=0;
}
```

The previous observations can be formalized in a manner similar to Theorem 2.14, yielding the *incremental reverse mode*

$$\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\qquad v_j = \varphi_j(v_i)_{i \prec j} \\ &\text{for } j = n+p+m-1, \ldots, n \\ &\qquad \left(v_{(1)i}\right)_{i \prec j} = \left(v_{(1)i}\right)_{i \prec j} + v_{(1)j} \cdot \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j} . \end{aligned} \tag{2.9}$$

The $v_{(1)j}$ are assumed to be initialized to $y_{(1)j}$ for $j = n+p, \ldots, n+p+m-1$ and to zero for $j = n, \ldots, n+p-1$, as they may represent both auxiliary variables (whose value is used exactly once) and program variables (whose value is potentially used several times). It is the user's responsibility to ensure correct initialization of the adjoint independent variables $v_{(1)j} = x_{(1)j}$, $j = 0, \ldots, n-1$, within the calling driver routine. Initialization to zero yields the accumulation of $\nabla F^T \cdot \mathbf{y}_{(1)}$ in $\mathbf{x}_{(1)}$ as an output of the (incremental) adjoint subroutine.

As a consequence of the transition to incremental adjoint code, it is required that adjoints of left-hand sides of assignments be set to zero immediately following the associated adjoint assignment. The overwritten memory location may have previously been used to store a right-hand side argument of an earlier assignment. Otherwise the corresponding adjoint assignment(s) would increment the wrong adjoint. For illustration, consider

```
void f(double& x) {
  double z;
  z=2*x; x=cos(z);
}
```

While the local variable z is in fact obsolete in this simple example, in a more complex situation it may well be used by subsequent computations. We keep it simple for the sake of clarity. Mechanical application of incremental reverse mode yields the following incorrect adjoint code:

```
void a1_f(double& x, double* a1_x) {
  double z, a1_z=0;
  double v1, a1_v1;
  double v2, a1_v2;

  // forward section
  v1=2*x;
  z=v1;
  v2=cos(z);
  x=v2;

  // reverse section
  a1_v2=a1_x;
  a1_z+=-sin(z)*a1_v2;
  a1_v1=a1_z;
  a1_x+=2*a1_v1;
}
```

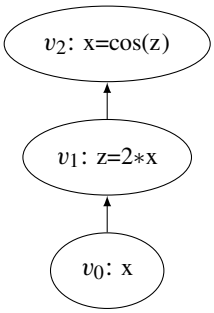


Figure 2.8. *Adjoint Code Generation Rule 2: Reset adjoint program variables.*

Adjoint program variables (a1_x, a1_z) are incremented. Adjoint local variables are initialized to zero (a1_z=0). Nevertheless, this code returns the wrong adjoint. The problem is the incrementation of a1_x by the last assignment. Refer to Figure 2.8 for illustration. In its current form, the code preserves the value of a1_x $\equiv v_{(1)2}$. However, according to (2.9) the last assignment is assumed to increment a1_x $\equiv v_{(1)0} = 0$, which fails because $v_{(1)0}$ and $v_{(1)2}$ share the same memory location. A feasible solution sets a1_x=0 immediately after the adjoint of x=v2. The correct adjoint code becomes

```
void a1_f(double& x, double* a1_x) {
  ...

  // reverse section
  a1_v2=a1_x; a1_x=0;
  ...
}
```

Adjoint Code Generation Rule 3: Store Overwritten Required Values

Consider the following implementation of a function $f : \mathbb{R} \to \mathbb{R}$:

```
void f(double& x) {
  x=sin(x*x);
}
```

Application of Adjoint Code Generation Rules 1 and 2 yields the following incorrect adjoint code:

```
1 void a1_f(double& x, double& a1_x) {
2   // SAC variables
3   double v=0, a1_v=0;
4
5   // forward section
6   v=x*x;
7   x=sin(v);
8
9   // reverse section
```

```
10   a1_v=cos(v)*a1_x; a1_x=0;
11   a1_x+=2*x*a1_v;
12 }
```

The problem lies in line 11, where the partial derivative of x∗x is evaluated incorrectly. The reason is not the expression itself; 2∗x is certainly correct. However, the value of x at this point is not what it should be due to overwriting in line 7. When evaluating the local partial derivative of the right-hand side of the assignment in line 6, we need the value of x before it is overwritten. Our preferred solution is to augment the forward section with statements that push any required (by the reverse section) value onto a stack before it is overwritten by the following assignment. We use stacks from the C++ standard library [42]. A single stack entry is required for our simple example, which yields

```
1 void a1_f(double& x, double& a1_x) {
2   // SAC variables
3   double v=0, a1_v=0;
4
5   // augmented forward section
6   v=x*x;
7   required_double.push(x); x=sin(v);
8
9   // reverse section
10  x=required_double.top(); required_double.pop();
11  a1_v=cos(v)*a1_x; a1_x=0;
12  a1_x+=2*x*a1_v;
13 }
```

Values of different data types may need to be stored within the augmented forward section. Floating-point values as well as integers and values of other data types may be required for a correct evaluation of the reverse section of the adjoint code. Consequently, several typed stacks may have to be provided. They are referred to as *required [double, integer, ...] data stacks*. As an immediate consequence of Theorem 2.14, data access is guaranteed to be LIFO (Last In First Out), making stacks the preferred data structure.

While these changes ensure that the adjoint is computed correctly, there is still one more problem to solve. As a result of restoring the input value of x in line 10, an incorrect function value is returned. If the latter is not used by the enclosing computation, then no further action is required. Otherwise, the function value(s) should be stored at the end of the augmented forward section, and subsequently should be restored at the end of the adjoint code. To this end, a *result checkpoint* is written. A correct adjoint code that fully satisfies all requirements is the following:

```
void a1_f(double& x, double& a1_x) {
  // SAC variables
  double v=0, a1_v=0;

  // augmented forward section
  v=x*x;
  required_double.push(x); x=sin(v);

  // store result
  result_double.push(x);
```

```
  // reverse section
  x=required_double.top(); required_double.pop();
  a1_v=cos(v)*a1_x; a1_x=0;
  a1_x+=2*x*a1_v;

  // restore result
  x=result_double.top(); result_double.pop();
}
```

Adjoint Code Generation Rule 4: Incomplete Assignment-Level SACs in Reverse Section

Consider the following modified version of the example that was used to illustrate Adjoint Code Generation Rule 3:

```
void f(double& x) {
  x=sin(x*x);
  x=sin(x*x);
}
```

A corresponding adjoint code is the following:

```
 1 void a1_f(double& x, double& a1_x) {
 2   // SAC variables
 3   double v=0, a1_v=0;
 4
 5   // augmented forward section
 6   v=x*x;
 7   required_double.push(x); x=sin(v);
 8   required_double.push(v); v=x*x;
 9   required_double.push(x); x=sin(v);
10
11   // store result
12   result_double.push(x);
13
14   // reverse section
15   x=required_double.top(); required_double.pop();
16   a1_v=cos(v)*a1_x; a1_x=0;
17   v=required_double.top(); required_double.pop();
18   a1_x+=2*x*a1_v;
19   x=required_double.top(); required_double.pop();
20   a1_v=cos(v)*a1_x; a1_x=0;
21   a1_x+=2*x*a1_v;
22
23   // restore result
24   x=result_double.top(); result_double.pop();
25 }
```

The value of v that is overwritten in line 8 is required by the adjoint of the assignment in line 7. Hence this value needs to be stored in addition to the instances of x in lines 7 and 9.

Note that without resetting a1_x to zero in lines 16 and 20 the wrong base values would be incremented in lines 18 and 21.

The amount of memory occupied by the required double data stack can be reduced by moving the construction of the assignment-level SACs to the reverse section of the adjoint code as follows:

```
1 void a1_f(double& x, double& a1_x) {
2   // SAC variables
3   double v=0, a1_v=0;
4
5   // augmented forward section
6   required_double.push(x); x=sin(x*x);
7   required_double.push(x); x=sin(x*x);
8
9   // store result
10  result_double.push(x);
11
12  // reverse section
13  x=required_double.top(); required_double.pop();
14  v=x*x; // incomplete SAC
15  a1_v=cos(v)*a1_x; a1_x=0;
16  a1_x+=2*x*a1_v;
17  x=required_double.top(); required_double.pop();
18  v=x*x; // incomplete SAC
19  a1_v=cos(v)*a1_x; a1_x=0;
20  a1_x+=2*x*a1_v;
21
22  // restore result checkpoint
23  x=result_double.top(); result_double.pop();
24 }
```

The forward sweep consists of the original statements augmented with code for storing all required data (lines 6 and 7). Adjoint versions are generated for all original assignments in reverse order; the corresponding stored data is recovered (lines 13 and 17), followed by the execution of the *incomplete SACs* (lines 14 and 18). The final assignment to the original right-hand side is omitted, as it would undo the previous recovery of the required values for x and thus lead to potentially incorrect adjoints. The adjoint statements (lines 15–16 and 19–20) remain unchanged.

For this simple example, while the savings in memory occupied by the required data stack is not impressive, they are likely to be significant for larger code. Further replication of the assignment x=sin(x*x) allows us to save a factor of 2, asymptotically. This number grows with growing right-hand sides of assignments.

Adjoint Code Generation Rule 5: Intraprocedural Control Flow Reversal

Consider the following implementation of (1.2):

```
1 void f(int n, double *x, double &y) {
2   int i=0;
3   y=0;
```

```
4    while (i<n) {
5      y=y+x[i]*x[i];
6      i=i+1;
7    }
8    y=y*y;
9  }
```

For any given value of n, the loop in line 4 can be unrolled, and the adjoint of the resulting straight-line code can be built according to Adjoint Code Generation Rules 1–4. A "general-purpose" adjoint code needs to be valid for arbitrary n. The order in which the assignments are executed in the original program for any set of inputs (n and x in this case) needs to be reversed. While this information is rather easily extracted from the given example code, the solution to this problem may be less straightforward for larger programs. A generally valid algorithmic approach is required.

The simplest way to reverse the order of all executed assignments is to enumerate them in the augmented forward section followed by pushing their respective indices onto a *control flow stack*, for example, stack ⟨**int**⟩ control. All indices are retrieved in LIFO order in the reverse section and the corresponding adjoint statements are executed. For example, the assignments in lines 3, 5, 6, and 8 receive indices, 0, 1, 2, and 3, respectively, yielding the following adjoint code:

```
void a1_f(int n, double* x, double* a1_x,
                 double& y, double a1_y) {
  int i=0;
  // augmented forward sweep
  control.push(0); y=0;
  while (i<n) {
    control.push(1); y=y+x[i]*x[i];
    control.push(2); required_integer.push(i); i=i+1;
  }
  control.push(3); required_double.push(y); y=y*y;

  // store result
  result_double.push(y);

  // reverse sweep
  while (!control.empty()) {
    if (control.top()==0)
      a1_y=0;
    else if (control.top()==1)
      a1_x[i]+=2*x[i]*a1_y;
    else if (control.top()==2) {
      i=required_integer.top();
      required_integer.pop();
    }
    else if (control.top()==3) {
      y=required_double.top();
      required_double.pop();
      a1_y=2*y*a1_y;
    }
```

```
    control.pop();
  }

  // restore result
  y=result_double.top();
  result_double.pop();
}
```

The individual adjoint statements are constructed according to Adjoint Code Generation Rules 1–4 followed by some obvious code optimizations. For example, the adjoint of y=y+x[i]∗x[i] is constructed from

```
1 // incomplete SAC
2 v1=x[i]*x[i]
3 v2=y+v1;
4 // adjoint statements
5 a1_v2=a1_y; a1_y=0;
6 a1_v1=a1_v2; a1_y+=a1_v2;
7 a1_x[i]+=2*x[i]*a1_v1;
```

Neither v1 nor v2 is used by the adjoint statements, making lines 2 and 3 obsolete. Copy propagation in lines 5–7 yields a1_x[i]+=2∗x[i]∗a1_y. The driver calls a1_f once to compute the entire gradient.

```
void driver(int n, double* x, double* g) {
  double y;
  for (int i=0;i<n;i++) g[i]=0;
  a1_f(n,x,g,y,1);
}
```

The size of the control stack can be significantly reduced by enumerating basic blocks instead of individual assignments. Reversing the order of the assignments within a basic block is trivial. Consequently, the augmented forward section of our example code becomes

```
  control.push(0); y=0;
  while (i<n) {
    control.push(1); y=y+x[i]*x[i];
    required_integer.push(i); i=i+1;
  }
  control.push(2); required_double.push(y); y=y*y;
```

and its reverse section

```
  while (!control.empty()) {
    if (control.top()==0)
      a1_y=0;
    else if (control.top()==1) {
      i=required_integer.top();
      required_integer.pop();
      a1_x[i]+=2*x[i]*a1_y;
    }
    else if (control.top()==2) {
      y=required_double.top();
      required_double.pop();
```

```
      a1_y=2*y*a1_y;
    }
    control.pop();
  }
```

The major advantages of the basic block enumeration method are its relative simplicity and its applicability to arbitrary flow of control. Older legacy code in particular sometimes makes excessive use of **goto** statements, thus making it hard or even impossible to identify loops in the flow of control at compile time. Further improvements are possible for *reducible flow of control* [2], which allows for all loops to be detected at compile time, potentially followed by a syntactic modification of the source code to make such loops explicit. In this case, the number of iterations can be counted for each loop within the augmented forward section. The adjoint loop is constructed to perform the same number of executions of the adjoint loop body within the reverse section as illustrated below.

```
void a1_f(int n, double* x, double* a1_x,
                 double& y, double a1_y) {
  int i=0;
  // augmented forward sweep
  y=0;
  int loop_counter=0;
  while (i<n) {
    y=y+x[i]*x[i];
    required_integer.push(i); i=i+1;
    loop_counter++;
  }
  control.push(loop_counter);
  required_double.push(y); y=y*y;

  // store result
  result_double.push(y);

  // reverse sweep
  y=required_double.top();
  required_double.pop();
  a1_y=2*y*a1_y;
  loop_counter=0;
  while (loop_counter<control.top()) {
    i=required_integer.top();
    required_integer.pop();
    a1_x[i]+=2*x[i]*a1_y;
    loop_counter++;
  }
  control.pop();
  a1_y=0;

  // restore result
  y=result_double.top();
  result_double.pop();
}
```

The memory savings are dramatic if, as in this case, the loop body is a basic block. Instead of storing n copies of the basic block index (1 in our case) we need only store a single integer that represents the number of iterations actually performed. If the loop body is not a basic block, then the savings due to counting loop iterations are mostly insignificant. Moreover, special care must be taken when considering nontrivial control flow constructs involving nested loops and branches. Refer to the AD tool TAPENADE [52] for an implementation of control flow reversal by counting loops and enumerating branches.

Sometimes the semantics of certain syntactic structures can be exploited for the generation of optimized adjoint code. For example, the reversal of *simple* **for**-loops in C/C++ such as

```
for (int i=0;i<n;i++) ...
```

can be implemented by an inverse **for**-loop starting with the target index value n−1 and decrementing the counter i down to the start value 0:

```
for (int i=n-1;i>=0;i--) ....
```

This technique is illustrated in the following implementation of an adjoint for (1.2). Note that even though the loop index is required by the adjoint code of the loop body, instead of saving it after each loop iteration, its required values are generated explicitly by the reversed loop.

```
1 void a1_f(int n, double* x, double* a1_x,
2                  double& y, double a1_y) {
3   // augmented forward section
4   y=0;
5   for (int i=0;i<n;i++) y=y+x[i]*x[i];
6   required_double.push(y); y=y*y;
7
8   // store result
9   result_double.push(y);
10
11  // reverse section
12  y=required_double.top(); required_double.pop();
13  a1_y=2*y*a1_y;
14  for (int i=n-1;i>=0;i--)
15    a1_x[i]=a1_x[i]+2*x[i]*a1_y;
16
17  // restore result
18  y=result_double.top(); result_double.pop();
19 }
```

The right-hand side value of y in line 6 is required in line 13 for the evaluation of the partial derivative of y*y; it is stored on the required double data stack required_double. If a1_f is not required to return the correct function value y computed in line 6, then lines 6–9 in the augmented forward section and lines 12 and 17–18 of the reverse section can be removed from the adjoint code. Thus, for this simple example the additional memory requirement of the adjoint code can be reduced to zero when compared with the tangent-linear code.

Run time results for the different approaches to intraprocedural control flow reversal are reported in Table 2.3.

Table 2.3. *Run times for adjoint code (in seconds). n function evaluations are compared with n evaluations of the adjoint code. The convenience of dynamic memory management provided by the C++ standard library is paid for with a considerable increase of this factor, both in versions v.*1 *(enumerate basic blocks) and v.*2. *(count loop iterations). Version v.*3 *(explicit* ***for****-loop reversal) avoids stack accesses for the most part and takes less than* $\mathcal{R} = 3$ *times the run time of a function evaluation (see rightmost column), which is close to optimal. Missing compiler optimization yields an (often dramatic) increase in the observed run-time ratio.*

	g++ -O0			g++ -O3			$\mathcal{R}$
n	10^4	$2\cdot10^4$	$4\cdot10^4$	10^4	$2\cdot10^4$	$4\cdot10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
a1_f (v.1)	46.5	186.2	746.0	2.4	9.5	39.0	≈ 12.6
a1_f (v.2)	15.1	65.3	243.1	1.0	3.9	15.1	≈ 4.9
a1_f (v.3)	2.1	8.3	33.3	0.4	1.8	7.0	≈ 2.3

In general, the optimization of adjoint code combines elements from both classical compiler theory and numerical analysis. Determining whether some overwritten value is required or not may not be entirely straightforward for complex programs. Conservatively, one may decide to store all overwritten values on the corresponding stacks; the resulting memory requirement is likely to become infeasible. Various program analysis techniques have been developed to identify a minimal set of required values; see, for example, [37, 38]. Refer to [31] for an alternative approach to adjoint code generation based on the recomputation of required values from selected *checkpoints*. Similar ideas will be exploited in Section 2.3 for the optimization of interprocedural adjoint code.

Adjoint Code Generation Rule 6: Interprocedural Adjoint Code

Particular attention must be paid to the scope of variables.When a variable v leaves its scope, the corresponding memory can be reassigned by the compiler to other variables. The value of v, which may be required, is lost.

Conceptually, the generation of interprocedural adjoint code does not pose any further difficulties. For illustration, we split the computation of $x = \sin^n(x) \equiv \sin(\sin(\ldots(\sin(x))\ldots))$ implemented as

```
void f(int n, double& x) {
  int i=0;
  while (i<n) {
    x=sin(x);
    i=i+1;
  }
}
```

into three parts

$$x = \sin^{n_1}(x)$$
$$x = \sin^{n_2}(x)$$
$$x = \sin^{n_3}(x)$$

where $n_1+n_2+n_3 = n$ and we let $x = \sin^{n_2}(x)$ be computed by a subprogram g as follows.

```
void g(int n, double& x) {
  double l;
  for (int i=0;i<n;i++) {
    l=x;
    x=sin(l);
  }
}

void f(int n, double& x) {
  int n1,n2,n3;
  n1=n/3; n2=n/3; n3=n-n1-n2;
  for (int i=0;i<n1;i++) x=sin(x);
  g(n2,x);
  for (int i=0;i<n3;i++) x=sin(x);
}
```

The local variable l, which is actually obsolete, has been added to g for illustration of the impact of variable scopes on the adjoint code.

The augmented forward section of f records the overwritten required values of x. Basic block enumeration is not necessary, as the intraprocedural flow of control is described entirely by the two simple **for**-loops. The augmented forward section of g needs to be executed as part of the augmented forward section of f. Therefore, the augmented forward and reverse sections are separated in g and can be called individually by setting the first integer argument of a1_g equal to 1 (augmented forward section) or 2 (reverse section); see line 9 in the following code listing:

```
1  void a1_f(int n, double& x, double& a1_x) {
2    int n1,n2,n3;
3    // augmented forward section
4    n1=n/3; n2=n/3; n3=n-n1-n2;
5    for (int i=0;i<n1;i++) {
6      required_double.push(x);
7      x=sin(x);
8    }
9    a1_g(1,n2,x,a1_x);
10   for (int i=0;i<n3;i++) {
11     required_double.push(x);
12     x=sin(x);
13   }
```

The adjoint of the last loop is executed first within the reverse section (lines 2–5 of the following code listing). It is followed in line 6 by a call of the reverse section of a1_g. Finally, in lines 7–10, the adjoint of the originally first loop is executed.

```
1    // reverse section
2    for (int i=n3-1;i>=0;i--) {
3      x=required_double.top(); required_double.pop();
4      a1_x=cos(x)*a1_x;
5    }
6    a1_g(2,n2,x,a1_x);
7    for (int i=n1-1;i>=0;i--) {
```

```
 8       x=required_double.top(); required_double.pop();
 9       a1_x=cos(x)*a1_x;
10     }
11   }
```

The following adjoint version of g separates the augmented forward and reverse sections (lines 5–11 and lines 13–19, respectively). An integer parameter mode is used to choose between them. The value of l overwritten in line 8 is required in line 16 by the partial derivative of sin(l); l is stored in line 7 and restored in line 17.

```
 1 void a1_g(int mode, int n, double& x, double& a1_x) {
 2   double l=0, a1_l=0;
 3   int i=0;
 4   if (mode==1) {
 5     // augmented forward section
 6     for (int i=0;i<n;i++) {
 7       required_double.push(l);
 8       l=x;
 9       x=sin(l);
10     }
11     required_double.push(l);
12   } else if (mode==2) {
13     // reverse section
14     l=required_double.top(); required_double.pop();
15     for (int i=n-1;i>=0;i--) {
16       a1_l=a1_l+cos(l)*a1_x; a1_x=0;
17       l=required_double.top(); required_double.pop();
18       a1_x=a1_x+a1_l; a1_l=0;
19     }
20   }
21 }
```

The storage of the value of l in line 11 and the subsequent recovery in line 14 are necessary because l leaves its scope after the execution of the augmented forward section. The value of l required to compute the correct local partial derivative of the last execution of x=sin(l) in line 9 would otherwise be lost.

2.2.2 Adjoint Code by Overloading

The favored approach to a run time version of the adjoint model is to build a *tape* (an augmented representation of the DAG) by overloading, followed by an interpretative reverse propagation of adjoints through the tape. In our case the tape is a statically allocated array of tape entries addressed by their position in the array. Each tape entry contains a code for the associated operation (oc), addresses of the operation's first and optional second arguments (arg1 and arg2), and two floating-point variables holding the current value (v) and the adjoint (a), respectively. The constructor marks the operation code and both arguments as undefined and it initializes both the value and the adjoint to zero.

```
class dco_a1s_tape_entry {
  public:
```

```
    int oc, arg1, arg2;
    double v,a;
    dco_a1s_tape_entry() :
      oc(DCO_A1S_UNDEF), arg1(DCO_A1S_UNDEF),
      arg2(DCO_A1S_UNDEF), v(0), a(0)
    {};
};
```

As in forward mode, an augmented data type is defined to replace the type of every active floating-point variable. The corresponding **class** dco_a1s_type (dco's adjoint 1st-order scalar type) contains the virtual address va (position in tape) of the current variable in addition to its value v.

```
class dco_a1s_type {
  public:
    int va;
    double v;
    dco_a1s_type() : va(DCO_A1S_UNDEF), v(0) {};
    dco_a1s_type(const double&);
    dco_a1s_type& operator=(const dco_a1s_type&);
};
```

Special constructors and a custom assignment operator are required. The latter either handles a self-assignment or generates a new tape entry with corresponding operation code and with copies of the right-hand side's value and virtual address. A global virtual address counter dco_a1s_vac is used to populate the tape.

```
dco_a1s_type& dco_a1s_type::operator=(const dco_a1s_type& x) {
  if (this==&x) return *this;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ASG;
  dco_a1s_tape[dco_a1s_vac].v=v=x.v;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  va=dco_a1s_vac++;
  return *this;
}
```

Passive values and constants are activated by a special constructor:

```
dco_a1s_type::dco_a1s_type(const double& x): v(x) {
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_CONST;
  dco_a1s_tape[dco_a1s_vac].v=x;
  va=dco_a1s_vac++;
};
```

All arithmetic operators and intrinsic functions make similar recordings on the tape, for example,

```
dco_a1s_type operator*(const dco_a1s_type& x1,
                       const dco_a1s_type& x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_MUL;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
```

```
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}
```

and

```
dco_a1s_type sin(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SIN;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}   .
```

The operation codes are implemented as macros (DCO_A1S_ASG, DCO_A1S_MUL, ...) to be replaced with some unique number by the C preprocessor.

The tape is constructed during a single execution of the overloaded original code; this is followed by an interpretation step for propagating adjoints through the tape in reverse order.

```
void dco_a1s_interpret_tape () {
  for (int i=dco_a1s_vac;i>=0;i--) {
    switch (dco_a1s_tape[i].oc) {
      case DCO_A1S_ASG : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_MUL : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          dco_a1s_tape[dco_a1s_tape[i].arg2].v*dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=
          dco_a1s_tape[dco_a1s_tape[i].arg1].v*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SIN : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=
          cos(dco_a1s_tape[dco_a1s_tape[i].arg1].v)
            *dco_a1s_tape[i].a;
        break;
      }
      ...
    }
  }
}
```

The driver program in Listing 2.2 uses the implementation of **class** dco_a1s_type in connection with a tape of size DCO_A1S_TAPE_SIZE (to be replaced with an integer value by the C preprocessor). The tape is allocated statically in `dco_a1s_type.cpp` and is later linked to the object code of the driver program. The latter computes the gradient of the

Listing 2.2. *Driver for adjoint code by overloading.*

```
1 #include <iostream>
2 #include "dco_a1s_type.hpp"
3 using namespace std;
4
5 const int n=4;
6
7 extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];
8
9 void f(dco_a1s_type *x, dco_a1s_type &y) {
10   y=0;
11   for (int i=0;i<n;i++) y=y+x[i]*x[i];
12   y=y*y;
13 }
14
15 int main() {
16
17   dco_a1s_type x[n], y;
18   for (int i=0;i<n;i++) {
19     for (int j=0;j<n;j++) x[j]=1
20     f(x,y);
21     dco_a1s_tape[y.va].a=1;
22     dco_a1s_interpret_tape();
23     cout << i << "\t" << dco_a1s_tape[x[i].va].a << endl;
24     dco_a1s_reset_tape();
25   }
26   return 0;
27 }
```

given implementation of (1.2), that is

$$f(\mathbf{x}) = \left(\sum_{i=0}^{n-1} x_i^2\right)^2,$$

for n = 4 at the point $x_i = 1$ for $i = 0,\ldots,3$. Running the augmented function

```
void f(dco_a1s_type *x, dco_a1s_type &y)
```

followed by the tape interpretation yields the two tapes in Figure 2.9. Arguments are referenced by their virtual address within the tape. For example, tape entry 11 represents the sum (oc=2) of the two arguments represented by tape entries 9 and 10. The tape is structurally equivalent to the DAG. The propagation of adjoints is preceded by the initialization of the adjoint of the tape entry that corresponds to the dependent variable y (tape entry 23). The desired gradient is accumulated in the adjoint components of the four tape entries 1, 3, 5, and 7.

Tape entries 0–7 correspond to the initialization of the x[j] in line 19 of Listing 2.2. The initialization of y inside of f (line 10) yields tape entries 8 and 9. The loop in line 11 produces the following twelve (four triplets) entries 10–21. Squaring y in line 12 adds the last two tape entries 22 and 23.

Tape: Interpreted Tape:

```
 0: [ 0, -1, -1,  1.0, 0.0 ][ 0, -1, -1,  1.0, 16.0 ]
 1: [ 1,  0, -1,  1.0, 0.0 ][ 1,  0, -1,  1.0, 16.0 ]
 2: [ 0, -1, -1,  1.0, 0.0 ][ 0, -1, -1,  1.0, 16.0 ]
 3: [ 1,  2, -1,  1.0, 0.0 ][ 1,  2, -1,  1.0, 16.0 ]
 4: [ 0, -1, -1,  1.0, 0.0 ][ 0, -1, -1,  1.0, 16.0 ]
 5: [ 1,  4, -1,  1.0, 0.0 ][ 1,  4, -1,  1.0, 16.0 ]
 6: [ 0, -1, -1,  1.0, 0.0 ][ 0, -1, -1,  1.0, 16.0 ]
 7: [ 1,  6, -1,  1.0, 0.0 ][ 1,  6, -1,  1.0, 16.0 ]
 8: [ 0, -1, -1,  0.0, 0.0 ][ 0, -1, -1,  0.0,  8.0 ]
 9: [ 1,  8, -1,  0.0, 0.0 ][ 1,  8, -1,  0.0,  8.0 ]
10: [ 4,  1,  1,  1.0, 0.0 ][ 4,  1,  1,  1.0,  8.0 ]
11: [ 2,  9, 10,  1.0, 0.0 ][ 2,  9, 10,  1.0,  8.0 ]
12: [ 1, 11, -1,  1.0, 0.0 ][ 1, 11, -1,  1.0,  8.0 ]
13: [ 4,  3,  3,  1.0, 0.0 ][ 4,  3,  3,  1.0,  8.0 ]
14: [ 2, 12, 13,  2.0, 0.0 ][ 2, 12, 13,  2.0,  8.0 ]
15: [ 1, 14, -1,  2.0, 0.0 ][ 1, 14, -1,  2.0,  8.0 ]
16: [ 4,  5,  5,  1.0, 0.0 ][ 4,  5,  5,  1.0,  8.0 ]
17: [ 2, 15, 16,  3.0, 0.0 ][ 2, 15, 16,  3.0,  8.0 ]
18: [ 1, 17, -1,  3.0, 0.0 ][ 1, 17, -1,  3.0,  8.0 ]
19: [ 4,  7,  7,  1.0, 0.0 ][ 4,  7,  7,  1.0,  8.0 ]
20: [ 2, 18, 19,  4.0, 0.0 ][ 2, 18, 19,  4.0,  8.0 ]
21: [ 1, 20, -1,  4.0, 0.0 ][ 1, 20, -1,  4.0,  8.0 ]
22: [ 4, 21, 21, 16.0, 0.0 ][ 4, 21, 21, 16.0,  1.0 ]
23: [ 1, 22, -1, 16.0, 0.0 ][ 1, 22, -1, 16.0,  1.0 ]
```

(a) (b)

Figure 2.9. *dco_a1s_tape for the computation of the gradient of* (1.2) *for* $n = 4$ *at the point* $x_i = 1$ *for* $i = 0,\ldots,3$. *The five columns show for each tape entry with virtual addresses from* `0` *to* `23`, *the operation code, the virtual addresses of the (up to two) arguments, the function value, and the adjoint value, where* `-1` ≡ DCO_A1S_UNDEF *in the third and fourth columns and with operation codes* `0` ≡ DCO_A1S_CONST, `1` ≡ DCO_A1S_ASG, `2` ≡ DCO_A1S_ADD, *and* `4` ≡ DCO_A1S_MUL.

The tape interpreter implements (2.9) without modification. Starting from tape entry 23, the adjoint value 1 of the dependent variable y is propagated to the single argument of the underlying assignment. The adjoint of tape entry 22 is set to 1 as the local partial derivative of an assignment is equal to 1. Tape entry 22 represents the multiplication y=y∗y in line 12 of Listing 2.2, where the value of y on the right-hand side of the assignment is represented by tape entry 21. The value of the local partial derivative (2∗y=2*4=8) is multiplied with the adjoint of tape entry 22, followed by incrementing the adjoint of tape entry 21, whose initial value is equal to 0. This process continues until all tape entries have been visited. The gradient can be retrieved from tape entries 1, 3, 5, and 7. If none of the independent variables is overwritten, then their va components contain the correct virtual addresses after calling the overloaded version of f. This is the case in the given example. Hence, lines 24–27 deliver the correct gradient in Listing 2.2. Otherwise, the virtual addresses of the independent variables need to be stored in order to ensure a correct retrieval of the gradient.

Table 2.4. *Run times for adjoint code by overloading (in seconds). n function evaluations are compared with n evaluations of the adjoint code including the generation and interpretation of the tape. We observe a difference of a factor of at least* 16 *when comparing the run time of the adjoint code with that of an original function evaluation in the rightmost column. This factor increases with growing values of n. Compiler optimization has almost no effect on the quality of the adjoint code. With increasing tape size, the run time is dominated by the memory accesses. The observed factor rises quickly to* 100 *and more. Version 1.0 of* `dco` *keeps the factor below* 20 *by exploiting advanced techniques whose discussion is beyond the scope of this introduction.*

	g++ -O0			g++ -O3			
n	10^4	$2\cdot10^4$	$4\cdot10^4$	10^4	$2\cdot10^4$	$4\cdot10^4$	/T(f)
f	0.9	3.6	13.7	0.2	0.8	3.1	1
a1_f	23.8	110.3	551.9	12.7	73.5	478.2	> 16

Listings of the full source code that implements adjoint mode AD by overloading can be found in Section A.2. If both **class** dco_a1s_type and the tape are implemented in the files `dco_a1s_type.hpp` and `dco_a1s_type.cpp`, and if the driver program is stored as `main.cpp`, then the build process is similar to that in Section 2.1.2. Run time measurements are reported in Table 2.4.

Tape-based reverse mode AD can be implemented in vector mode by redefining tape entries as follows:

```
class dco_a1v_tape_entry {
  public:
    int oc, arg1, arg2;
    double v,*a;
  ...
};
```

The overloaded operators and functions remain unchanged. The tape interpreter needs to be altered to enable the propagation of vectors of adjoints. Remarks similar to those made in Section 2.1.2 apply.

Several implementations of reverse mode AD by overloading have been proposed over the past decades. Popular representatives for C++ include ADOL-C [34], cppAD [7], and FADBAD [9]. While the fundamental concepts are similar to what we have described here, the actual implementations vary in terms of the functionality and efficiency of the resulting code. Version 0.9 of `dco` is not meant to compete with the established tools. Later versions of `dco` provide a wider range of functionalities (checkpointing, parallelism, hybrid tangent-linear and adjoint modes) while yielding more robust and efficient derivative code.

2.2.3 Seeding and Harvesting Adjoint Code

When applying compression techniques in adjoint mode, we aim to compute

$$B_a = A^T \cdot S_a, \tag{2.10}$$

where $A \equiv \nabla F(\mathbf{x})$ and $S_a \in \{0,1\}^{m\times l_a}$ such that $\forall a_{i,j} \neq 0\ \exists b^a_{l,i} \in B_a : a_{i,j} = b^a_{l,i}$. Each nonzero element $a_{i,j}$ in the Jacobian must be present in B_a. Similar to the tangent-linear

case, the matrix S_a is referred to as the seed matrix and B_a as the compressed transposed Jacobian. The number of columns in S_a is denoted by l_a. Harvesting solves (2.10) by substitution. Refer to [30] for details on the combinatorial problem that is to minimize l_a by graph coloring algorithms.

Example 2.15 An adjoint version of the implementation in Example 2.10,

```
void a1_f(int n,double *x,double *a1_x,
          int m,double *y,double *a1_y),
```

is generated by reverse mode AD. A driver for computing the compressed transposed Jacobian B_a uses the row-partition $\mathcal{I}_1 = \{0,2\}$ and $\mathcal{I}_2 = \{1\}$ as follows:

```
int main() {
  double x[4]=..., a1_x[4];
  double y[3];
  {
    double a1_y[3]={1,0,1};
    a1_f(4,x,a1_x,3,y,a1_y); // rows 0 and 2
  }
  ...
  {
    double a1_y[3]={0,1,0};
    a1_f(4,x,a1_x,3,y,a1_y); // row 1
  }
  ...
}
```

The unknown nonzero entries $x_{i,j}$ of the transposed Jacobian are obtained by substitution from

$$\begin{pmatrix} x_{0,0} & 0 & 0 \\ 0 & 0 & x_{2,1} \\ 0 & x_{1,2} & x_{2,2} \\ x_{0,3} & x_{1,3} & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} a_{0,0} & 0 \\ a_{2,1} & 0 \\ a_{2,2} & a_{1,2} \\ a_{0,3} & a_{1,3} \end{pmatrix}. \qquad \blacksquare$$

Combinations of tangent-linear and adjoint compression may give better compression rates as described, for example, in [36]. Arrow-shaped matrices are prime examples for this type of bidirectional seeding and harvesting.

2.3 Call Tree Reversal

Consider the interprocedural adjoint code used to illustrate Adjoint Code Generation Rule 6 in Section 2.2.1. For $n = 10$, the size of the required double data stack hits its maximum of 11 at the end of the augmented forward section. Suppose that the available memory allows the storage of only 9 double precision floating-point values.[7] It would follow that this adjoint code cannot be run on the given computer.

Adjoint Code Generation Rule 7: Subroutine Argument Checkpointing

Buying more memory may be an option. However, we prefer an algorithmic solution that will allow us to generate suitable adjoint code for arbitrary available hardware. We focus

[7] We leave it to the reader to multiply this number by 10^k in order to get to a more realistic number.

on *subroutine argument checkpointing* in order to reduce the overall memory requirement of an interprocedural adjoint code at the expense of additional floating-point operations. Data required for the propagation of adjoints through called subroutines (a1_g) is generated within the reverse section of the caller (a1_f) rather than in its augmented forward section. An argument checkpoint is stored so that we are able to run the adjoint callee (its augmented forward section immediately followed by its reverse section) "out of context," that is, independent of the enclosing data flow.

The adjoint version a1_g of g provides three modes: The augmented forward section of a1_g no longer needs to be separated (also referred to as *split* in [36]) from its reverse section. Adjoint code generated according to Adjoint Code Generation Rules 1–5 is executed for mode==1. If mode==3, then an argument checkpoint is stored. It is recovered if mode==4, that is, the values stored for mode==3 are copied back into the input variables of g. The case mode==2 is skipped to ensure consistency with split mode.

```
void a1_g(int mode, int n, double& x, double& a1_x) {
  double l=0, a1_l=0;
  int i=0;
  if (mode==1) {
    // augmented forward section
    for (int i=0;i<n;i++) {
      required_double.push(l);
      l=x;
      x=sin(l);
    }
    // reverse section
    for (int i=n-1;i>=0;i--) {
      a1_l=a1_l+cos(l)*a1_x; a1_x=0;
      l=required_double.top(); required_double.pop();
      a1_x=a1_x+a1_l; a1_l=0;
    }
  } else if (mode==3) {
    arguments_double.push(x);
  } else if (mode==4) {
    x=arguments_double.top(); arguments_double.pop();
  }
}
```

There is no need to store the outputs at the end of the augmented forward section, as their values are *dead* within the reverse section of the calling routine a1_f. Liveness analysis [2] eliminates the corresponding statements from the adjoint code for g.

The adjoint code for f calls g and its adjoint a1_g as follows:

```
1 void a1_f(int n, double& x, double& a1_x) {
2   int n1,n2,n3;
3   // augmented forward section
4   n1=n/3; n2=n/3; n3=n-n1-n2;
5   for (int i=0;i<n1;i++) {
6     required_double.push(x);
7     x=sin(x);
```

```
 8   }
 9   // store argument checkpoint
10   a1_g(3,n2,x,a1_x);
11   g(n2,x);
12   for (int i=0;i<n3;i++) {
13     required_double.push(x);
14     x=sin(x);
15   }
16
17   // store results
18   results_double.push(x);
19
20   // reverse section
21   for (int i=n3-1;i>=0;i--) {
22     x=required_double.top(); required_double.pop();
23     a1_x=cos(x)*a1_x;
24   }
25   // restore argument checkpoint
26   a1_g(4,n2,x,a1_x);
27   a1_g(1,n2,x,a1_x);
28   for (int i=n1-1;i>=0;i--) {
29     x=required_double.top(); required_double.pop();
30     a1_x=cos(x)*a1_x;
31   }
32
33   // restore results
34   x=results_double.top(); results_double.pop();
35 }
```

The original version of g is called in line 11 as part of the augmented forward section of f. No required data is recorded. No additional memory is required. Instead, the value of x is stored as an argument checkpoint in line 10. Note that x is the only input of g whose value is overwritten by the subsequent statements in g or f. The value of the second input n2 remains unchanged throughout the entire program and thus does not need to be checkpointed. Once the propagation of adjoints through f reaches the point where adjoints need to be propagated through g, the argument checkpoint is restored (line 26). Subsequent recording of all required data within the augmented forward section of g is followed by the propagation of the adjoints through its reverse section. The results enter the remainder of the reverse section of f (lines 28–31). Result checkpointing is taken care of in lines 18 and 34 if required.

Note that the size of the required double data stack never exceeds 7. The additional memory requirement of the adjoint code is increased to 8 by the argument checkpoint of g. The first part of the reverse section of f (lines 21–24) decreases the stack size to 3. Subsequent execution of the augmented forward section of g lets it grow up to 6 again before all of the remaining entries are recovered. The reduced memory requirement comes at the expense of a single evaluation of g. Thus checkpointing enables the computation of adjoints within the given memory constraints at an additional computational cost.

2.3.1 Call Tree Reversal Modes

The general DATA FLOW REVERSAL (also DAG REVERSAL) problem concerns the selection of appropriate intermediate values as checkpoints for a given upper bound on the available *additional* memory, that is, the memory available on top of the duplicated data segment of the original program. This problem is NP-complete [49]. It is therefore unlikely that an efficient algorithm (with run time polynomial in the size of the DAG) for its deterministic solution can be formulated.

Our approach to the generation of interprocedural adjoint code suggests a focus on subroutine arguments as potential checkpoints. The associated CALL TREE REVERSAL (CTR) problem is a special case of DAG REVERSAL and is also NP-complete [48]. (Approximate) solutions for given instances of CTR turn out to be easier to integrate into adjoint versions of the corresponding code.

For a call of a subroutine g inside of another subroutine f represented by the call tree

```
|_ f
   |_ g
```

we distinguish the following two call reversal modes.

Split Call Reversal: The *split reversal* of the call of g inside of f is defined as

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|_ a1_f(ADJOIN)
        |_ a1_g(ADJOIN)
```

Subroutine calls are denoted by |_. The augmented forward section (RECORD) generates a recording of all data that is required by the reverse section and is potentially lost due to overwriting / deallocation. Adjoints are propagated by the reverse section (ADJOIN). The order of execution in such *reversal trees* is top-down.

Joint Call Reversal: The *joint reversal* of the call of g inside of f is defined as

```
|_ a1_f(RECORD)
|       |_ a1_g(STORE_INPUTS)
|       |_ g
|_ a1_f(ADJOIN)
        |_ a1_g(RESTORE_INPUTS)
        |_ a1_g(RECORD)
        |_ a1_g(ADJOIN)
```

Rather than recording the data that is required by the reverse section of a1_g, an argument checkpoint is stored (STORE_INPUT) and the original subroutine g is executed. The checkpoint is restored within the reverse section of a1_f, followed by runs of the augmented forward (RECORD) and reverse (ADJOIN) sections of a1_g.

Refer to [36] as the original source of the terms *split* and *joint* reversal modes. Split mode refers to the augmented forward and reverse sections being separated during the execution of the entire adjoint code. In joint mode, the reverse section follows the forward augmented section immediately.

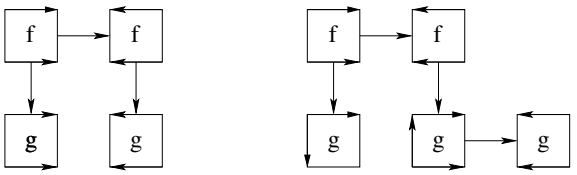


Figure 2.10. *Call reversal in spilt* (a) *and joint* (b) *modes; squares represent (sections of) subroutines. Execution of a subroutine is denoted by an overset arrow pointing to the right. A downward arrow indicates the storage of an argument checkpoint; its recovery is denoted by an upward arrow. Two rightward pointing arrows represent the augmented forward section. A reverse section is denoted by two leftward pointing arrows. The order of execution is depth-first and from left to right.*

Split and joint call reversal exhibit different memory requirements. If the size $\mathrm{MEM}(\mathbf{x}^{\mathrm{g}})$ of an argument checkpoint for g is considerably smaller than the amount of data to be recorded for the reversal of its data flow, then joint reversal yields a decreased memory requirement at the expense of an additional function evaluation.

For illustration, let f_0 and f_1 denote the two parts of f preceding and succeeding the call of g, respectively, as in the example discussed in the context of Adjoint Code Generation Rules 6 and 7. While the maximal memory requirement of split reversal is

$$\mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathrm{f}_1),$$

that of joint reversal amounts to

$$\mathrm{MEM}(\mathrm{f}_1) + \max(\mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) + \mathrm{MEM}(\mathrm{f}_1), \mathrm{MEM}(\mathrm{g})).$$

For example, if $\mathrm{MEM}(\mathrm{f}_0) = \mathrm{MEM}(\mathrm{f}_1) = \mathrm{MEM}(\mathrm{g}) = 10$ (memory units) and $\mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) = 1$, then the memory requirement of joint reversal (21) undercuts that of split reversal (30) by nearly a third. Graphical representations of split and joint call reversals are shown in Figure 2.10.

2.3.2 Call Tree Reversal Problem

The computational cost of a reversal scheme $R = R(T)$ for a call tree $T = (N, A)$ with nodes N and arcs A is defined by

1. the maximum amount of memory consumed in addition to the memory requirement of the original program, which is denoted by $\mathrm{MEM}(R)$;

2. the number of arithmetic operations performed in addition to those required for recording, denoted by $\mathrm{OPS}(R)$;

The choice between split and joint reversal is made independently for each arc in the call tree. Consequently, the call tree $T = (N, A)$ given as

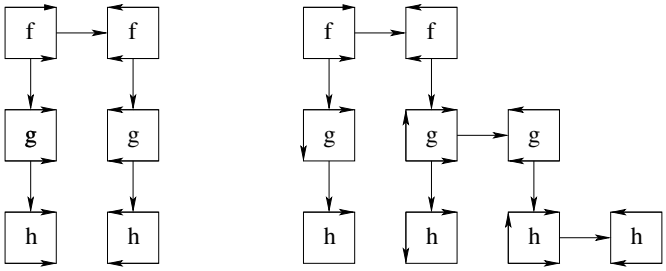


Figure 2.11. *Call tree reversal in global split* (a) *and global joint* (b) *modes.*

```
|_ f
    |_ g
        |_ h
```

yields a total of four possible *data flow reversal schemes* $R_j \subsetneq A \times \{0,1\}$, $j = 1,\ldots,4$. The reversal of a call of g inside of f in split (joint) mode is denoted as (f,g,0) [(f,g,1)]. A subroutine f is separated into $f_0,\ldots,f_k$ if it performs k subroutine calls. MEM(f_i) denotes the memory required to record f_i for $i = 0,\ldots,k$. The computational cost of running f_i is denoted by OPS(f_i). We set $\text{MEM(f)} = \sum_{i=0}^{k} \text{MEM}(f_i)$ and $\text{OPS(f)} = \sum_{i=0}^{k} \text{OPS}(f_i)$. The memory occupied by an input checkpoint of f is denoted by MEM($\mathbf{x}^{\text{f}}$). Consequently, we have the choice between the following four CTR schemes:

- $R_1 = \{(\text{f},\text{g},0),(\text{g},\text{h},0)\}$ (**global split**)

  ```
  |_ a1_f(RECORD)
  |       |_ a1_g(RECORD)
  |              |_ a1_h(RECORD)
  |_ a1_f(ADJOIN)
          |_ a1_g(ADJOIN)
                 |_ a1_h(ADJOIN)
  ```

  A graphical representation is shown in Figure 2.11 (a). Additional memory requirement and operations count are given by

  $$\begin{aligned}\text{MEM}(R_1) &= \text{MEM(f)} + \text{MEM(g)} + \text{MEM(h)},\\ \text{OPS}(R_1) &= \text{OPS(f)} + \text{OPS(g)} + \text{OPS(h)}.\end{aligned}$$

- $R_2 = \{(\text{f},\text{g},1),(\text{g},\text{h},0)\}$ (**joint over split mode**)

  ```
  |_ a1_f(RECORD)
  |       |_ a1_g(STORE_INPUTS)
  |       |_ g
  |            |_ h
  |_ a1_f(ADJOIN)
  ```

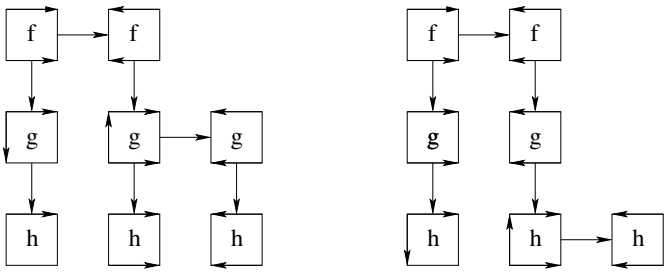


Figure 2.12. *CTR in joint over split* (a) *and split over joint* (b) *modes.*

```
|_ a1_g(RESTORE_INPUTS)
|_ a1_g(RECORD)
|       |_ a1_h(RECORD)
|_ a1_g(ADJOIN)
        |_ a1_h(ADJOIN)
```

A graphical representation is shown in Figure 2.12 (a). Additional memory requirement and operations count are given by

$$\mathrm{MEM}(R_2) = \max \left\{ \begin{matrix} \mathrm{MEM(f)} + \mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM(g)} + \mathrm{MEM(h)} \end{matrix} \right\},$$
$$\mathrm{OPS}(R_2) = \mathrm{OPS(f)} + 2 \cdot (\mathrm{OPS(g)} + \mathrm{OPS(h)}).$$

- $R_3 = \{(\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{h},1)\}$ (**split over joint mode**)

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|               |_ a1_h(STORE_INPUTS)
|               |_ h
|_ a1_f(ADJOIN)
        |_ a1_g(ADJOIN)
                |_ a1_h(RESTORE_INPUTS)
                |_ a1_h(RECORD)
                |_ a1_h(ADJOIN)
```

A graphical representation is shown in Figure 2.12 (b). Additional memory requirement and operations count are given by

$$\mathrm{MEM}(R_3) = \max \left\{ \begin{matrix} \mathrm{MEM(f)} + \mathrm{MEM(g)} + \mathrm{MEM}(\mathbf{x}^{\mathrm{h}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}_0) + \mathrm{MEM(h)} \end{matrix} \right\},$$
$$\mathrm{OPS}(R_3) = \mathrm{OPS(f)} + \mathrm{OPS(g)} + 2 \cdot \mathrm{OPS(h)}.$$

- $R_4 = \{(\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{h},1)\}$ (**global joint mode**)

```
|_ a1_f (RECORD)
|       |_ a1_g (STORE_INPUTS)
|       |_ g
|          |_ h
|_ a1_f (ADJOIN)
        |_ a1_g (RESTORE_INPUTS)
        |_ a1_g (RECORD)
        |       |_ a1_h (STORE_INPUTS)
        |       |_ h
        |_ a1_g (ADJOIN)
                |_ a1_h (RESTORE_INPUTS)
                |_ a1_h (RECORD)
                |_ a1_h (ADJOIN)
```

A graphical representation is shown in Figure 2.11 (b). Additional memory requirement and operations count are given by

$$\mathrm{MEM}(R_4) = \max \left\{ \begin{array}{l} \mathrm{MEM}(\mathrm{f}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{h}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}_0) + \mathrm{MEM}(\mathrm{h}) \end{array} \right\},$$
$$\mathrm{OPS}(R_4) = \mathrm{OPS}(\mathrm{f}) + 2 \cdot \mathrm{OPS}(\mathrm{g}) + 3 \cdot \mathrm{OPS}(\mathrm{h}).$$

Formally, the CTR problem aims to determine for a given call tree $T = (N, A)$ and an integer $K > 0$ a reversal scheme $R \subsetneq A \times \{0,1\}$ such that $\mathrm{OPS}(R) \to \min$ subject to $\mathrm{MEM}(R) \leq K$. Ongoing research investigates heuristics for determining a near-optimal reversal scheme in (preferably) linear time. A simple greedy *smallest-recording-first* heuristic starts with a global joint reversal and switches edge labels from 1 to 0 in increasing order of the callee's recording size. Ties are broken according to some enumeration of the nodes in T. The constraints of the CTR problem are guaranteed to be satisfied under the assumption that $\mathrm{MEM}(\mathbf{x}^{\mathrm{f}}) \leq \mathrm{MEM}(\mathrm{f})$ for all $f \in N$. In this case, global joint reversal yields the minimal memory requirement. Effective use of the larger available memory may allow for certain calls to be reversed in split rather than joint mode, as illustrated by the following example.

Example 2.16 Consider the call tree $T = (N, A)$ in Figure 2.13 (a). Nodes are annotated with the sizes of the respective input checkpoints (left) and the sizes of the recordings (below the nodes). For example, $\mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) = 5$ and $\mathrm{MEM}(\mathrm{g}_0) = \mathrm{MEM}(\mathrm{g}_1) = \mathrm{MEM}(\mathrm{g}_2) = 10$, and hence $\mathrm{MEM}(\mathrm{g}) = 30$. We assume $\mathrm{MEM}(p) = \nu \cdot \mathrm{OPS}(p)$ for any program code fragment p and for some $\nu \in \mathbb{R}$. This assumption turns out to be reasonable in most practical situations. For simplicity, we set $\nu = 1$ in this example.

There are $2^{|A|} = 8$ distinct reversal schemes, each with a potentially different computational cost. Global joint reversal $R_j = ((\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{s},1),(\mathrm{g},\mathrm{h},1))$ yields $\mathrm{MEM}(R_j) = 225$ and $\mathrm{OPS}(R_j) = 830$ which minimizes the overall memory requirement. Global split mode $R_s = ((\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{s},0),(\mathrm{g},\mathrm{h},0))$ minimizes the operation count ($\mathrm{OPS}(R_s) = 300$) at the expense of a maximum memory requirement of $\mathrm{MEM}(R_s) = 300$. A graphical illustration of global split and global joint CTR modes is given in Figure 2.13 (b) and Figure 2.14, respectively.

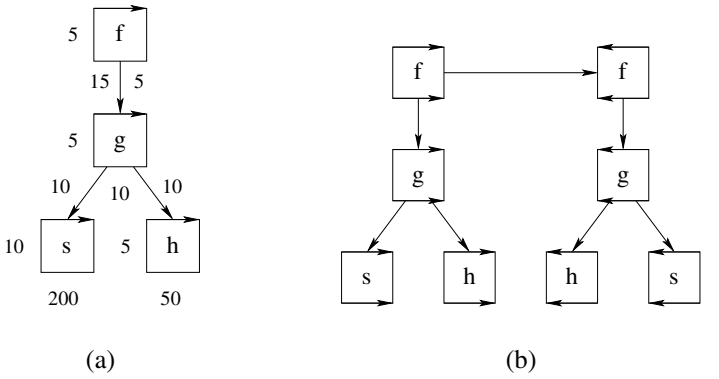


Figure 2.13. *Annotated call tree* (a) *and its global split reversal* (b).

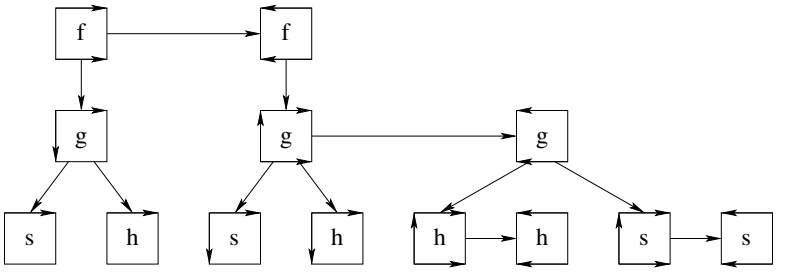


Figure 2.14. *Global joint reversal of the call tree in Figure* 2.13 (a).

Let the available memory be of size 250. Global split reversal becomes infeasible. Global joint reversal is an option, but can we do better? The given small example allows us to perform an exhaustive search for a solution of the CTR problem yielding the following six reversal schemes in addition to the global split and joint reversals discussed above:

- $R_1 = ((\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{s},1),(\mathrm{g},\mathrm{h},0))$ with $\mathrm{MEM}(R_1) = 225$ and $\mathrm{OPS}(R_1) = 780$;
- $R_2 = ((\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{s},0),(\mathrm{g},\mathrm{h},1))$ with $\mathrm{MEM}(R_2) = 285$ and $\mathrm{OPS}(R_2) = 630$;
- $R_3 = ((\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{s},0),(\mathrm{g},\mathrm{h},0))$ with $\mathrm{MEM}(R_3) = 295$ and $\mathrm{OPS}(R_3) = 580$;
- $R_4 = ((\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{s},1),(\mathrm{g},\mathrm{h},1))$ with $\mathrm{MEM}(R_4) = 225$ and $\mathrm{OPS}(R_4) = 550$;
- $R_5 = ((\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{s},1),(\mathrm{g},\mathrm{h},0))$ with $\mathrm{MEM}(R_5) = 225$ and $\mathrm{OPS}(R_5) = 500$;
- $R_6 = ((\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{s},0),(\mathrm{g},\mathrm{h},1))$ with $\mathrm{MEM}(R_6) = 285$ and $\mathrm{OPS}(R_6) = 350$;

R_1, R_4, and R_5 turn out to be feasible. R_5 yields the lowest operation count and represents the unique solution for the given instance of the CTR problem.

The greedy *smallest-recording-first* heuristic starts with the global joint reversal scheme and switches the reversal mode to split for the call of the subroutine with the smallest recording size; that is, $(\mathrm{f},\mathrm{g},1) \to (\mathrm{f},\mathrm{g},0)$. The operation count is decreased significantly

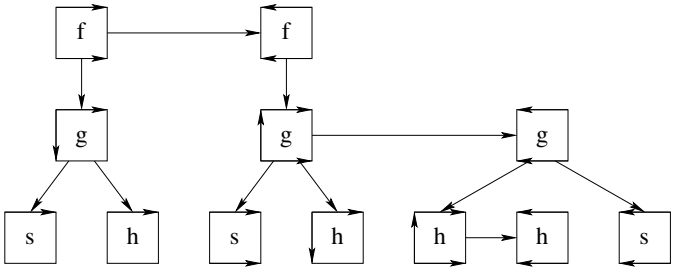


Figure 2.15. *Optimal reversal scheme for the call tree in Figure 2.13* (a) *for an available memory of size* 250*.*

to 550, whereas the memory requirement remains unchanged. Performing the next switch $(\mathrm{g},\mathrm{h},1) \rightarrow (\mathrm{g},\mathrm{h},0)$ reduces the operation count even further to 500 while preserving the memory requirement of 225. A last potential split $(\mathrm{g},\mathrm{s},1) \rightarrow (\mathrm{g},\mathrm{s},0)$ fails due to violation of the memory bound since $300 > 250$. The greedy *smallest-recording-first* heuristic succeeds in finding the optimal CTR scheme R_5 (shown in Figure 2.15) for the given instance of the CTR problem.

The greedy *largest-recording-first* heuristic starts with the global joint reversal scheme and attempts to switch the reversal mode to split for the call of the subroutine with the largest recording size, that is, $(\mathrm{g},\mathrm{s},1) \rightarrow (\mathrm{g},\mathrm{s},0)$, which yields the infeasible CTR scheme R_3. Rejection of the first switch is followed by $(\mathrm{g},\mathrm{h},1) \rightarrow (\mathrm{g},\mathrm{h},0)$, resulting in the feasible reversal scheme R_1. Finally, switching $(\mathrm{f},\mathrm{g},1) \rightarrow (\mathrm{f},\mathrm{g},0)$ gives the optimal result R_5 (shown in Figure 2.15).

Both greedy heuristics happen to lead to the same solution for the given example. Refer to the exercise in Section 2.4.5 for a call tree instance in which the two heuristics yield different results. ∎

A call tree is an image of the calling structure of a given program at run time. A (near-)optimal reversal scheme for a given call tree is of limited use if the calling structure of the program changes dynamically as a function of the inputs. In such cases, we need conservative solutions that guarantee feasible and reasonably efficient run time characteristics for all possible call trees on average. The automation of the (near-)optimal placement of checkpoints is the subject of ongoing research. AD will never become truly automatic unless robust software solutions for the DAG REVERSAL problem are developed.

2.4 Exercises

2.4.1 Code Differentiation Rules

1. Write tangent-linear code for Listing 2.3.

   Use the tangent-linear code to compute the Jacobian of the dependent outputs x and y with respect to the independent input x. Use central finite differences for verification.

Listing 2.3. *Disputable implementation of a function.*

```
void g(int n, double* x, double& y) {
  y=1.0;
  for (int i=0;i<n;i++)
    y*=x[i]*x[i];
}

void f(int n, double* x, double &y) {
  for (int i=0;i<n;i++) x[i]=sqrt(x[i]/x[(i+1)%n]);
  g(n,x,y);
  y=cos(y);
}
```

2. Write adjoint code for

```
void g(int n, double* x, double& y) {
  double l;
  int i=0;
  y=0;
  while (i<n) {
    l=x[i];
    y=y+x[i]*l;
    i=i+1;
  }
}
```

   and use it for the computation of the gradient of the dependent output y with respect to the independent input x. Apply backward finite differences for verification.

3. Write adjoint code (split mode) for the example code in Listing 2.3. Use the adjoint code to accumulate the gradient of the dependent output y with respect to the independent input x. Ensure that the correct function values are returned in addition to the gradient.

4. Write adjoint code (joint mode) for the example code in Listing 2.3. Use it to accumulate the gradient of the dependent output y with respect to the independent input x. Correct function values need not be returned.

5. Use the adjoint code developed in Exercises 3 and 4 to compute the gradient of the dependent output x[0] with respect to the independent input x. Optimize the adjoint code by eliminating obsolete (dead) statements.

2.4.2 Derivatives for Systems of Nonlinear Equations

Consider an implementation of the discrete residual $\mathbf{r} = F(\mathbf{y})$ for the SFI problem introduced in Example 1.2.

1. Implement the tangent-linear model $\mathbf{r}^{(1)} = \nabla F(\mathbf{y}) \cdot \mathbf{y}^{(1)}$ by writing a tangent-linear code by hand, and use it to accumulate $\nabla F(\mathbf{y})$ with machine accuracy. Compare the numerical results with those obtained by the finite difference approximation in Section 1.4.2.

2. Implement the adjoint model $\mathbf{y}_{(1)} = \mathbf{y}_{(1)} + \nabla F(\mathbf{y})^T \cdot \mathbf{r}_{(1)}$ by writing an adjoint code by hand, and use it to accumulate $\nabla F(\mathbf{y})$ with machine accuracy. Compare the numerical results with those obtained by the tangent-linear approach.

3. Use `dco` to implement the tangent-linear and adjoint models.

4. Use the Newton algorithm as well as a corresponding matrix-free implementation based on the CG method to solve the SFI problem. Compare the run times.

2.4.3 Derivatives for Nonlinear Programming

Consider the same implementation of the extended Rosenbrock function $y = f(\mathbf{x})$ as in Section 1.4.3.

1. Implement the tangent-linear model $y^{(1)} = \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ by writing a tangent-linear code by hand, and use it to accumulate $\nabla f(\mathbf{x})$ with machine accuracy. Compare the numerical results with those obtained by the finite difference approximation in Section 1.4.3.

2. Implement the adjoint model $\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \nabla f(\mathbf{x})^T \cdot y_{(1)}$ by writing an adjoint code by hand, and use it to accumulate $\nabla f(\mathbf{x})$ with machine accuracy. Compare the numerical results with those obtained by the tangent-linear approach.

3. Use `dco` to implement the tangent-linear and adjoint models.

4. Use the steepest descent algorithm with both first derivative models to minimize the extended Rosenbrock function. Compare the run times.

2.4.4 Derivatives for Numerical Libraries

1. Use the tangent-linear model with your favorite solver for systems of nonlinear equations to find a numerical solution of the SFI problem; repeat for further MINPACK-2 test problems.

2. Use the adjoint model with your favorite solver for nonlinear programming to minimize the extended Rosenbrock function; repeat for the other two test problems from Section 1.4.3.

2.4.5 Call Tree Reversal

1. Consider the following modification of the example code from Section 2.4.1:

```
void h(double& x) {
  x*=x;
}

void g(int n, double* x, double& y) {
  y=0;
  for (int i=0;i<n;i++) {
```

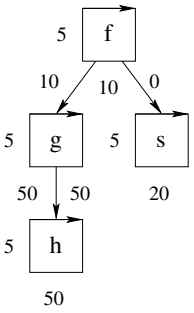


Figure 2.16. *Annotated call tree for Exercise* 2.4.5.

```
    h(x[i]); y*=x[i];
  }
}

void f(int n, double* x, double &y) {
  for (int i=0;i<n;i++) x[i]=sqrt(x[i]/x[(i+1)%n]);
  g(n,x,y);
  y=cos(y);
}
```

Write adjoint code that corresponds to the four reversal schemes

- $R_1 = \{(\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{h},0)\}$,
- $R_2 = \{(\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{h},0)\}$,
- $R_3 = \{(\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{h},1)\}$,
- $R_4 = \{(\mathrm{f},\mathrm{g},1),(\mathrm{g},\mathrm{h},1)\}$,

respectively. Apply the reversal mode of (g,h) to all n calls of h inside of g.

2. Consider the annotated call tree in Figure 2.16.

   (a) Derive all call tree reversal schemes. Compute their respective operation counts and memory requirements.

   (b) Compare the results of the greedy *smallest-* and *largest-recording-first* heuristics for an available memory of size 140, 150, and 160.

Chapter 3

Higher Derivative Code

Forward and reverse mode AD transform implementations of multivariate vector functions into tangent-linear and adjoint code. The reapplication of the same ideas yields higher derivative code. Second- and higher-order tangent-linear code is obtained by recursive application of forward mode AD. Sequences of applications of forward and reverse mode that involve at least one application of reverse mode result in higher-order adjoint code. An exponential number of combinations can be considered, for example, fourth-order adjoint code generated in forward-over-reverse-over-forward-over-reverse mode. We show why, for example, fourth-order adjoint code should rather be generated in forward-over-forward-over-forward-over-reverse mode. The second- and higher-order tangent-linear and adjoint code can be used to accumulate corresponding derivative tensors or projections thereof. Second derivatives, in particular, play an important role in nonlinear programming as outlined in Chapter 1.

The following presentation of second derivative code is based on the notation and terminology introduced in Section 3.1. Second-order tangent-linear code and its generation using source transformation and overloading techniques is discussed in Section 3.2. Second-order adjoint code is covered by Section 3.3. The generation of higher derivative code turns out to be reasonably straightforward as shown in Section 3.4. This chapter is supported by exercises in Section 3.5 and (remarks on) the corresponding solutions in Section C.3.

3.1 Notation and Terminology

Derivative code that is generated by AD can compute projections of derivative tensors of arbitrary order, for example, (transposed) Jacobian-vector products in the first-order case, Hessian-vector products in the scalar second-order case, and so forth. Sums of projections of tensors of various orders are returned by higher derivative code. AD users need to understand the effects of choosing certain directions for these projections (the *seeding* of derivative code) in order to be able to retrieve (*harvest*) the desired results. In this chapter, we propose a special kind of (derivative) tensor notation for this purpose. It has been found a useful tool in our ongoing attempt to explain the result of AD transformations to potential users.

The Jacobian is a function that maps an n-vector onto an $(m \times n)$-matrix, that is

$$F' \equiv \nabla F : \mathbb{R}^n \supseteq D \to \mathbb{R}^{m \times n}.$$

Differentiation of F' yields a 3-tensor as defined next.

Definition 3.1. *Let $D \subseteq \mathbb{R}^n$ be an open domain and let $F : D \to \mathbb{R}^m$ be twice continuously differentiable on D. Let $F' \equiv \nabla F$ denote the Jacobian of F. The* 3*-tensor*

$$\nabla^2 F(\mathbf{x}_0) \equiv \begin{pmatrix} \nabla F'_0(\mathbf{x}_0) & \cdots & \nabla F'_{n-1}(\mathbf{x}_0) \\ \nabla F'_n(\mathbf{x}_0) & \cdots & \nabla F'_{2\cdot n-1}(\mathbf{x}_0) \\ \vdots & \cdots & \vdots \\ \nabla F'_{(m-1)\cdot n}(\mathbf{x}_0) & \cdots & \nabla F'_{m\cdot n-1}(\mathbf{x}_0) \end{pmatrix} \in \mathbb{R}^{m \times n \times n}$$

is called the Hessian of F at point $\mathbf{x}_0$.

Example 3.2 The Hessian $\nabla^2 \mathbf{r}(\mathbf{y}, \nu) \in \mathbb{R}^{4\times4\times4}$ of the residual of the SFI problem from Example 1.2 becomes very sparse with

$$\nabla^2 r_{i,j,k} = \begin{cases} -h^2 \cdot \lambda \cdot e^{y_{1,1}} & \text{if } i = j = k = 0, \\ -h^2 \cdot \lambda \cdot e^{y_{1,2}} & \text{if } i = j = k = 1, \\ -h^2 \cdot \lambda \cdot e^{y_{2,1}} & \text{if } i = j = k = 2, \\ -h^2 \cdot \lambda \cdot e^{y_{2,2}} & \text{if } i = j = k = 3, \\ 0 & \text{otherwise.} \end{cases} \quad \blacksquare$$

kth derivative tensors are defined recursively as Jacobians of $(k-1)$th derivatives.

Example 3.3 The third derivative tensor $\nabla^3 \mathbf{r}(\mathbf{y}, \nu) \in \mathbb{R}^{4\times4\times4\times4}$ of the residual of the SFI problem becomes

$$\nabla^3 r_{i,j,k,l} = \begin{cases} -h^2 \cdot \lambda \cdot e^{y_{1,1}} & \text{if } i = j = k = l = 0, \\ -h^2 \cdot \lambda \cdot e^{y_{1,2}} & \text{if } i = j = k = l = 1, \\ -h^2 \cdot \lambda \cdot e^{y_{2,1}} & \text{if } i = j = k = l = 2, \\ -h^2 \cdot \lambda \cdot e^{y_{2,2}} & \text{if } i = j = k = l = 3, \\ 0 & \text{otherwise.} \end{cases} \quad \blacksquare$$

Conceptually, the computation of higher derivatives does not pose any exceptional difficulties. The notation is complicated by the need to work with higher-order tensors. Tensor notation is not necessarily required for first derivatives. Nevertheless we use it as an intuitive entry point into the following formalism.

Definition 3.4. *Let $A \equiv (a_{k,j})_{j=0,\ldots,n-1}^{k=0,\ldots,m-1} \in \mathbb{R}^{m\times n}$ be a* 2*-tensor (a matrix). A* first-order tangent-linear projection of A in direction $\mathbf{v} \in \mathbb{R}^n$ *is defined as the usual matrix vector product $A \cdot \mathbf{v}$. Alternatively, we use the inner product notation*

$$\mathbf{b} \equiv \langle A, \mathbf{v} \rangle \in \mathbb{R}^m,$$

where $\mathbf{b} = (b_k)_{k=0,\ldots,m-1}$ *and*

$$b_k = \langle a_{k,*}, \mathbf{v} \rangle \equiv \sum_{l=0}^{n-1} a_{k,l} \cdot v_l$$

for $k = 0, \ldots, m-1$. *The* k*th row of* A *is denoted by* $a_{k,*}$. *The expression* $\langle a_{k,*}, \mathbf{v} \rangle$ *denotes the usual scalar product of two vectors in* $\mathbb{R}^n$.

A first-order adjoint projection

$$\mathbf{c} \equiv \langle \mathbf{w}, A \rangle \in \mathbb{R}^n$$

of A in direction $\mathbf{w} \in \mathbb{R}^m$, *where* $\mathbf{c} = (c_j)_{j=0,\ldots,n-1}$, *is defined as*

$$c_j = \langle \mathbf{w}, a_{*,j} \rangle \equiv \sum_{l=0}^{m-1} w_l \cdot a_{l,j}$$

for $j = 0, \ldots, n-1$. *The* j*th column of* A *is denoted by* $a_{*,j}$.

A first-order adjoint projection of A in direction $\mathbf{v} \in \mathbb{R}^n$ *is defined as*

$$\langle \mathbf{v}, A \rangle \equiv \langle A, \mathbf{v} \rangle \in \mathbb{R}^m.$$

The definition of a first-order adjoint projection of a matrix in a direction in $\mathbb{R}^n$ as a tangent-linear projection is purely technical. Refer to the exercises in Section 3.5.3 and to their solutions in Section C.3.3 for corresponding uses in the context of higher-order adjoint code.

The tangent-linear model $\mathbf{y}^{(1)} = \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \in \mathbb{R}^m$ computes a first-order tangent-linear projection of the Jacobian matrix $\nabla F(\mathbf{x}) \in \mathbb{R}^{m \times n}$ in direction $\mathbf{x}^{(1)} \in \mathbb{R}^n$. A first-order adjoint projection $\mathbf{x}_{(1)} = \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \in \mathbb{R}^n$ of $\nabla F(\mathbf{x})$ in direction $\mathbf{y}_{(1)} \in \mathbb{R}^m$ is computed by the adjoint model.

As a relevant special case, we introduce tensor notation for second derivatives separately. If the function $\mathbf{y} = F(\mathbf{x})$ is twice continuously differentiable at any point of interest, then the second derivative tensor (the *Hessian*) is *partially symmetric* in the sense that

$$\frac{\partial^2 y_i}{\partial x_j \partial x_k} = \frac{\partial^2 y_i}{\partial x_k \partial x_j}$$

for $i = 0, \ldots, m-1$ and $j, k = 0, \ldots, n-1$. A corresponding property holds for third and higher derivative tensors as discussed in Section 3.4. Hence, the notation to be developed can be restricted to partially symmetric k-tensors for $k \geq 3$. In the following, the "partially" will be omitted for the sake of brevity.

Definition 3.5. *Consider a symmetric* 3*-tensor* $A \in \mathbb{R}^{m \times n \times n}$ *defined as*

$$A = (a_{k,j,i})_{i=0,\ldots,n-1}^{k=0,\ldots,m-1}$$

with $a_{k,i,j} = a_{k,j,i}$ *for* $i, j = 0, \ldots, n-1$.

A first-order tangent-linear projection

$$B \equiv \langle A, \mathbf{v} \rangle \in \mathbb{R}^{m \times n}$$

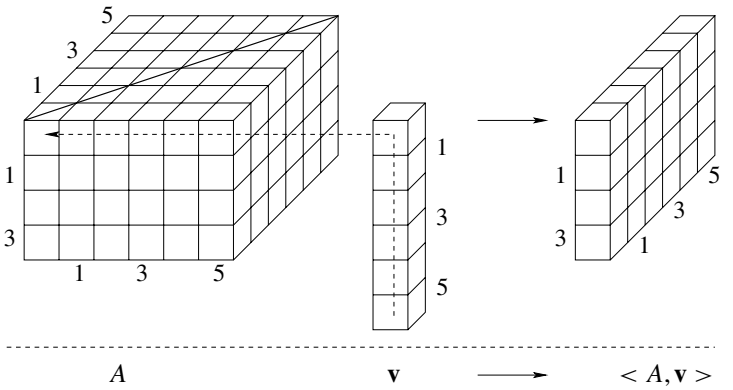


Figure 3.1. *First-order tangent-linear (equivalently, adjoint) projection of a symmetric* 3*-tensor* $A \in \mathbb{R}^{4\times 6\times 6}$ *in direction* $\mathbf{v} \in \mathbb{R}^6$: $\langle A, \mathbf{v}\rangle = \langle \mathbf{v}, A\rangle \in \mathbb{R}^{4\times 6}$. *The line of symmetry in A is shown as well as the direction of the projection and its result.*

of A in direction $\mathbf{v} \in \mathbb{R}^n$ *with* $B = (b_{k,j})_{j=0,\ldots,n-1}^{k=0,\ldots,m-1}$ *is defined as*

$$b_{k,j} = \langle a_{k,j,*}, \mathbf{v}\rangle \equiv \sum_{l=0}^{n-1} a_{k,j,l} \cdot v_l$$

for $k = 0,\ldots,m-1$ *and* $l = 0,\ldots,n-1$.

A first-order adjoint projection

$$C \equiv \langle \mathbf{w}, A\rangle \in \mathbb{R}^{n\times n}$$

of A in direction $\mathbf{w} \in \mathbb{R}^m$ *with* $C = (c_{j,i})_{i=0,\ldots,n-1}^{j=0,\ldots,n-1}$ *is defined as*

$$c_{j,i} = \langle \mathbf{w}, a_{*,j,i}\rangle \equiv \sum_{l=0}^{m-1} w_l \cdot a_{l,j,i}$$

for $i, j = 0,\ldots,n-1$.

For technical reasons, a first-order adjoint projection of A in direction $\mathbf{v} \in \mathbb{R}^n$ *is defined to be equivalent to the corresponding tangent-linear projection, that is,*

$$\langle \mathbf{v}, A\rangle \equiv \langle A, \mathbf{v}\rangle \in \mathbb{R}^{m\times n}.$$

Figures 3.1 and 3.2 provide a graphical illustration. Refer to the exercises in Section 3.5.3 and to their solutions in Section C.3.3 for applications of first-order adjoint projections of second derivative tensors in directions in $\mathbb{R}^n$ in the context of third- and higher-order adjoint models.

Lemma 3.6. *Let* $A \in \mathbb{R}^{m\times n\times n}$ *be a symmetric* 3*-tensor as defined in Definition* 3.5. *Then,* $C = \langle \mathbf{w}, A\rangle \in \mathbb{R}^{n\times n}$ *is symmetric for all* $\mathbf{w} \in \mathbb{R}^m$.

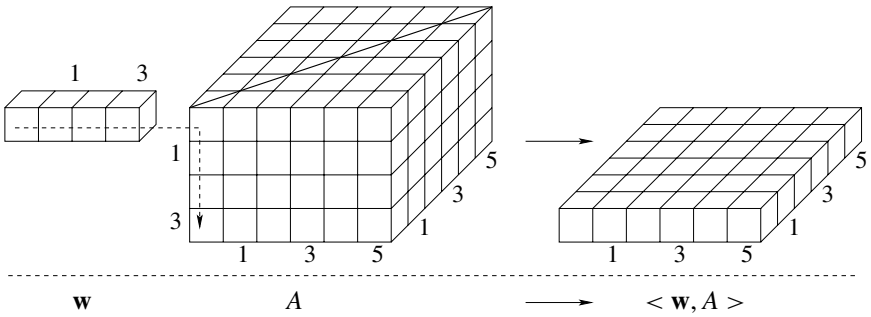


Figure 3.2. *First-order adjoint projection of a symmetric* 3*-tensor* $A \in \mathbb{R}^{4\times 6\times 6}$ *in direction* $\mathbf{w} \in \mathbb{R}^4 : \langle \mathbf{w}, A\rangle \in \mathbb{R}^{6\times 6}$.

Proof. This result follows immediately from Definition 3.5, as $C = (c_{j,i})_{i=0,\ldots,n-1}^{j=0,\ldots,n-1}$, where

$$c_{j,i} = \langle \mathbf{w}, a_{*,j,i}\rangle \equiv \sum_{l=0}^{m-1} w_l \cdot a_{l,j,i}.$$

Hence, $c_{j,i} = c_{i,j}$ because $a_{k,i,j} = a_{k,j,i}$ for $i, j = 0,\ldots,n-1$ and $k = 0,\ldots,m-1$. □

Definition 3.7. *A* second-order tangent-linear (equivalently, adjoint) projection $\langle A, \mathbf{u}, \mathbf{v}\rangle = \langle \mathbf{v}, \mathbf{u}, A\rangle \in \mathbb{R}^m$ *of a symmetric* 3*-tensor* A*, as defined in Definition* 3.5*, in directions* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ *is defined as a first-order tangent-linear (equivalently, adjoint) projection in direction* $\mathbf{v}$ *of the first-order tangent-linear (equivalently, adjoint) projection of* A *in direction* $\mathbf{u}$*, that is,*

$$\langle A, \mathbf{u}, \mathbf{v}\rangle \equiv \langle \langle A, \mathbf{u}\rangle, \mathbf{v}\rangle$$

(equivalently, $\langle \mathbf{v}, \mathbf{u}, A\rangle \equiv \langle \mathbf{v}, \langle \mathbf{u}, A\rangle\rangle$*).*

A second-order adjoint projection $\langle \mathbf{v}, \mathbf{w}, A\rangle \in \mathbb{R}^n$ *of* A *in directions* $\mathbf{w} \in \mathbb{R}^m$ *and* $\mathbf{v} \in \mathbb{R}^n$ *is defined as a first-order adjoint projection in direction* $\mathbf{v}$ *of the first-order adjoint projection of* A *in direction* $\mathbf{w}$*, that is*

$$\langle \mathbf{v}, \mathbf{w}, A\rangle \equiv \langle \mathbf{v}, \langle \mathbf{w}, A\rangle\rangle.$$

Figures 3.3 and 3.4 provide a graphical illustration.

The definition of first- and second-order projections of derivative tensors in directions $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^m$ captures all possible situations arising from different orders of applications of forward and reverse mode AD to tangent-linear and adjoint versions of implementations of multivariate vector functions. Higher-order projections of symmetric k-tensors will be introduced in Section 3.4 for use in derivative models of arbitrary order.

The following lemmas form the basis for upcoming results on the computational complexities of second and higher derivative models.

Lemma 3.8. *Let* $A \in \mathbb{R}^{m\times n\times n}$ *be a symmetric* 3*-tensor as defined in Definition* 3.5*, and let* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$*. Then,*

$$\langle A, \mathbf{u}, \mathbf{v}\rangle = \langle A, \mathbf{v}, \mathbf{u}\rangle.$$

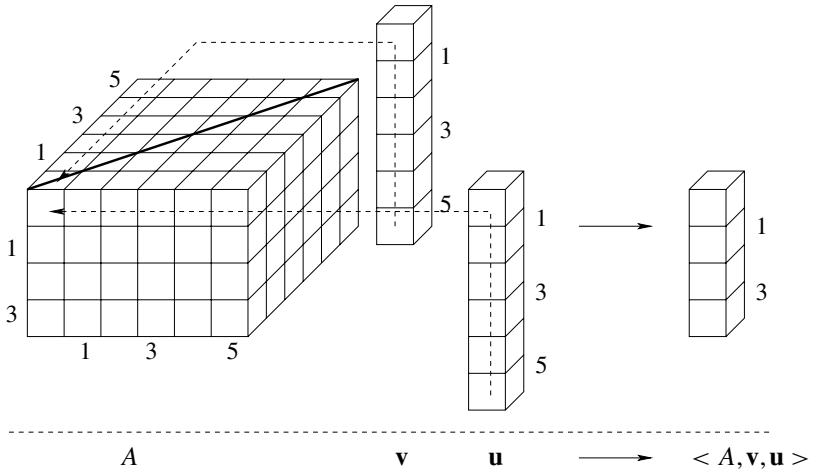


Figure 3.3. *Second-order tangent-linear (equivalently, adjoint) projection of a symmetric* 3*-tensor* $A \in \mathbb{R}^{4\times 6\times 6}$ *in directions* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^6 : \langle\langle A, \mathbf{u}\rangle, \mathbf{v}\rangle = \langle \mathbf{v}, \langle \mathbf{u}, A\rangle\rangle \in \mathbb{R}^4$.

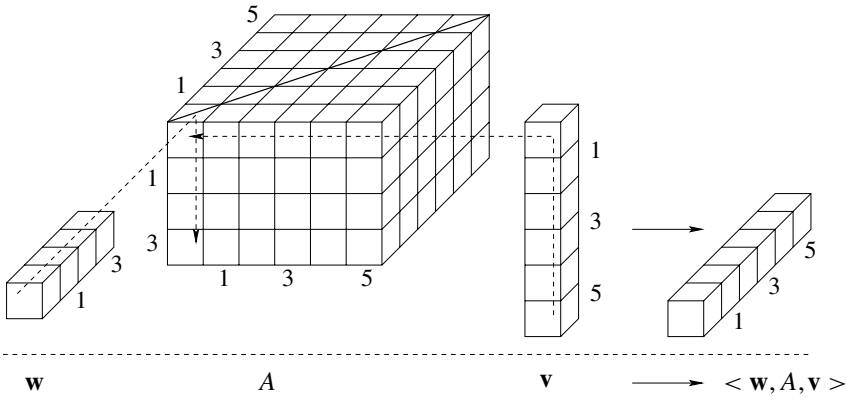


Figure 3.4. *Second-order adjoint projection of a symmetric* 3*-tensor* $A \in \mathbb{R}^{4\times 6\times 6}$ *in directions* $\mathbf{v} \in \mathbb{R}^6$ *and* $\mathbf{w} \in \mathbb{R}^4 : \langle \mathbf{w}, \langle A, \mathbf{v}\rangle\rangle = \langle\langle \mathbf{w}, A\rangle, \mathbf{v}\rangle = \langle \mathbf{v}, \langle \mathbf{w}, A\rangle\rangle \in \mathbb{R}^6$.

Proof. Let $B = \langle A, \mathbf{v}\rangle \in \mathbb{R}^{m\times n}$, $\mathbf{c} = \langle B, \mathbf{u}\rangle = \langle A, \mathbf{v}, \mathbf{u}\rangle \in \mathbb{R}^m$, and $D = \langle A, \mathbf{u}\rangle \in \mathbb{R}^{m\times n}$. Then, $b_{k,j} = \sum_{l=0}^{n-1} a_{k,j,l} \cdot v_l$ for $j = 0, \ldots, n-1$ and $k = 0, \ldots, m-1$. Moreover,

$$\begin{aligned} c_k &= \sum_{l=0}^{n-1} b_{k,l} \cdot u_l \\ &= \sum_{l=0}^{n-1} \sum_{p=0}^{n-1} a_{k,l,p} \cdot v_p \cdot u_l \qquad \text{(substitution)} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{p=0}^{n-1} \sum_{l=0}^{n-1} a_{k,l,p} \cdot v_p \cdot u_l && \text{(switch loops)} \\
&= \sum_{p=0}^{n-1} \sum_{l=0}^{n-1} a_{k,l,p} \cdot u_l \cdot v_p && \text{(commutativity)} \\
&= \sum_{p=0}^{n-1} d_{k,p} \cdot v_p
\end{aligned}
$$

for $k = 0, \ldots, m-1$, and hence $\mathbf{c} = \langle A, \mathbf{v}, \mathbf{u} \rangle = \langle B, \mathbf{u} \rangle = \langle D, \mathbf{v} \rangle = \langle A, \mathbf{u}, \mathbf{v} \rangle$. □

The next two lemmas yield three incarnations of second-order adjoint code to be discussed in Section 3.3.

Lemma 3.9. *Let $A \in \mathbb{R}^{m \times n \times n}$ be a symmetric 3-tensor as in Definition 3.5 and let $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^m$. Then,*

$$\langle \mathbf{v}, \mathbf{w}, A \rangle = \langle \mathbf{w}, A, \mathbf{v} \rangle.$$

Proof. The proof is similar to that of Lemma 3.8. Let $B = \langle \mathbf{w}, A \rangle \in \mathbb{R}^{n \times n}$, $\mathbf{c} = \langle \mathbf{v}, B \rangle = \langle \mathbf{v}, \mathbf{w}, A \rangle \in \mathbb{R}^n$, and $D = \langle A, \mathbf{v} \rangle \in \mathbb{R}^{m \times n}$. Then, $b_{j,i} = \sum_{l=0}^{m-1} w_l \cdot a_{l,j,i}$ for $i, j = 0, \ldots, n-1$. Moreover,

$$
\begin{aligned}
c_j &= \sum_{l=0}^{n-1} b_{j,l} \cdot v_l \\
&= \sum_{l=0}^{n-1} \sum_{p=0}^{m-1} a_{p,j,l} \cdot w_p \cdot v_l && \text{(substitution)} \\
&= \sum_{p=0}^{m-1} \sum_{l=0}^{n-1} a_{p,j,l} \cdot w_p \cdot v_l && \text{(switch loops)} \\
&= \sum_{p=0}^{m-1} \sum_{l=0}^{n-1} a_{p,j,l} \cdot v_l \cdot w_p && \text{(commutativity)} \\
&= \sum_{p=0}^{m-1} d_{p,j} \cdot w_p
\end{aligned}
$$

for $j = 0, \ldots, n-1$, and hence $\mathbf{c} = \langle \mathbf{v}, \mathbf{w}, A \rangle = \langle B, \mathbf{v} \rangle = \langle \mathbf{w}, D \rangle = \langle \mathbf{w}, A, \mathbf{v} \rangle$. □

Lemma 3.10. *Let $A \in \mathbb{R}^{m \times n \times n}$ be a symmetric 3-tensor as defined in Definition 3.5, and let $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^m$. Then,*

$$\langle \mathbf{w}, A, \mathbf{v} \rangle = \langle \langle \mathbf{w}, A \rangle, \mathbf{v} \rangle = \langle \mathbf{w}, \langle A, \mathbf{v} \rangle \rangle.$$

Proof. Let $B = \langle \mathbf{w}, A\rangle \in \mathbb{R}^{n\times n}$, $\mathbf{c} = \langle B, \mathbf{v}\rangle = \langle \mathbf{w}, A, \mathbf{v}\rangle \in \mathbb{R}^n$, and $D = \langle A, \mathbf{v}\rangle \in \mathbb{R}^{m\times n}$. Then, $b_{j,i} = \sum_{p=0}^{m-1} w_p \cdot a_{p,j,i}$ for $i, j = 0, \ldots, n-1$. Moreover,

$$\begin{aligned}
c_j &= \sum_{l=0}^{n-1} b_{j,l} \cdot v_l \\
&= \sum_{l=0}^{n-1}\sum_{p=0}^{m-1} w_p \cdot a_{p,j,l} \cdot v_l && \text{(substitution)} \\
&= \sum_{p=0}^{m-1}\sum_{l=0}^{n-1} w_p \cdot a_{p,j,l} \cdot v_l && \text{(switch loops)} \\
&= \sum_{p=0}^{m-1} w_p \cdot \sum_{l=0}^{n-1} a_{p,j,l} \cdot v_l && \text{(distributivity)} \\
&= \sum_{p=0}^{m-1} w_p \cdot d_{p,j}
\end{aligned}$$

for $j = 0, \ldots, n-1$, and hence

$$\mathbf{c} = \langle B, \mathbf{v}\rangle = \langle \langle \mathbf{w}, A\rangle, \mathbf{v}\rangle = \langle \mathbf{w}, D\rangle = \langle \mathbf{w}, \langle A, \mathbf{v}\rangle\rangle. \qquad \square$$

Application of forward and reverse mode AD to tangent-linear and adjoint code yields a total of four different kinds of second derivative code obtained in forward-over-forward, forward-over-reverse, reverse-over-forward, and reverse-over-reverse modes. Lemmas 3.9 and 3.10 imply that the last three modes yield equivalent projections of the second derivative tensor as discussed in further detail in Section 3.3.

3.2 Second-Order Tangent-Linear Code

Code for computing second derivatives of a function F is obtained by applying forward mode AD twice (forward-over-forward mode) to the given implementation of F.

Definition 3.11. *The Hessian $\nabla^2 F = \nabla^2 F(\mathbf{x}) \in \mathbb{R}^{m\times n\times n}$ of a multivariate vector function $\mathbf{y} = F(\mathbf{x})$, $F : \mathbb{R}^n \to \mathbb{R}^m$, induces a bilinear mapping $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$ defined by*

$$(\mathbf{u}, \mathbf{v}) \mapsto \langle \nabla^2 F, \mathbf{u}, \mathbf{v}\rangle.$$

The function $F^{(1,2)} : \mathbb{R}^{3\cdot n} \to \mathbb{R}^m$, that is defined as

$$F^{(1,2)}(\mathbf{x}, \mathbf{u}, \mathbf{v}) \equiv \langle \nabla^2 F(\mathbf{x}), \mathbf{u}, \mathbf{v}\rangle, \tag{3.1}$$

is referred to as the second-order tangent-linear model *of F.*

For $m = 1$, the mapping becomes

$$F^{(1,2)}(\mathbf{x}, \mathbf{u}, \mathbf{v}) \equiv \mathbf{u}^T \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{v},$$

where $\nabla^2 F(\mathbf{x}) \in \mathbb{R}^{n\times n}$ and $\cdot$ denotes the usual matrix multiplication.

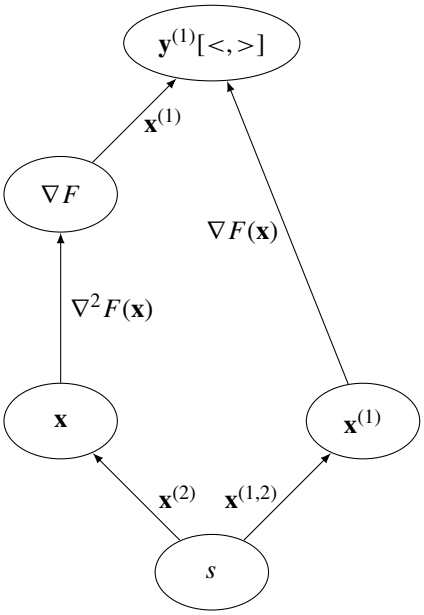


Figure 3.5. *Tangent-linear extension of the linearized DAG of the tangent-linear model* $\mathbf{y}^{(1)} = \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle$ *of* $\mathbf{y} = F(\mathbf{x})$.

Theorem 3.12. *The application of forward mode AD to the tangent-linear model yields the second-order tangent-linear model.*

Proof. The application of Definition 2.5 (definition of the tangent-linear model of a given multivariate vector function) to the tangent-linear model

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle,$$

with

$$\mathbf{x}^{(2)} \equiv \frac{\partial \mathbf{x}}{\partial s} \quad \text{and} \quad \mathbf{x}^{(1,2)} \equiv \frac{\partial \mathbf{x}^{(1)}}{\partial s},$$

yields

$$\mathbf{y}^{(1,2)} = \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1,2)} \rangle + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle, \tag{3.2}$$

and hence, for $\mathbf{x}^{(1,2)} = 0$, (3.1). □

A graphical illustration is provided in Figure 3.5. It shows the tangent-linear extension of the linearized DAG for the tangent-linear model of the original function $\mathbf{y} = F(\mathbf{x})$. Application of (1.5) yields (3.2). Tangent-linear directions are annotated with superscripts $^{(i)}$. The value of i marks the corresponding vector as a direction used for tangent-linear projection by the ith application of forward mode AD. For example, $\mathbf{x}^{(2)}$ denotes the derivative of $\mathbf{x}$ with respect to s introduced by the second application of forward mode AD. The variable $\mathbf{x}^{(1,2)}$ represents the corresponding derivative of $\mathbf{x}^{(1)}$.

Application of forward mode AD (by source transformation; see Section 3.2.1) to a tangent-linear subroutine

```
void t1_f(int n, int m, double* x, double* t1_x,
                        double* y, double* t1_y)
```

that implements

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

yields an implementation of

$$F^{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m :$$
$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1,2)} \end{pmatrix} = F^{(1,2)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1,2)}),$$

where

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\ \mathbf{y}^{(1,2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1,2)} \rangle + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle. \end{aligned}$$

A second-order tangent-linear subroutine with the following signature is generated:

```
void t2_t1_f(int n, int m,
   double *x, double *t2_x, double *t1_x, double *t2_t1_x,
   double *y, double *t2_y, double *t1_y, double *t2_t1_y);
```

Superscripts of second-order tangent-linear subroutine and variable names are replaced with the prefixes t2_ and t1_, that is, $\mathbf{v}^{(2)} \equiv$ t2_v and $\mathbf{v}^{(1,2)} \equiv$ t2_t1_v. The Hessian at point $\mathbf{x}$ can be accumulated at the computational cost of $O(n^2) \cdot Cost(F)$, where $Cost(F)$ denotes the computational cost of evaluating F, by setting $\mathbf{x}^{(1,2)} = 0$ initially and by letting $\mathbf{x}^{(2)}$ and $\mathbf{x}^{(1)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$. The computational complexity is the same as that of second-order finite differences. Some run-time savings result from the possible exploitation of symmetry as in second-order finite differences.

3.2.1 Source Transformation

The construction of second-order tangent-linear code turns out to be straightforward. It amounts to a simple augmentation of the given tangent-linear code.

The application of forward mode AD to the tangent-linear SAC in (2.2) yields

$$\begin{aligned}
v_j^{(1,2)} &= \left\langle \left(\frac{\partial^2 \varphi_j(v_i)_{i \prec j}}{\partial v_k \partial v_l} \right)_{\{k,l\} \prec j}, \left(v_i^{(1)}\right)_{i \prec j}, \left(v_i^{(2)}\right)_{i \prec j} \right\rangle \\
&\quad + \left\langle \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j}, \left(v_i^{(1,2)}\right)_{i \prec j} \right\rangle \\
v_j^{(1)} &= \left\langle \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j}, \left(v_i^{(1)}\right)_{i \prec j} \right\rangle \\
v_j^{(2)} &= \left\langle \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j}, \left(v_i^{(2)}\right)_{i \prec j} \right\rangle \\
v_j &= \varphi_j(v_i)_{i \prec j},
\end{aligned} \tag{3.3}$$

where $v_j^{(1)} = x_j^{(1)}$ and $v_j^{(2)} = x_j^{(2)}$ for $j = n, \ldots, n+p+m-1$. Initialization of $v_i^{(1,2)} = 0$ for $i = 0, \ldots, n-1$ yields the corresponding second-order tangent-linear projection of the Hessian in $v_j^{(1,2)}$ for $j = n+p, \ldots, n+p+m-1$. Practically, all assignments of the tangent-linear code are preceded by their respective tangent-linear versions as illustrated by the following example. Tangent-linear subroutine calls are replaced with calls to the respective second-order tangent-linear subroutine.

Example 3.13 We apply forward mode AD to the tangent-linear code t1_f from Example 2.8.

```
1  void t2_t1_f(int n, double* x, double* t2_x,
2                      double* t1_x, double* t2_t1_x,
3                      double& y, double& t2_y,
4                      double& t1_y, double& t2_t1_y) {
5      t2_t1_y=0;
6      t1_y=0;
7      t2_y=0;
8      y=0;
9      for (int i=0;i<n;i++) {
10         t2_t1_y=t2_t1_y+2*(t2_x[i]*t1_x[i]+x[i]*t2_t1_x[i]);
11         t1_y=t1_y+2*x[i]*t1_x[i];
12         t2_y=t2_y+2*x[i]*t2_x[i];
13         y=y+x[i]*x[i];
14     }
15     t2_t1_y=2*t2_y*t1_y+2*y*t2_t1_y;
16     t1_y=2*y*t1_y;
17     t2_y=2*y*t2_y;
18     y=y*y;
19 }
```

According to Tangent-Linear Code Generation Rule 1, all **double** parameters of the tangent-linear subroutine (x, t1_x, y,and t1_y) are duplicated. The new variables are augmented with the t2_ prefix. Inputs are augmented with inputs and outputs with outputs. Tangent-linear

versions of all assignments are inserted into the tangent-linear code in lines 5, 7, 10, 12, 15, and 17. The flow of control remains unchanged. First-order tangent-linear projections of the gradient in directions t1_x and t2_x are returned in t1_y and t2_y, respectively. The function value is returned in y. If only second derivatives are required, then dead code elimination results in further optimization of the second-order tangent-linear code. For example, lines 8, 13, and 16–18 become obsolete in this case. Knowing that all entries of t2_t1_x are equal to zero, the assignment in line 10 can be simplified to t2_t1_y=t2_t1_y+2*t2_x[i]*t1_x[i].

The following driver computes all entries of the lower triangular part of the Hessian.

```
1 int main() {
2   const int n=4; int i,j;
3   double x[n],t1_x[n],t2_x[n],t2_t1_x[n],y,t1_y,t2_y,t2_t1_y;
4
5   for (j=0;j<n;j++) { x[j]=1; t2_t1_x[j]=t2_x[j]=t1_x[j]=0; }
6   for (j=0;j<n;j++) {
7     t1_x[j]=1;
8     for (i=0;i<=j;i++) {
9       t2_x[i]=1;
10      t2_t1_f(n,x,t2_x,t1_x,t2_t1_x,y,t2_y,t1_y,t2_t1_y);
11      cout << "H[" << j << "][" << i << "]=" << t2_t1_y << endl;
12      t2_x[i]=0;
13    }
14    t1_x[j]=0;
15  }
16  return 0;
17 }
```

It exploits the fact that x is not overwritten inside of f. Consequently, neither t1_x nor t2_x is overwritten in t2_t1_f, and their entries can be set and reset individually in lines 7, 9, 12, and 14. $O(n^2)$ evaluations of the second-order tangent-linear code are performed in line 10. The Hessian entries are returned individually in t2_t1_y, and they are printed to the standard output in line 11. ∎

3.2.2 Overloading

The computation of second derivatives is supported by dco through the provision of the second-order scalar tangent-linear data type dco_t2s_t1s_type whose value (v) and derivative (t) components are tangent-linear scalars of type dco_t1s_type.

```
class dco_t2s_t1s_type {
  public :
    dco_t1s_type v,t;
    ...
};
```

The definition of the arithmetic operators and intrinsic functions does not yield any surprises; for example,

```
dco_t2s_t1s_type operator*(const dco_t2s_t1s_type& x1,
                           const dco_t2s_t1s_type& x2) {
  dco_t2s_t1s_type tmp;
```

```
    tmp.v=x1.v*x2.v;
    tmp.t=x1.t*x2.v+x1.v*x2.t;
    return tmp;
  }.
```

The driver program in Listing 3.1 uses the implementation of the second-order tangent-linear model by overloading to compute the Hessian $\nabla^2 F$ of (1.2) for $n = 4$ at the point $x_i = 1$ for $i = 0, \ldots, 3$. All data members of variables of type dco_t2s_t1s_type are initialized to zero at the time of construction. Hence, $x_i^{(1,2)} \equiv$ x[i].t.t does not need to be initialized explicitly. Both $x_i^{(1)} \equiv$ x[i].t.v and $x_i^{(2)} \equiv$ x[i].v.t range in lines 17 and 19 independently over the Cartesian basis vectors in $\mathbb{R}^n$. Resetting can be restricted in lines 24 and 21 to the individual components as x is not overwritten in f. The desired Hessian entries are retrieved from y.t.t $\equiv y^{(1,2)}$, and they are printed in line 22 to the standard output.

Listing 3.1. *Driver for second-order tangent-linear code by overloading.*

```
1 #include<iostream>
2 using namespace std;
3 #include "dco_t2s_t1s_type.hpp"
4
5 const int n=4;
6
7 void f(dco_t2s_t1s_type *x, dco_t2s_t1s_type &y) {
8   y=0;
9   for (int i=0;i<n;i++) y=y+x[i]*x[i];
10  y=y*y;
11 }
12
13 int main() {
14   dco_t2s_t1s_type x[n], y;
15   for (int i=0;i<n;i++) x[i]=1.;
16   for (int j=0;j<n;j++) {
17     x[j].t.v=1;
18     for (int i=0;i<=j;i++) {
19       x[i].v.t=1;
20       f(x,y);
21       x[i].v.t=0;
22       cout << "H[" << j << "][" << i << "]=" << y.t.t << endl;
23     }
24     x[j].t.v=0;
25   }
26   return 0;
27 }
```

Let the **class** dco_t2s_t1s_type be defined in the files `dco_t2s_t1s_type.hpp` and `dco_t2s_t1s_type.cpp`, and let the driver program be stored as `main.cpp`. Suppose that the files `dco_t1s_type.hpp` and `dco_t1s_type.cpp` that implement the first-order tangent-linear model by overloading are located in the same directory. An executable

Table 3.1. *Run times for second-order tangent-linear code (in seconds). In order to determine the relative computational complexity $\mathcal{R}$ of the derivative code, n function evaluations are compared with n evaluations of the second-order tangent-linear code (t2_t1_f) and with the same number of evaluations of an implementation of the second-order tangent-linear model by overloading (dco_t2s_t1s_f). As in Chapter 2, the compiler optimizations are either switched off (*`g++ -O0`*) or the full set of optimizations is enabled (*`g++ -O3`*). We observe a factor of approximately 2.3 when comparing the run time of a single run of the second-order tangent-linear code with that of an original function evaluation in the rightmost column. Implementation by overloading adds a factor of almost 20 due to less effective compiler optimization.*

	`g++ -O0`			`g++ -O3`			$\mathcal{R}$
n	10^4	$2\cdot10^4$	$4\cdot10^4$	10^4	$2\cdot10^4$	$4\cdot10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t2_t1_f	3.0	12.0	47.3	0.5	1.9	7.2	≈ 2.3
dco_t2s_t1s_f	35.1	134.6	562.3	8.6	32.9	128.4	≈ 41.5

is built by calling

```
$(CPPC) -c dco_t1s_type.cpp
$(CPPC) -c dco_t2s_t1s_type.cpp
$(CPPC) -c main.cpp
$(CPPL) -o main dco_t1s_type.o dco_t2s_t1s_type.o main.o
```

where `$(CPPC)` and `$(CPPL)` denote the native C++ compiler and linker, respectively. Run time results are reported in Table 3.1.

3.3 Second-Order Adjoint Code

The remaining three approaches to the generation of second derivative code involve at least one application of reverse mode AD. According to Lemmas 3.9 and 3.10, all three alternatives implement the *second-order adjoint model* that is defined next.

Definition 3.14. *The Hessian $\nabla^2 F = \nabla^2 F(\mathbf{x}) \in \mathbb{R}^{m\times n\times n}$ of a multivariate vector function $\mathbf{y} = F(\mathbf{x})$, $F : \mathbb{R}^n \to \mathbb{R}^m$, induces a bilinear mapping $\mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ defined by*

$$(\mathbf{v}, \mathbf{w}) \mapsto \langle \mathbf{w}, \nabla^2 F, \mathbf{v} \rangle.$$

The function $F'' : \mathbb{R}^{2\cdot n+m} \to \mathbb{R}^n$ that is defined as

$$F''(\mathbf{x}, \mathbf{v}, \mathbf{w}) \equiv \langle \mathbf{w}, \nabla^2 F(\mathbf{x}), \mathbf{v} \rangle \tag{3.4}$$

is referred to as the second-order adjoint model *of F.*

We distinguish between implementations generated in forward-over-reverse, reverse-over-forward, and reverse-over-reverse modes. The corresponding second-order adjoint models will be denoted by $F^{(2)}_{(1)}$, $F^{(1)}_{(2)}$, and $F_{(1,2)}$, respectively. All three models yield the same computational complexity of $O(n \cdot m) \cdot Cost(F)$ for the accumulation of the whole Hessian. Actual run times and memory requirements vary as illustrated in Section 3.3.1.

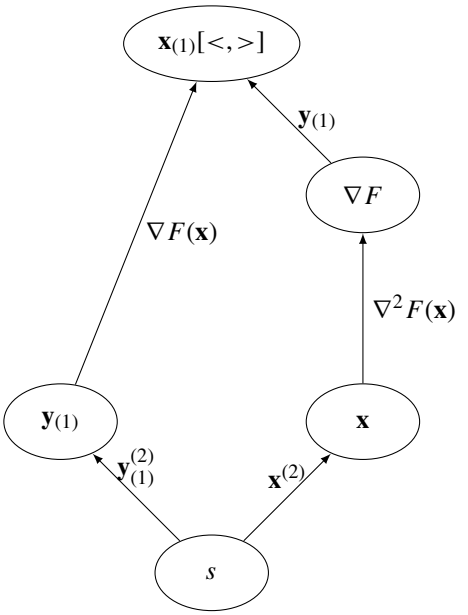


Figure 3.6. *Tangent-linear extension of the linearized DAG of the adjoint model* $\mathbf{x}_{(1)} = \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle$ *of* $\mathbf{y} = F(\mathbf{x})$.

Forward-over-Reverse Mode

Theorem 3.15. *The application of forward mode AD to the adjoint model yields an implementation of the second-order adjoint model.*

Proof. The application of forward mode AD as defined in Section 2.1.1 to the adjoint model $\mathbf{x}_{(1)} = \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle$ gives

$$\begin{aligned} \mathbf{x}_{(1)}^{(2)} &= \langle \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle, \\ \mathbf{x}_{(1)} &= \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle, \end{aligned}$$

and hence, with $\mathbf{y}_{(1)}^{(2)} = 0$, (3.4). □

A graphical illustration in the form of the tangent-linear extension of the linearized DAG of the adjoint model can be found in Figure 3.6, where

$$\mathbf{x}^{(2)} \equiv \frac{\partial \mathbf{x}_{(1)}}{\partial s}, \quad \mathbf{y}_{(1)}^{(2)} \equiv \frac{\partial \mathbf{y}_{(1)}}{\partial s},$$

and where $\mathbf{x}_{(1)}^{(2)}$ is computed as the partial derivative of $\mathbf{x}_{(1)}$ with respect to s according to (1.5).

In forward-over-reverse mode, forward mode AD is applied to the adjoint code $(\mathbf{y},\mathbf{x}_{(1)},\mathbf{y}_{(1)}) = F_{(1)}(\mathbf{x},\mathbf{x}_{(1)},\mathbf{y}_{(1)})$ yielding

$$F_{(1)}^{(2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m :$$
$$(\mathbf{y},\mathbf{y}^{(2)},\mathbf{x}_{(1)},\mathbf{x}_{(1)}^{(2)},\mathbf{y}_{(1)},\mathbf{y}_{(1)}^{(2)}) = F_{(1)}^{(2)}(\mathbf{x},\mathbf{x}^{(2)},\mathbf{x}_{(1)},\mathbf{x}_{(1)}^{(2)},\mathbf{y}_{(1)},\mathbf{y}_{(1)}^{(2)}),$$

such that

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(1)}^{(2)} &= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y}_{(1)} &= 0 \\
\mathbf{y}_{(1)}^{(2)} &= 0.
\end{aligned} \tag{3.5}$$

For $m = 1$, we get $\langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle = \mathbf{y}_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$. The corresponding second-order adjoint subroutine has the following signature:

```
void t2_a1_f(int n, int m,
   double *x, double *t2_x, double *a1_x, double *t2_a1_x,
   double *y, double *t2_y, double *a1_y, double *t2_a1_y);
```

Subscripts (superscripts) of second-order adjoint subroutine and variable names are replaced with the prefixes a1_ and t2_; for example, $\mathbf{v}^{(2)} \equiv$ t2_v and $\mathbf{v}_{(1)}^{(2)} \equiv$ t2_a1_v. The computation of projections of the Hessian in directions $\mathbf{x}^{(2)}$ and $\mathbf{y}_{(1)}$ requires $\mathbf{y}_{(1)}^{(2)} = 0$ initially. The entire Hessian can be accumulated at a computational cost of $O(n \cdot m) \cdot Cost(F)$ by letting $\mathbf{x}^{(2)}$ and $\mathbf{y}_{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. For $m = 1$, a single Hessian-vector product can be computed at the computational cost of $O(1) \cdot Cost(F)$, that is, at a constant multiple of the cost of evaluating F. The magnitude of this constant factor depends on details of the implementation as illustrated in Sections 3.3.1 and 3.3.2.

Reverse-over-Forward Mode

Theorem 3.16. *The application of reverse mode AD to the tangent-linear model yields an implementation of the second-order adjoint model.*

Proof. The application of reverse mode AD as defined in Section 2.2.1 to the tangent-linear model $\mathbf{y}^{(1)} = \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle$ gives

$$\begin{aligned}
\mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}^{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle
\end{aligned}$$

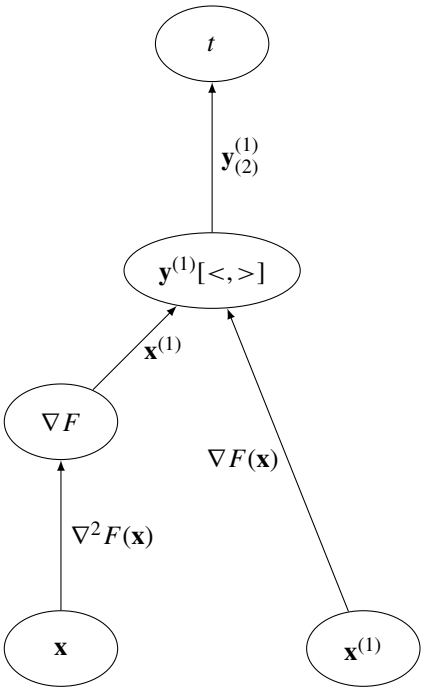


Figure 3.7. *Adjoint extension of the linearized DAG of the tangent-linear model* $\mathbf{y}^{(1)} = \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle$ *of* $\mathbf{y} = F(\mathbf{x})$.

$$\begin{aligned} \mathbf{x}_{(2)}^{(1)} &= \mathbf{x}_{(2)}^{(1)} + \langle \mathbf{y}_{(2)}^{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(2)}^{(1)} &= 0. \end{aligned}$$

With $\mathbf{x}_{(2)} = 0$ initially, the second line yields (3.4). □

A graphical illustration in the form of the adjoint extension of the linearized DAG of the tangent-linear model can be found in Figure 3.7, where

$$\mathbf{y}_{(2)}^{(1)} \equiv \frac{\partial t}{\partial \mathbf{y}^{(1)}},$$

and where $\mathbf{x}_{(2)}$ and $\mathbf{x}_{(2)}^{(1)}$ are computed according to (1.5) as partial derivatives of t with respect to $\mathbf{x}$ and $\mathbf{x}^{(1)}$.

The application of reverse mode AD to a tangent-linear code that implements

$$(\mathbf{y}, \mathbf{y}^{(1)}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

results in

$$\begin{aligned} F_{(2)}^{(1)} &: \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m : \\ &(\mathbf{y}, \mathbf{y}^{(1)}, \mathbf{x}_{(2)}, \mathbf{x}_{(2)}^{(1)}, \mathbf{y}_{(2)}, \mathbf{y}_{(2)}^{(1)}) = F_{(2)}^{(1)}(\mathbf{x}, \mathbf{x}_{(2)}, \mathbf{x}^{(1)}, \mathbf{x}_{(2)}^{(1)}, \mathbf{y}_{(2)}, \mathbf{y}_{(2)}^{(1)}), \end{aligned}$$

where

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}^{(1)}_{(2)} &= \mathbf{x}^{(1)}_{(2)} + \langle \mathbf{y}^{(1)}_{(2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(2)} &= 0 \\
\mathbf{y}^{(1)}_{(2)} &= 0.
\end{aligned} \tag{3.6}$$

The corresponding second-order adjoint subroutine has the following signature:

```
void a2_t1_f(int n, int m,
   double *x, double *a2_x, double *t1_x, double *a2_t1_x,
   double *y, double *a2_y, double *t1_y, double *a2_t1_y);
```

Subscripts (superscripts) of second-order adjoint subroutine and variable names are replaced with the prefixes t1_ and a2_; for example, $\mathbf{v}_{(2)} \equiv$ a2_v and $\mathbf{v}^{(1)}_{(2)} \equiv$ a2_t1_v. The entire Hessian can be accumulated at a computational cost of $O(n \cdot m) \cdot Cost(F)$ by letting $\mathbf{x}^{(1)}$ and $\mathbf{y}^{(1)}_{(2)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. Hence, for $m = 1$, the computational cost of products of the Hessian with a vector $\mathbf{x}^{(1)} \in \mathbb{R}^n$ is $O(1) \cdot Cost(F)$.

Reverse-over-Reverse Mode

Theorem 3.17. *The application of reverse mode AD to the adjoint model yields an implementation of the second-order adjoint model.*

Proof. The application of reverse mode AD as defined in Section 2.2.1 to the adjoint model $\mathbf{x}_{(1)} = \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle$ gives

$$\begin{aligned}
\mathbf{x}_{(1)} &= \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,2)} &= \mathbf{y}_{(1,2)} + \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(1,2)} &= 0.
\end{aligned}$$

If $\mathbf{x}_{(2)}$ is equal to zero initially, then the second line yields (3.4) as, according to Lemma 3.9, $\langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle = \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}_{(1,2)} \rangle$. □

A graphical illustration in the form of the adjoint extension of the linearized DAG of the adjoint model can be found in Figure 3.8, where

$$\mathbf{x}_{(1,2)} \equiv \frac{\partial t}{\partial \mathbf{x}_{(1)}},$$

and where $\mathbf{x}_{(2)}$ and $\mathbf{y}_{(1,2)}$ are computed according to (1.5) as the partial derivatives of t with respect to $\mathbf{x}$ and $\mathbf{y}_{(1)}$.

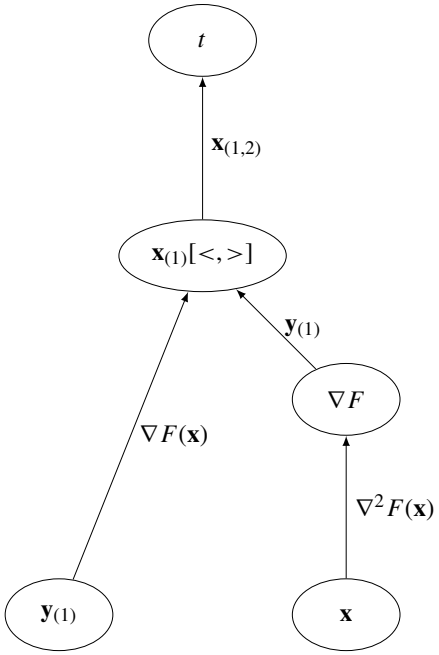


Figure 3.8. *Adjoint extension of the linearized DAG of the adjoint model* $\mathbf{x}_{(1)} = \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle$ *of* $\mathbf{y} = F(\mathbf{x})$.

The application of reverse mode AD with required data stack s and result checkpoint r to an adjoint code that implements $(\mathbf{y}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}) = F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$ yields

$$F_{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m :$$
$$(\mathbf{y}, \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \mathbf{y}_{(1,2)}, \mathbf{y}_{(2)}) = F_{(1,2)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}, \mathbf{x}_{(2)}, \mathbf{x}_{(1,2)}, \mathbf{y}_{(2)}),$$

where

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
s[0] &= \mathbf{y}_{(1)} \\
\mathbf{y}_{(1)} &= 0 \\
r[0] &= \mathbf{y};\ r[1] = \mathbf{x}_{(1)};\ r[2] = \mathbf{y}_{(1)} \\
\mathbf{y}_{(1)} &= s[0] \\
\mathbf{y}_{(1,2)} &= 0 \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,2)} &= \mathbf{y}_{(1,2)} + \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F \rangle;\ \mathbf{y}_{(2)} = 0 \\
\mathbf{y} &= r[0];\ \mathbf{x}_{(1)} = r[1];\ \mathbf{y}_{(1)} = r[2].
\end{aligned}$$

The resulting second-order adjoint code computes

$$\begin{aligned}
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F \rangle + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,2)} &= \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(2)} &= 0 \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1)} &= 0.
\end{aligned} \tag{3.7}$$

The corresponding second-order adjoint subroutine has the following signature:

```
void a2_a1_f(int n, int m,
   double *x, double *a2_x, double *a1_x, double *a2_a1_x,
   double *y, double *a2_y, double *a1_y, double *a2_a1_y);
```

Subscripts of second-order adjoint subroutine and variable names are replaced with the prefixes a1_ and a2_; for example, $\mathbf{v}_{(2)} \equiv$ a2_v and $\mathbf{v}_{(1,2)} \equiv$ a2_a1_v. The entire Hessian can be accumulated at a computational cost of $O(n \cdot m) \cdot Cost(F)$ by letting $\mathbf{x}_{(1,2)}$ and $\mathbf{y}_{(1)}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. For $m = 1$, a single Hessian-vector product can be computed at a computational cost of $O(1) \cdot Cost(F)$.

3.3.1 Source Transformation

Second derivative code is most likely to be used in the context of numerical algorithms that require second as well as first derivatives. The Hessian or projections thereof may be needed in addition to the gradient at the current point. If we assume that an adjoint code exists in order to compute the gradient efficiently, then a second-order adjoint code generated in "forward-over-reverse" mode is the most likely choice despite the fact that the computational complexity of computing a projected Hessian is the same in "reverse-over-forward" mode. In practice, the repeated reversal of the data flow in "reverse-over-reverse" mode turns out to result in a considerable computational overhead.

Forward-over-Reverse Mode

The application of forward mode AD to an adjoint SAC generated in incremental reverse mode as defined in (2.9) yields

$$\begin{aligned}
&\text{for } j = n, \ldots, n+p+m-1 \\
&\qquad v_j^{(2)} = \left\langle \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j}, \left(v_k^{(2)} \right)_{k \prec j} \right\rangle \\
&\qquad v_j = \varphi_j(v_i)_{i \prec j}
\end{aligned}$$

for $j = n + p + m - 1, \ldots, n$

$$\begin{aligned}
\left(v_{(1)i}^{(2)}\right)_{i \prec j} &= \left(v_{(1)i}^{(2)}\right)_{i \prec j} + \left\langle v_{(1)j}, \left(\frac{\partial^2 \varphi_j(v_i)_{i \prec j}}{\partial v_k \partial v_l}\right)_{\{k,l\} \prec j}, \left(v_k^{(2)}\right)_{k \prec j} \right\rangle \\
&\quad + \left\langle v_{(1)j}^{(2)}, \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k}\right)_{k \prec j} \right\rangle \\
\left(v_{(1)i}\right)_{i \prec j} &= \left(v_{(1)i}\right)_{i \prec j} + \left\langle v_{(1)j}, \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k}\right)_{k \prec j} \right\rangle.
\end{aligned} \tag{3.8}$$

As in (2.9), the $v_{(1)n+p+j}$ are assumed to be initialized to $y_{(1)j}$ for $j = 0, \ldots, m-1$. Moreover, the caller is expected to set $v_j^{(2)} = x_j^{(2)}$ and $v_{(1)n+p+j}^{(2)} = v_{(1)i}^{(2)} = 0$ for $j = 0, \ldots, m-1$ and $i = 0, \ldots, n-1$ if projections of the Hessian in directions $\mathbf{y}_{(1)} \equiv (v_{(1)n+p+j})_{j=0,\ldots,m-1}$ and $\mathbf{x}^{(2)} \equiv (v_j^{(2)})_{j=0,\ldots,n-1}$ shall be returned in $\mathbf{x}_{(1)}^{(2)} \equiv (v_{(1)j}^{(2)})_{j=0,\ldots,n-1}$. Adjoints of intermediate variables are initialized to zero by default, which is exploited in the following example.

Example 3.18 For illustration, we consider the scalar function $y = f(\mathbf{x}) = \sin(x_0 \cdot x_1)$. In forward-over-reverse mode, the SAC and its adjoint are differentiated in forward mode yielding

[tangent-linear SAC]

$$\begin{aligned}
v_2^{(2)} &= v_0 \cdot v_1^{(2)} + v_0^{(2)} \cdot v_1 \\
v_2 &= v_0 \cdot v_1 \\
v_3^{(2)} &= \cos(v_2) \cdot v_2^{(2)} \\
v_3 &= \sin(v_2)
\end{aligned}$$

[tangent-linear adjoint SAC]

$$\begin{aligned}
v_{(1)2}^{(2)} &= v_{(1)3}^{(2)} \cdot \cos(v_2) - v_{(1)3} \cdot \sin(v_2) \cdot v_2^{(2)} \\
v_{(1)2} &= v_{(1)3} \cdot \cos(v_2) \\
v_{(1)0}^{(2)} &= v_1^{(2)} \cdot v_{(1)2} + v_1 \cdot v_{(1)2}^{(2)} \\
v_{(1)0} &= v_1 \cdot v_{(1)2} \\
v_{(1)1}^{(2)} &= v_0^{(2)} \cdot v_{(1)2} + v_0 \cdot v_{(1)2}^{(2)} \\
v_{(1)1} &= v_0 \cdot v_{(1)2}.
\end{aligned}$$

It is straightforward to verify that by setting $v_{(1)3} = y_{(1)}$ and $v_i^{(2)} = x_i^{(2)}$ for $i = 0, 1$ we obtain the scaled Hessian-vector product

$$\mathbf{x}_{(1)}^{(2)} \equiv \begin{pmatrix} x_{(1)0}^{(2)} \\ x_{(1)1}^{(2)} \end{pmatrix} = y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \begin{pmatrix} x_{(1)0}^{(2)} \\ x_{(1)1}^{(2)} \end{pmatrix},$$

where

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} -x_1^2 \cdot \sin(x_0 \cdot x_1) & \cos(x_0 \cdot x_1) - x_0 \cdot x_1 \cdot \sin(x_0 \cdot x_1) \\ \cos(x_0 \cdot x_1) - x_0 \cdot x_1 \cdot \sin(x_0 \cdot x_1) & -x_0^2 \cdot \sin(x_0 \cdot x_1) \end{pmatrix}$$

if both $\mathbf{x}_{(1)}^{(2)} = (v_{(1)0}^{(2)}, v_{(1)1}^{(2)})^T$ and $y_{(1)}^{(2)} = v_{(1)3}^{(2)}$ are initialized to zero. ∎

Example 3.19 The application of forward mode AD to the adjoint version

```
void a1_f(int n, double* x, double* a1_x,
                 double& y, double a1_y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  required_double.push(y);
  y=y*y;

  y=required_double.top(); required_double.pop();
  a1_y=2*y*a1_y;
  for (int i=n-1;i>=0;i--)
    a1_x[i]=a1_x[i]+2*x[i]*a1_y;
}
```

of the implementation of (1.2) given in Section 1.1.2 yields

```
 1 void t2_a1_f(int n, double* x, double* t2_x,
 2                     double* a1_x, double* t2_a1_x,
 3                     double& y, double& t2_y,
 4                     double a1_y, double t2_a1_y) {
 5   t2_y=0; y=0;
 6   for (int i=0;i<n;i++) {
 7     t2_y=t2_y+2*x[i]*t2_x[i]; y=y+x[i]*x[i];
 8   }
 9   t2_required_double.push(t2_y); required_double.push(y);
10   t2_y=2*y*t2_y; y=y*y;
11
12   t2_result_double.push(t2_y); result_double.push(y);
13
14   t2_y=t2_required_double.top(); t2_required_double.pop();
15   y=required_double.top(); required_double.pop();
16   t2_a1_y=2*(t2_y*a1_y+y*t2_a1_y);
17   a1_y=2*y*a1_y;
18   for (int i=n-1;i>=0;i--) {
19     t2_a1_x[i]=t2_a1_x[i]+2*(t2_x[i]*a1_y+x[i]*t2_a1_y);
20     a1_x[i]=a1_x[i]+2*x[i]*a1_y;
21   }
22   t2_a1_y=0; a1_y=0;
23
24   t2_y=t2_result_double.top(); t2_result_double.pop();
25   y=result_double.top(); result_double.pop();
26 }
```

The store/restore mechanism for required data prevents y and t2_y from holding the correct function and first derivative values on output. See lines 9, 14, and 15. If the correct value of y is preserved by writing a result checkpoint in the adjoint code, then the tangent-linear version of the store/restore statements for y also recovers the correct directional derivative t2_y. See lines 12, 24, and 25.

The following driver computes the Hessian of (1.2) at point $x_i = 1$, $i = 0,\ldots,3$, set in line 4.

```
1 int main() {
2   const int n=4;
3   double x[n],y,t2_x[n],t2_y,a1_x[n],a1_y,t2_a1_x[n],t2_a1_y;
4   for (int i=0;i<n;i++) { x[i]=1; t2_x[i]=0; }
5   t2_a1_y=0;
6   for (int i=0;i<n;i++) {
7     for (int j=0;j<n;j++) a1_x[j]=t2_a1_x[j]=0;
8     a1_y=1;
9     t2_x[i]=1;
10    t2_a1_f(n,x,t2_x,a1_x,t2_a1_x,y,t2_y,a1_y,t2_a1_y);
11    for (int j=0;j<=i;j++)
12      cout << "H[" << i << "][" << j << "]="
13           << t2_a1_x[j] << endl;
14    t2_x[i]=0;
15  }
16  return 0;
17 }
```

It contains in line 6 a loop over the Cartesian basis vectors in $\mathbb{R}^4$ that are assigned in line 9 to the initially zero vector t2_x (see line 6) for the fixed first-order adjoint a1_y=1 of the original output, set in line 8. The corresponding t2_x entries can be reset to zero individually in line 14 as x is not overwritten in a1_f; hence, it is not modified by t2_a1_f either. Initialization of t2_a1_y in line 5 is crucial for avoiding the addition of first derivative information to t2_a1_x. According to (3.5), t2_a1_y is kept equal to zero by the repeated calls of t2_a1_f. The columns of the Hessian are returned in t2_a1_x and they are printed to the standard output in lines 11–13. Both a1_x and t2_a1_x need to be reset to zero prior to each iteration (see line 7) because of the incremental nature of the adjoint code.

A total of $n = 4$ evaluations of the second-order adjoint code are required to compute all entries of the Hessian. A single Hessian-vector product is obtained at a constant factor of the cost of evaluating the original code in Section 1.1.2. Refer to Table 3.2 for run-time measurements. ∎

Reverse-over-Forward Mode

To obtain an implementation of the second-order adjoint model in reverse-over-forward mode, reverse mode AD is applied to (2.2) yielding

$$\begin{aligned}
&\text{for } j = n,\ldots,n+p+m-1\\
&\quad v_j^{(1)} = \left\langle \left(\frac{\partial \varphi_j(v_i)_{i\prec j}}{\partial v_k} \right)_{k\prec j}, \left(v_k^{(1)}\right)_{k\prec j} \right\rangle\\
&\quad v_j = \varphi_j(v_i)_{i\prec j}
\end{aligned}$$

for $j = n+p+m-1,\ldots,n$

$$\begin{aligned}
\left(v_{(2)i}\right)_{i\prec j} &= \left(v_{(2)i}\right)_{i\prec j} + \left\langle v_{(2)j}, \left(\frac{\partial \varphi_j(v_i)_{i\prec j}}{\partial v_k}\right)_{k\prec j}\right\rangle \\
\left(v_{(2)i}\right)_{i\prec j} &= \left(v_{(2)i}\right)_{i\prec j} + \left\langle v^{(1)}_{(2)j}, \left(\frac{\partial^2 \varphi_j(v_i)_{i\prec j}}{\partial v_k \partial v_l}\right)_{\{k,l\}\prec j}, \left(v^{(1)}_k\right)_{k\prec j}\right\rangle \\
\left(v^{(1)}_{(2)k}\right)_{k\prec j} &= \left(v^{(1)}_{(2)k}\right)_{k\prec j} + \left\langle v^{(1)}_{(2)j}, \left(\frac{\partial \varphi_j(v_i)_{i\prec j}}{\partial v_k}\right)_{k\prec j}\right\rangle.
\end{aligned} \tag{3.9}$$

Setting $v^{(1)}_{(2)n+p+i} = y^{(1)}_{(2)i}$ for $i = 0,\ldots,m-1$ and $v^{(1)j} = x^{(1)j}$ for $j = 0,\ldots,n-1$ yields a projection of the Hessian in directions $\mathbf{y}^{(1)}_{(2)}$ and $\mathbf{x}^{(1)}$ if $v_{(2)n+p+i} = y_{(2)i}$ and $v_{(2)j} = x_{(2)j}$ are initialized to zero for $i = 0,\ldots,m-1$ and $i = 0,\ldots,n-1$, respectively. Adjoints of intermediate variables are initialized to zero by default, which is exploited in the following example.

Example 3.20 Again, we consider $y = f(\mathbf{x}) = \sin(x_0 \cdot x_1)$. In reverse-over-forward mode the original tangent-linear SAC is succeeded by its adjoint yielding

[tangent-linear SAC]

$$\begin{aligned}
v^{(1)}_2 &= v_0 \cdot v^{(1)}_1 + v^{(1)}_0 \cdot v_1 \\
v_2 &= v_0 \cdot v_1 \\
v^{(1)}_3 &= \cos(v_2) \cdot v^{(1)}_2 \\
v_3 &= \sin(v_2)
\end{aligned}$$

adjoint [tangent-linear SAC]:

$$\begin{aligned}
v_{(2)2} &= v_{(2)3} \cdot \cos(v_2) \\
v_{(2)2} &= v_{(2)2} - v^{(1)}_{(2)3} \cdot \sin(v_2) \cdot v^{(1)}_2 \\
v^{(1)}_{(2)2} &= v^{(1)}_{(2)3} \cdot \cos(v_2) \\
v_{(2)0} &= v_1 \cdot v_{(2)2} \\
v_{(2)1} &= v_0 \cdot v_{(2)2} \\
v_{(2)0} &= v_{(2)0} + v^{(1)}_{(2)2} \cdot v^{(1)}_1 \\
v^{(1)}_{(2)1} &= v^{(1)}_{(2)2} \cdot v_0 \\
v_{(2)1} &= v_{(2)1} + v^{(1)}_{(2)2} \cdot v^{(1)}_0 \\
v^{(1)}_{(2)0} &= v^{(1)}_{(2)2} \cdot v_1.
\end{aligned}$$

It is straightforward to verify that by setting $v^{(1)}_{(2)3} = y^{(1)}_{(2)}$ and $v^{(1)}_i = x^{(1)}_i$ for $i = 0,1$ we obtain the scaled Hessian-vector product

$$\mathbf{x}_{(2)} \equiv \begin{pmatrix} x_{(2)0} \\ x_{(2)1} \end{pmatrix} = y_{(2)}^{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \begin{pmatrix} x_0^{(1)} \\ x_1^{(1)} \end{pmatrix}$$

if both $\mathbf{x}_{(2)} = (v_{(2)0}, v_{(2)1})^T$ and $y_{(2)} = v_{(2)3}$ are initialized to zero. ∎

Example 3.21 The application of reverse mode AD to the tangent-linear code

```
void t1_f(int n, double* x, double* t1_x,
                 double& y, double& t1_y) {
  t1_y=0; y=0;
  for (int i=0;i<n;i++) {
    t1_y=t1_y+2*x[i]*t1_x[i]; y=y+x[i]*x[i];
  }
  t1_y=2*y*t1_y; y=y*y;
}
```

from Example 2.8 yields

```
1 void a2_t1_f(int n, double* x, double* a2_x,
2                     double* t1_x, double* a2_t1_x,
3                     double& y, double& a2_y,
4                     double& t1_y, double& a2_t1_y) {
5   t1_y=0; y=0;
6   for (int i=0;i<n;i++) {
7     t1_y=t1_y+2*x[i]*t1_x[i]; y=y+x[i]*x[i];
8   }
9   required_double.push(t1_y); t1_y=2*y*t1_y;
10  required_double.push(y); y=y*y;
11
12  y=required_double.top(); required_double.pop();
13  a2_y=2*y*a2_y;
14  t1_y=required_double.top(); required_double.pop();
15  a2_y=a2_y+2*t1_y*a2_t1_y;
16  a2_t1_y=2*y*a2_t1_y;
17  for (int i=n-1;i>=0;i--) {
18    a2_x[i]=a2_x[i]+2*x[i]*a2_y;
19    a2_x[i]=a2_x[i]+2*t1_x[i]*a2_t1_y;
20    a2_t1_x[i]=a2_t1_x[i]+2*x[i]*a2_t1_y;
21  }
22  a2_y=0; a2_t1_y=0;
23 }
```

Values of y and t1_y that are used in lines 13, 15, and 16 of the reverse section are overwritten by the assignments in lines 9 and 10. Consequently, their values are stored on the required data stack in lines 9 and 10, and they are restored in lines 12 and 14.

The following driver computes the Hessian of (1.2) at point $x_i = 1$, $i = 0, \ldots, 3$, set in line 4.

```
1 int main() {
2   const int n=4;
3   double x[n],y,a2_x[n],a2_y,t1_x[n],t1_y,a2_t1_x[n],a2_t1_y;
4   for (int i=0;i<n;i++) { x[i]=1; t1_x[i]=0; }
```

```
5   a2_y=0;
6   for (int i=0;i<n;i++) {
7     for (int j=0;j<n;j++) a2_x[j]=a2_t1_x[j]=0;
8     a2_t1_y=1;
9     t1_x[i]=1;
10    a2_t1_f(n,x,a2_x,t1_x,a2_t1_x,y,a2_y,t1_y,a2_t1_y);
11    for (int j=0;j<=i;j++)
12      cout << "H[" << i << "][" << j << "]=" << a2_x[j] << endl;
13    t1_x[i]=0;
14  }
15  return 0;
16 }
```

It contains in line 6 a loop over the Cartesian basis vectors in $\mathbb{R}^4$ that are assigned to the initially zero (see line 4) vector t1_x in line 9. The second-order adjoint a2_t1_y of the original output is set to one in line 8. The corresponding t1_x entries can be reset to zero individually in line 13 as t1_x is not overwritten in t1_f and hence is not modified by a2_t1_f. Initialization of a2_y=0 in line 5 is crucial for avoiding the addition of first derivative information to a2_x. According to (3.6), a2_y is kept equal to zero by the repeated calls of a2_t1_f. The columns of the Hessian are returned in a2_x, and they are printed to the standard output in lines 11 and 12. Both a2_x and a2_t1_x need to be reset to zero in line 7 prior to each call of a2_t1_f as the adjoint code is generated in incremental reverse mode.

A total of n = 4 evaluations of the second-order adjoint code are required to compute all entries of the Hessian. A single Hessian-vector product is obtained at a constant factor of the cost of evaluating the original code in Section 1.1.2. Typically, second-order adjoint code generated in reverse-over-forward mode is slightly less efficient than its competitor that is generated in forward-over-reverse mode. The latter can be optimized more effectively by the native C++ compiler. The impact of this effect is almost negligible for our simple example as illustrated by the run-time measurements in Table 3.2. It turns out to be more significant for larger simulations. ∎

Table 3.2. *Run times for second-order adjoint code (in seconds). In order to determine the relative computational complexity $\mathcal{R}$ of the derivative code, n function evaluations are compared with a full Hessian accumulation. We observe a factor of approximately* 3.3 *when comparing the time taken by a single run of the second-order adjoint code that was generated in forward-over-reverse mode with that of an original function evaluation in the right-most column. Reverse-over-forward mode performs slightly worse if compiler optimization is switched off (*`g++ -O0`*). Reverse-over-reverse mode turns out to be infeasible (runs out of memory) for $n = 4 \cdot 10^4$. Its run time significantly exceeds that of the other two modes for $n = 10^4$ and $n = 2 \cdot 10^4$.*

	`g++ -O0`			`g++ -O3`			$\mathcal{R}$
n	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t2_a1_f	4.0	15.9	62.2	0.7	2.6	10.2	≈ 3.3
a2_t1_f	4.6	18.1	69.6	0.7	2.6	10.4	≈ 3.3
a2_a1_f	12.3	47.4	fail	2.2	8.8	fail	≈ 11

Reverse-over-Reverse Mode

The implementation of reverse-over-reverse mode becomes very tedious, even for simple cases. Its performance falls below that of forward-over-reverse and reverse-over-forward modes because of the repeated data flow reversal. While reverse-over-reverse mode is likely not to be used in practice, its investigation contributes to a better understanding of first- and higher-order adjoint code, which is why we decided to consider it here. In order to obtain an implementation of the second-order adjoint model in reverse-over-reverse mode, reverse mode AD is applied to (2.9) yielding

$$
\begin{aligned}
&\text{for } j = n, \ldots, n+p+m-1\\
&\qquad v_j = \varphi_j(v_i)_{i \prec j}\\
&\text{for } j = n+p+m-1, \ldots, n\\
&\qquad \left(v_{(1)k}\right)_{k \prec j} = \left(v_{(1)k}\right)_{k \prec j} + \left\langle v_{(1)j}, \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k}\right)_{k \prec j} \right\rangle\\
&\text{for } j = n, \ldots, n+p+m-1\\
&\qquad v_{(1,2)j} = v_{(1,2)j} + \left\langle \left(v_{(1,2)k}\right)_{k \prec j}, \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k}\right)_{k \prec j} \right\rangle\\
&\qquad \left(v_{(2)i}\right)_{i \prec j} = \left(v_{(2)i}\right)_{i \prec j} + \left\langle \left(v_{(1,2)k}\right)_{k \prec j}, v_{(1)j}, \left(\frac{\partial^2 \varphi_j(v_i)_{i \prec j}}{\partial v_k \partial v_l}\right)_{\{k,l\} \prec j} \right\rangle\\
&\text{for } j = n+p+m-1, \ldots, n\\
&\qquad \left(v_{(2)i}\right)_{i \prec j} = \left(v_{(2)i}\right)_{i \prec j} + \left\langle v_{(2)j}, \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k}\right)_{k \prec j} \right\rangle.
\end{aligned}
\tag{3.10}
$$

Setting $v_{(1)n+p+i} = y_{(1)i}$ for $i = 0, \ldots, m-1$ and $v_{(1,2)j} = x_{(1,2)j}$ for $j = 0, \ldots, n-1$ yields a projection of the Hessian in directions $\mathbf{y}_{(1)}$ and $\mathbf{x}_{(1,2)}$ if $v_{(2)n+p+i} = y_{(2)i}$ and $v_{(2)j} = x_{(2)j}$ are initialized to zero for $i = 0, \ldots, m-1$ and $i = 0, \ldots, n-1$, respectively. First- and second-order adjoints of all intermediate variables are initialized to zero by default, which is exploited in the following example.

Example 3.22 Once again, we consider $y = f(\mathbf{x}) = \sin(x_0 \cdot x_1)$. In reverse-over-reverse mode, the original SAC and its adjoint are succeeded by the second- and first-order adjoint SACs due to the second application of reverse mode AD.

[SAC]

$$
\begin{aligned}
v_2 &= v_0 \cdot v_1\\
v_3 &= \sin(v_2)
\end{aligned}
$$

[adjoint SAC]

$$
\begin{aligned}
v_{(1)2} &= v_{(1)3} \cdot \cos(v_2)\\
v_{(1)1} &= v_{(1)2} \cdot v_0\\
v_{(1)0} &= v_{(1)2} \cdot v_1
\end{aligned}
$$

[adjoint adjoint SAC]

$$\begin{aligned}
v_{(1,2)2} &= v_{(1,2)0} \cdot v_1 \\
v_{(2)1} &= v_{(1,2)0} \cdot v_{(1)2} \\
v_{(1,2)2} &= v_{(1,2)2} + v_{(1,2)1} \cdot v_0 \\
v_{(2)0} &= v_{(1,2)1} \cdot v_{(1)2} \\
v_{(1,2)3} &= v_{(1,2)2} \cdot \cos(v_2) \\
v_{(2)2} &= -v_{(1,2)2} \cdot v_{(1)3} \cdot \sin(v_2)
\end{aligned}$$

[adjoint SAC]

$$\begin{aligned}
v_{(2)2} &= v_{(2)2} + v_{(2)3} \cdot \cos(v_2) \\
v_{(2)1} &= v_{(2)1} + v_{(2)2} \cdot v_0 \\
v_{(2)0} &= v_{(2)0} + v_{(2)2} \cdot v_1 .
\end{aligned}$$

Incrementations of adjoint intermediates whose initial value is known to be equal to zero have been omitted. It is straightforward to verify that by setting $v_{(1)3} = y_{(1)}$ and $v_{(1,2)i} = x_{(1,2)i}$ for $i = 0, 1$ we obtain the scaled Hessian-vector product

$$\mathbf{x}_{(2)} \equiv \begin{pmatrix} x_{(2)0} \\ x_{(2)1} \end{pmatrix} = y_{(1)} \cdot \nabla^2 f(\mathbf{x}) \cdot \begin{pmatrix} x_{(1,2)0} \\ x_{(1,2)1} \end{pmatrix}$$

if both $\mathbf{x}_{(2)} = (v_{(2)0}, v_{(2)1})^T$ and $y_{(2)} = v_{(2)3}$ are initialized to zero. ∎

Example 3.23 Reverse mode AD is applied to the adjoint version

```
void a1_f(int n, double* x, double* a1_x,
                 double& y, double a1_y) {
  y=0;
  for (int i=0;i<n;i++) y=y+x[i]*x[i];
  required_double_1.push(y);
  y=y*y;

  result_double_1.push(y);

  y=required_double_1.top(); required_double_1.pop();
  a1_y=2*y*a1_y;
  for (int i=n-1;i>=0;i--)
    a1_x[i]=a1_x[i]+2*x[i]*a1_y;
  a1_y=0;

  y=result_double_1.top(); result_double_1.pop();
}
```

of the implementation of (1.2) given in Section 1.1.2. The required data stack is renamed in order to distinguish it from the stack that is generated by the second application of reverse mode. The second-order adjoint code becomes

```
1 void a2_a1_f(int n, double* x, double* a2_x,
2                     double* a1_x, double* a2_a1_x,
3                     double& y, double& a2_y,
4                     double a1_y, double a2_a1_y) {
5   // augmented first-order adjoint
6   y=0;
7   for (int i=0;i<n;i++) y=y+x[i]*x[i];
8   required_double_1.push(y);
9   required_double_2.push(y);
10  y=y*y;
11  result_double_1.push(y);
12  y=required_double_1.top(); required_double_1.pop();
13  required_double_2.push(a1_y); a1_y=2*y*a1_y;
14  for (int i=n-1;i>=0;i--) a1_x[i]=a1_x[i]+2*x[i]*a1_y;
15  required_double_2.push(a1_y); a1_y=0;
16  y=result_double_1.top(); result_double_1.pop();
17
18  // store results of first-order adjoint
19  result_double_2.push(y);
20  for (int i=0;i<n;i++) result_double_2.push(a1_x[i]);
21
22  // adjoint first-order adjoint
23  a2_result_double_1.push(a2_y); a2_y=0;                          // 16
24  a1_y=required_double_2.top(); required_double_2.pop();          // 15
25  a2_a1_y=0;
26  for (int i=0;i<n;i++) {                                         // 14
27    a2_x[i]=a2_x[i]+2*a1_y*a2_a1_x[i];
28    a2_a1_y=a2_a1_y+2*x[i]*a2_a1_x[i];
29  }
30  a1_y=required_double_2.top(); required_double_2.pop();          // 13
31  a2_y=a2_y+2*a1_y*a2_a1_y;
32  a2_a1_y=2*y*a2_a1_y;
33  a2_required_double_1.push(a2_y); a2_y=0;                        // 12
34  a2_y=a2_result_double_1.top(); a2_result_double_1.pop();        // 11
35  y=required_double_2.top(); required_double_2.pop();             // 9
36  a2_y=2*y*a2_y;                                                  // 10
37  a2_y=a2_required_double_1.top();                                // 8
38  a2_required_double_1.pop();
39  for (int i=n-1;i>=0;i--) a2_x[i]=a2_x[i]+2*x[i]*a2_y;           // 7
40  a2_y=0;                                                         // 6
41
42  // restore results of first-order adjoint
43  for (int i=n-1;i>=0;i--)
44    a1_x[i]=result_double_2.top(); result_double_2.pop();
45  y=result_double_2.top(); result_double_2.pop();
46 }
```

It consists of the usual four parts, namely, the augmented forward section (the first-order adjoint code augmented with the storage of required overwritten values; lines 6–16), the storage of the results (of the first-order adjoint code; lines 19–20), the reverse section (adjoint versions of all statements in the first-order adjoint code augmented with the recovery of required values that are stored in the augmented forward section; lines 23–40), and the recovery of the results (lines 43–45). Comments link adjoint statements with their counterparts in the augmented forward section; for example, line 36 holds the adjoint version of the assignment in line 10.

The entire data segment of the first-order adjoint code is duplicated according to Adjoint Code Generation Rule 1 including the required data and result checkpoint stacks, yielding a2_required_double_1 and a2_result_double_1. The treatment of stack accesses exploits the fact that all stack values are both written and read exactly once. Hence, the adjoint version of required_double_1.push(y) in line 8 yields reading a2_y from a2_required_double_1 in line 37 followed by removing in line 38 the top of the stack. No further augmentation is necessary as no required value is overwritten. Lines 11 and 34 form an analogous pair. All remaining statements are the result of the straight application of the Adjoint Code Generation Rules to the first-order adjoint code. For example, a required value of a1_y is overwritten in line 13 and hence is stored on the required_double_2 stack. The corresponding adjoint assignments in lines 31 and 32 are preceded by the recovery of the required value and its removal in line 30 from required_double_2.

The following driver computes the Hessian of (1.2) at point $x_i = 1$, $i = 0,\ldots,3$, set in line 4.

```
1 int main() {
2   const int n=4;
3   double x[n],y,a2_x[n],a2_y,a1_x[n],a1_y,a2_a1_x[n],a2_a1_y;
4   for (int i=0;i<n;i++) { x[i]=1; a2_a1_x[i]=0; }
5   a2_y=0;
6   for (int i=0;i<n;i++) {
7     for (int j=0;j<n;j++) a2_x[j]=0;
8     a1_y=1;
9     a2_a1_x[i]=1;
10    a2_a1_f(n,x,a2_x,a1_x,a2_a1_x,y,a2_y,a1_y,a2_a1_y);
11    for (int j=0;j<=i;j++)
12      cout << "H[" << i << "][" << j << "]=" << a2_x[j] << endl;
13    a2_a1_x[i]=0;
14  }
15  return 0;
16 }
```

It contains in line 6 a loop over the Cartesian basis vectors in $\mathbb{R}^4$ that are assigned to the initially zero (see line 4) vector a2_a1_x. The first-order adjoint a1_y of the original output is set to one in line 8. The corresponding a2_a1_x entries can be reset to zero individually in line 13 as a2_a1_x is, according to (3.7), left unchanged by a2_a1_f. Initialization of a2_y=0 in line 5 is crucial for avoiding the addition of first derivative information to a2_x. According to (3.7), a2_y is kept equal to zero by the repeated calls of a2_a1_f. The columns of the Hessian are returned in a2_x, and they are sent to the standard output in line 12. Only a2_x needs to be reset to zero in line 7 prior to each call of a2_a1_f as the adjoint code is generated in incremental reverse mode.

Again, n = 4 evaluations of the second-order adjoint code are required to compute all entries of the Hessian. A single Hessian-vector product is obtained at a constant factor of the cost of evaluating the original code in Section 1.1.2. Typically, second-order adjoint code generated in reverse-over-reverse mode is significantly less efficient than the other two variants of implementing the second-order adjoint model, as illustrated by the run-time measurements in Table 3.2. ∎

The derivative code compiler `dcc` supports the generation of second-order adjoint code in all three modes. Thus, it may contribute to a better understanding of the principles that AD is based on. Refer to Chapter 5 for further details.

3.3.2 Overloading

dco supports both forward-over-reverse and reverse-over-forward modes. Reverse-over-reverse mode has been omitted due to its obvious drawbacks as a result of the repeated data flow reversal. The given tangent-linear or adjoint code is treated analogous to any other target code. Active floating-point variables are redeclared as dco_t1s_type in forward mode or as dco_a1s_type in reverse mode. A tape is generated and interpreted in reverse mode. The use of the corresponding second derivative code is very similar to what was discussed in the previous section. Qualitatively, the run time behavior matches that of second derivative code generated by source transformation.

Forward-over-Reverse Mode

The second-order adjoint model can be implemented by changing the types of all floating-point members in **class** dco_a1s_tape_entry and **class** dco_a1s_type from Section 2.2.2 to dco_t1s_type as defined in Section 2.1.2 yielding the data type **class** dco_t2s_a1s_tape_entry and **class** dco_t2s_a1s_type shown in the following code listing. See lines 4 and 11 for the respective type changes.

```
1  class dco_t2s_a1s_tape_entry {
2    public:
3      int oc, arg1, arg2;
4      dco_t1s_type v, a;
5      ...
6  };
7
8  class dco_t2s_a1s_type {
9    public:
10     int va;
11     dco_t1s_type v;
12     ...
13 };
```

This approach yields an implementation of the second-order adjoint model in forward-over-reverse mode. The driver program in Listing 3.2 uses this implementation of the second-order adjoint model to compute the Hessian of (1.2) for $n = 4$, set in line 5, at the point $x_i = 1$ for $i = 0,\ldots,3$.

The second-order adjoint data type dco_t2s_a1s_type is declared in the header file dco_t2s_a1s_type.hpp. Its declaration is included in the driver program in line 3, and it is used to activate the target code (the function f in lines 7–11) by changing the type of all floating-point variables from **double** to dco_t2s_a1s_type. A tape of size DCO_T2S_A1S_TAPE_SIZE (to be replaced with an integer value by the C preprocessor) is allocated statically in dco_t2s_a1s_type.cpp and is later linked to the object code of the driver program.

Both the taping and the interpretation of the tape are performed in tangent-linear mode. Hence, x[i].v.t $\equiv x_i^{(1)}$ and dco_t2s_a1s_tape[x[i].va].v.t $\equiv x_i^{(1)}$ need to be initialized simultaneously as shown in line 21. A new tape is generated for each column of the Hessian. Therefore, the virtual address counter dco_t2s_a1s_vac as well as all adjoint tape entries are reset to zero prior to each iteration of the loop in line 18 by calling in line 19 the function

Listing 3.2. *Driver for forward-over-reverse mode by overloading.*

```
1 #include <iostream>
2 using namespace std;
3 #include "dco_t2s_a1s_type.hpp"
4
5 const int n=4;
6
7 void f(dco_t2s_a1s_type *x, dco_t2s_a1s_type &y) {
8   y=0;
9   for (int i=0;i<n;i++) y=y+x[i]*x[i];
10  y=y*y;
11 }
12
13 extern dco_t2s_a1s_tape_entry
14   dco_t2s_a1s_tape[DCO_T2S_A1S_TAPE_SIZE];
15
16 int main() {
17   dco_t2s_a1s_type x[n],y;
18   for (int i=0;i<n;i++) {
19     dco_t2s_a1s_reset_tape();
20     for (int j=0;j<n;j++) x[j]=1;
21     x[i].v.t=dco_t2s_a1s_tape[x[i].va].v.t=1;
22     f(x,y);
23     dco_t2s_a1s_tape[y.va].a.v=1;
24     dco_t2s_a1s_interpret_tape();
25     cout << "H[" << i << "][" << i << "]="
26          << dco_t2s_a1s_tape[x[i].va].a.t << endl;
27   }
28   return 0;
29 }
```

dco_t2s_a1s_reset_tape . The independent variables are registered during their initialization in line 20 in the tape by calling an appropriately overloaded assignment operator (see code listing further below). The call in line 22 of the overloaded target code is followed by the initialization in line 23 of the adjoint of the dependent output. Here, $y_{(1)} \equiv$ dco_t2s_a1s_tape [y.va].a.v is set to one in order to compute unscaled products of the Hessian with the vectors $\mathbf{x}^{(2)}$ set in line 21. Interpretation of the tape in line 24 is followed by the retrieval of the diagonal entries of the Hessian in lines 25–27. The relevant entries of the ith column are returned in $\mathbf{x}^{(2)}_{(1)j} \equiv$ dco_t2s_a1s_tape[x[j].va].a.t for $j = 0,\ldots,i$.

The tape is very similar to that generated in first-order adjoint mode discussed in Section 2.2.2. A tape entry is a quintuple consisting of an operation code (oc in the definition of the **class** dco_t2s_a1s_tape_entry), the virtual addresses of at most two arguments (arg1 and arg2), the intermediate function value (v), and the associated adjoint (a). Both the function value and the adjoint are of type dco_t1s_type making them pairs that contain a value component (v.v $\equiv v$ and a.v $\equiv v_{(1)}$) and the corresponding directional derivatives (v.t $\equiv v^{(2)}$ and a.t $\equiv v^{(2)}_{(1)}$). We consider the state of the tape immediately after its generation (after line 22 in Listing 3.2; see Figure 3.9 (a)) and after its interpretation (after line 24 in Listing 3.2; see Figure 3.9 (b)) for the computation of the third column of the Hessian, that is, for $i = 2$.

Tape:

```
 0: [ 0,-1,-1,   (1.,0.), (0.,0.) ]
 1: [ 1, 0,-1,   (1.,0.), (0.,0.) ]
 2: [ 0,-1,-1,   (1.,0.), (0.,0.) ]
 3: [ 1, 2,-1,   (1.,0.), (0.,0.) ]
 4: [ 0,-1,-1,   (1.,0.), (0.,0.) ]
 5: [ 1, 4,-1,   (1.,1.), (0.,0.) ]
 6: [ 0,-1,-1,   (1.,0.), (0.,0.) ]
 7: [ 1, 6,-1,   (1.,0.), (0.,0.) ]
 8: [ 0,-1,-1,   (0.,0.), (0.,0.) ]
 9: [ 1, 8,-1,   (0.,0.), (0.,0.) ]
10: [ 4, 1, 1,   (1.,0.), (0.,0.) ]
11: [ 2, 9,10,   (1.,0.), (0.,0.) ]
12: [ 1,11,-1,   (1.,0.), (0.,0.) ]
13: [ 4, 3, 3,   (1.,0.), (0.,0.) ]
14: [ 2,12,13,   (2.,0.), (0.,0.) ]
15: [ 1,14,-1,   (2.,0.), (0.,0.) ]
...
21: [ 1,20,-1,   (4.,2.), (0.,0.) ]
22: [ 4,21,21, (16.,16.), (0.,0.) ]
23: [ 1,22,-1, (16.,16.), (0.,0.) ]
```

(a)

Interpreted Tape:

```
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (1.,0.), (16.,24.) ]
[ ...   (1.,1.), (16.,24.) ]
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (1.,0.),  (16.,8.) ]
[ ...   (0.,0.),   (8.,4.) ]
[ ...   (0.,0.),   (8.,4.) ]
[ ...   (1.,0.),   (8.,4.) ]
[ ...   (1.,0.),   (8.,4.) ]
[ ...   (1.,0.),   (8.,4.) ]
[ ...   (1.,0.),   (8.,4.) ]
[ ...   (2.,0.),   (8.,4.) ]
[ ...   (2.,0.),   (8.,4.) ]
[ ...   (4.,2.),   (8.,4.) ]
[ ... (16.,16.),   (1.,0.) ]
[ ... (16.,16.),   (1.,0.) ]
```

(b)

Figure 3.9. *dco_t2s_a1s_tape for the computation of third column of the Hessian. The five columns show for each tape entry the operation code, the virtual addresses of the (up to two) arguments, the tangent-linear function value, and the tangent-linear adjoint value. Tangent-linear quantities are pairs that consist of the original value and the corresponding directional derivative.*

The tape entries 0, 2, 4, and 6 represent the constants on the right-hand side of the assignment in line 20 of Listing 3.2 that are converted into variables of type dco_t2s_a1s_type by the constructor

```
1 dco_t2s_a1s_type::dco_t2s_a1s_type(const double& x): v(x) {
2   dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_CONST;
3   dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=x;
4   va=dco_t2s_a1s_vac++;
5 };
```

Their values are assigned to x[j], yielding tape entries 1, 3, 5, and 7 as a result of calling the overloaded assignment operator

```
1 dco_t2s_a1s_type&
2 dco_t2s_a1s_type::operator=(const dco_t2s_a1s_type& x) {
3   if (this==&x) return *this;
4   dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_ASG;
5   dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=v=x.v;
6   dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x.va;
7   va=dco_t2s_a1s_vac++;
8   return *this;
9 }
```

All tangent-linear components $v^{(2)}$ are initialized to zero by the assignments in lines 3 and 5 of the constructor and the assignment operator, respectively. Note that these assignments are overloaded for variables of type dco_t1s_type. To compute the third column of the Hessian, $x_2^{(2)} \equiv$ x[2].v.t $\equiv$ dco_t2s_a1s_tape[x[2].va].v.t is set to one in line 21 of Listing 3.2. The fifth components of all tape entries are initialized to $(v_{(1)}, v_{(1)}^{(2)}) = (0,0)$ by the function dco_t2s_a1s_reset_tape that is called in line 19.

The tape entries 8 and 9 correspond to line 8 in Listing 3.2. Four evaluations of the assignment in line 9 yield the twelve tape entries 10–21. For example, tape entry 10 stands for the square operation applied to x[0] (tape entry 1) followed by tape entry 11 that represents the addition of the result to y (tape entry 9). A new live instance of y is generated by the subsequent assignment (tape entry 12). This new instance of y is incremented during the next loop iteration (see tape entry 14) and so forth. When processing line 10 of Listing 3.2, the square of the live instance of y at the end of the loop (tape entry 21) is squared (tape entry 22) and the result is assigned to the output y of the subroutine f (tape entry 23). According to (3.5), we obtain

$$\begin{aligned} v_{(2)22} &= 2 \cdot v_{21} \cdot v_{21}^{(2)} = 2 \cdot 4 \cdot 2 = 16, \\ v_{22} &= v_{21}^2 = 4^2 = 16. \end{aligned}$$

The interpretation of the tape is preceded by the initialization of the adjoint output $y_{(1)} \equiv$ dco_t2s_a1s_tape[y.va].a.v yielding a modification of the first-order adjoint component of tape entry 23 in Figure 3.9 (b). Initialization with one results in a first-order adjoint accumulation of the gradient. Overloading of the interpreter in tangent-linear mode adds the propagation of tangent-linear projections of the Hessian in direction $x_i^{(2)} \equiv$ dco_t2s_a1s_type [x[i].va].v.t for i $= 0, \ldots, 3$. The first- and second-order adjoints are copied into tape entry 22 that represents the result of the product y$*$y in line 10 of Listing 3.2 without modification. According to (3.5), the interpretation of tape entry 22 yields

$$\begin{aligned} v_{(1)21} &= v_{(1)21} + v_{(1)22} \cdot 2 \cdot v_{21} = 0 + 1 \cdot 2 \cdot 4 = 8, \\ v_{(1)21}^{(2)} &= v_{(1)21}^{(2)} + v_{(1)22} \cdot 2 \cdot v_{21}^{(2)} = 0 + 1 \cdot 2 \cdot 2 = 4. \end{aligned}$$

Tape entries 21–10 are processed in a similar manner yielding the third column of the Hessian in tape entries 1, 3, 5, and 7.

Let both **class** dco_t2s_a1s_type and **class** dco_t2s_a1s_tape_entry be defined in the file `dco_t2s_a1s.cpp` with an interface provided in the file `dco_t2s_a1s.hpp`, and let the driver program be stored as `main.cpp`. If `dco_t1s.hpp` and `dco_t1s.cpp` are located in the same directory, then the build process is similar to that described in Section 3.2.2.

Reverse-over-Forward Mode

An implementation of the second-order adjoint model in reverse-over-forward mode is obtained by changing the types of all floating-point members in **class** dco_t1s_type from Section 2.1.2 to dco_a1s_type as defined in Section 2.2.2 yielding

```
1  class dco_a2s_t1s_type {
2    public :
```

```
3       dco_a1s_type v, t;
4     ...
5   };
```

The driver program performs the same task as that in Listing 3.2.

A shortened version of the tape that is generated for the computation of the second column of the Hessian is shown in Figure 3.10 (a) (tape after recording) and Figure 3.10 (b) (tape after interpretation). The increase in the length of the tape by a factor of approximately four is due to the entire tangent-linear code being recorded. Initialization of the independent variables in line 20 of Listing 3.3 yields four tape entries, respectively. For example, tape entries 4 and 5 represent a call of the constructor

```
dco_a2s_t1s_type::dco_a2s_t1s_type(const double& x):t(0) v(x) {}
```

that converts the constants on the right-hand side of the assignment x[i]=1 to variables of type dco_a2s_t1s_type. One tape entry is generated for the value (v; tape entry 5) and for the

Listing 3.3. *Driver for reverse-over-forward mode by overloading.*

```
1 #include <iostream>
2 using namespace std;
3 #include "dco_a2s_t1s_type.hpp"
4
5 const int n=4;
6
7 void f(dco_a2s_t1s_type *x, dco_a2s_t1s_type &y) {
8   y=0;
9   for (int i=0;i<n;i++) y=y+x[i]*x[i];
10  y=y*y;
11 }
12
13 extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];
14
15 int main() {
16
17   dco_a2s_t1s_type x[n],y;
18   for (int i=0;i<n;i++) {
19     dco_a1s_reset_tape();
20     for (int j=0;j<n;j++) x[j]=1;
21     x[i].t.v=dco_a1s_tape[x[i].t.va].v=1;
22     f(x,y);
23     dco_a1s_tape[y.t.va].a=1;
24     dco_a1s_interpret_tape();
25     cout << "H[" << i << "][" << i << "]="
26          << dco_a1s_tape[x[i].v.va].a << endl;
27   }
28   return 0;
29 }
```

Tape:

```
...
 4: [ 0, -1, -1,  0., 0. ]
 5: [ 0, -1, -1,  1., 0. ]
 6: [ 1,  4, -1,  1., 0. ]
 7: [ 1,  5, -1,  1., 0. ]
....
38: [ 4,  7,  6,  1., 0. ]
39: [ 4,  6,  7,  1., 0. ]
40: [ 2, 39, 38,  2., 0. ]
41: [ 1, 40, -1,  2., 0. ]
42: [ 4,  7,  7,  1., 0. ]
43: [ 1, 42, -1,  1., 0. ]
...
46: [ 2, 34, 41,  2., 0. ]
47: [ 1, 46, -1,  2., 0. ]
48: [ 2, 35, 43,  2., 0. ]
49: [ 1, 48, -1,  2., 0. ]
...
86: [ 4, 83, 82,  8., 0. ]
87: [ 4, 82, 83,  8., 0. ]
88: [ 2, 87, 86, 16., 0. ]
89: [ 1, 88, -1, 16., 0. ]
90: [ 4, 83, 83, 16., 0. ]
91: [ 1, 90, -1, 16., 0. ]
92: [ 1, 89, -1, 16., 0. ]
93: [ 1, 91, -1, 16., 0. ]
```

(a)

Interpreted Tape:

```
...
 4: [ 0, -1, -1,  0., 16. ]
 5: [ 0, -1, -1,  1., 24. ]
 6: [ 1,  4, -1,  1., 16. ]
 7: [ 1,  5, -1,  1., 24. ]
...
38: [ 4,  7,  6,  1.,  8. ]
39: [ 4,  6,  7,  1.,  8. ]
40: [ 2, 39, 38,  2.,  8. ]
41: [ 1, 40, -1,  2.,  8. ]
42: [ 4,  7,  7,  1.,  4. ]
43: [ 1, 42, -1,  1.,  4. ]
...
82: [ 1, 79, -1,  2.,  8. ]
83: [ 1, 81, -1,  4.,  4. ]
...
86: [ 4, 83, 82,  8.,  1. ]
87: [ 4, 82, 83,  8.,  1. ]
88: [ 2, 87, 86, 16.,  1. ]
89: [ 1, 88, -1, 16.,  1. ]
90: [ 4, 83, 83, 16.,  0. ]
91: [ 1, 90, -1, 16.,  0. ]
92: [ 1, 89, -1, 16.,  1. ]
93: [ 1, 91, -1, 16.,  0. ]
```

(b)

Figure 3.10. *dco_a2s_t1s_tape for the computation of the second column of the Hessian; the five columns show for each tape entry the operation code, the virtual addresses of the (up to two) arguments, the function value, and the adjoint value. First- and second-order adjoints are propagated during the interpretation of the tape of the underlying first-order tangent-linear code.*

directional derivative component (t; tape entry 4), respectively. The overloaded assignment operator

```
1 dco_a2s_t1s_type&
2 dco_a2s_t1s_type::operator=(const dco_a2s_t1s_type& x) {
3     if (this==&x) return *this;
4     t=x.t; v=x.v;
5     return *this;
6   }
```

adds tape entries 6 and 7 that represent the two assignments in line 4. According to (3.6), we obtain $v_6 \equiv v_7^{(1)} = 1 \cdot v_5^{(1)} \equiv v_4 = 0$, which does not match the value in Figure 3.10 (a).

The displayed value is due to line 21 in Listing 3.3, where $x_1^{(2)} \equiv v_7^{(1)} \equiv v_6$ is set equal to one explicitly.

A total of six tape entries are generated for products as the original multiplication is augmented with the product rule in the tangent-linear code. For example, the product performed in line 9 of Listing 3.3 during the second iteration of the enclosing loop yields tape entries 38, 39, 40, and 42. The respective assignments to temporary variables that are performed by the overloaded multiplication operator

```
1 dco_a2s_t1s_type operator*(const dco_a2s_t1s_type& x1,
2                            const dco_a2s_t1s_type& x2) {
3   dco_a2s_t1s_type tmp;
4   tmp.t=x1.t*x2.v+x1.v*x2.t;
5   tmp.v=x1.v*x2.v;
6   return tmp;
7 }
```

are represented by tape entries 41 and 43. According to (3.6), the two assignments in lines 4 and 5 of the code for **operator**$*$ result in the following computation:

$$\begin{aligned}
v_{38} &= v_7 \cdot v_7^{(1)} = v_7 \cdot v_6 = 1 \cdot 1 = 1 \\
v_{39} &= v_7^{(1)} \cdot v_7 = v_6 \cdot v_7 = 1 \cdot 1 = 1 \\
v_{40} &= v_{39} + v_{38} = 1 + 1 = 2 \\
v_{43}^{(1)} \equiv v_{41} &= v_{40} = 2 \\
v_{42} &= v_7 \cdot v_7 = 1 \cdot 1 = 1 \\
v_{43} &= v_{42} = 1.
\end{aligned}$$

Additions yield four tape entries; for example, 46–49, where $v_{47} \equiv v_{49}^{(1)}$. The product in line 10 of Listing 3.3 followed by the assignment to the output y of the subroutine f is represented by the last eight tape entries 86–93, where $v_{89} \equiv v_{91}^{(1)}$ and $v_{92} \equiv v_{93}^{(1)}$. With $v_{83} = 4$ and $v_{82} \equiv v_{83}^{(1)} = 2$, the following steps are performed according to (3.6):

$$\begin{aligned}
v_{86} &= v_{83} \cdot v_{83}^{(1)} = v_{83} \cdot v_{82} = 4 \cdot 2 = 8 \\
v_{87} &= v_{83}^{(1)} \cdot v_{83} = v_{82} \cdot v_{83} = 2 \cdot 4 = 8 \\
v_{88} &= v_{87} + v_{86} = 8 + 8 = 16 \\
v_{91}^{(1)} \equiv v_{89} &= v_{88} = 16 \\
v_{90} &= v_{83} \cdot v_{83} = 4 \cdot 4 = 16 \\
v_{91} &= v_{90} = 16 \\
y^{(1)} \equiv v_{92} &= v_{91}^{(1)} = 16 \\
y \equiv v_{93} &= v_{91} = 16.
\end{aligned}$$

The interpretation of the tape is preceded by the initialization of the second-order adjoint component $y_{(2)}^{(1)}$ of the output yielding the modified tape entry 92 in Figure 3.10 (b). Its first-order adjoint (fifth component of tape entry 93) remains equal to zero. After the

Table 3.3. *Run times for second-order adjoint code by overloading (in seconds). In order to determine the relative computational complexity $\mathcal{R}$ of the derivative code, n function evaluations are compared with n evaluations of the second-order adjoint code required for a full Hessian accumulation. We observe a constantly growing factor of at least* 39 *when comparing the run time of a single run of the second-order adjoint that was code generated in forward-over-reverse mode with that of an original function evaluation in the right-most column. This factor is approximately double the factor that was observed for first-order adjoint code generated by overloading. The impact of compiler optimization is even less significant for second-order adjoint code generated in reverse-over-forward mode.*

	g++ -O0			g++ -O3			$\mathcal{R}$
n	10^4	$2\cdot10^4$	$4\cdot10^4$	10^4	$2\cdot10^4$	$4\cdot10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t2_a1_f	87.8	360.6	1435.0	31.4	164.1	772.6	> 39
a2_t1_f	147.6	562.6	2297.9	101.8	474.0	1886.3	> 127

interpretation of the tape, all first-order adjoint components of tape entries that represent variables in the original code contain second-order adjoint projections of the local Hessians; for example, tape entries 82 and 83 represent the right-hand side instance of y in line 10 of Listing 3.3. The local Hessian is equal to the constant scalar 2. According to (3.6), we get

$$\begin{aligned} v_{(2)83} &= v_{(2)90}\cdot 2\cdot v_{83} + v^{(1)}_{(2)90}\cdot 2\cdot v^{(1)}_{83} \\ &= v_{(2)90}\cdot 2\cdot v_{83} + v_{(2)88}\cdot 2\cdot v_{82} = 0\cdot 2\cdot 4 + 1\cdot 2\cdot 2 = 4 \\ v_{(2)82} \equiv v^{(1)}_{(2)83} &= v^{(1)}_{(2)83} + v^{(1)}_{(2)90}\cdot 2\cdot v_{83} \\ &= v_{(2)82} + v_{(2)88}\cdot 2\cdot v_{83} = 0 + 1\cdot 2\cdot 4 = 8. \end{aligned}$$

Similarly,

$$\begin{aligned} v_{(2)7} &= v_{(2)42}\cdot 2\cdot v_7 + v^{(1)}_{(2)42}\cdot 2\cdot v^{(1)}_7 \\ &= v_{(2)42}\cdot 2\cdot v_7 + v_{(2)40}\cdot 2\cdot v_6 = 4\cdot 2\cdot 1 + 8\cdot 2\cdot 1 = 24 \\ v_{(2)6} \equiv v^{(1)}_{(2)7} &= v^{(1)}_{(2)7} + v^{(1)}_{(2)42}\cdot 2\cdot v_7 \\ &= v_{(2)6} + v_{(2)40}\cdot 2\cdot v_7 = 0 + 8\cdot 2\cdot 1 = 16. \end{aligned}$$

The second diagonal entry of the Hessian (24) is accumulated in the adjoint component of tape entry 7. Tape entry 6 contains the second gradient entry (16) in its adjoint component. According to Theorem 3.16, the same value is contained in $y^{(1)} \equiv v_{92}$. Refer to Table 3.3 for run time measurements.

Reverse-over-Reverse Mode

The discussion of an implementation of second-order adjoint code in reverse-over-reverse mode by overloading is omitted; this approach is irrelevant in practice. The repeated reversal

of the data flow yields an excessive memory requirement due to recursive taping as well as an increased computational cost caused by the complexity of the interpretation procedure.

3.3.3 Compression of Sparse Hessians

We recall the basics of compression techniques for second derivative tensors based on second-order adjoint projections as in [17]. Scalar functions $F : \mathbb{R}^n \to \mathbb{R}^m$, where $m = 1$ are of particular interest in the context of nonlinear programming. A corresponding second-order adjoint code computes products of the Hessian with a vector as $\mathbf{y}_{(1)} \cdot \nabla^2 F(\mathbf{x}) \cdot \mathbf{x}^{(2)}$, where $\mathbf{y}_{(1)} = 1$ and $\mathbf{x}^{(2)} \in \mathbb{R}^n$. Tangent-linear compression techniques as described in Section 2.1.3 can be applied. Moreover, symmetry should be exploited potentially yielding better compression rates. For vector functions ($m > 1$), the sparsity of the Hessian is closely related to that of the Jacobian. Hence, a combination of tangent-linear and adjoint compression is likely to give the best compression rate as described in [36].

Example 3.24 Let

$$\nabla^2 F = \begin{pmatrix} h_{0,0} & 0 & h_{0,2} \\ 0 & h_{1,1} & h_{1,2} \\ h_{0,2} & h_{1,2} & h_{2,2} \end{pmatrix}.$$

The dense third row appears to make unidirectional compression as in Section 2.1.3 inapplicable. However, symmetry of the Hessian implies that only one instance of $h_{0,2}$ and $h_{1,2}$ needs to be recovered, respectively. Consequently, the following compression can be applied:

$$\nabla^2 F \cdot S_t = \begin{pmatrix} h_{0,0} & 0 & h_{0,2} \\ 0 & h_{1,1} & h_{1,2} \\ h_{0,2} & h_{1,2} & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} h_{0,0} & h_{0,2} \\ h_{1,1} & h_{1,2} \\ h_{0,2}+h_{1,2} & h_{2,2} \end{pmatrix}.$$

All five distinct nonzero entries of the Hessian can be recovered by direct substitution. ∎

Definition 3.25. *Let $A \in \mathbb{R}^{m \times n \times n}$ be a symmetric 3-tensor, $S_a = (s^a_{j,i}) \in \mathbb{R}^{m \times l_a}$, and $S_t = (s^t_{j,i}) \in \mathbb{R}^{n \times l_t}$. Then, the* compressed Hessian *$B \equiv (b_{k,j,i}) = \langle S_a, A, S_t \rangle \in \mathbb{R}^{l_a \times l_t \times l_t}$ is defined as*

$$b_{k,j,*} = \langle \langle s^a_{k,*}, A \rangle, s^t_{*,j} \rangle$$

for $k = 1, \ldots, l_a$ and $j = 1, \ldots, l_t$.

When applying compression techniques in second-order adjoint mode, we aim to find seed matrices S_a and S_t with minimal numbers of columns l_a and l_t such that

$$B = \langle S_a, A, S_t \rangle \in \mathbb{R}^{l_a \times l_t}, \tag{3.11}$$

where $A \equiv \nabla^2 F(\mathbf{x})$, $S_a \in \{0,1\}^{m \times l_a}$, $S_t \in \{0,1\}^{n \times l_t}$, and $\forall a_{i,j} \neq 0, i \leq j \; \exists b_{l,i} \in B : a_{i,j} = b_{l,i}$. All nonzero entries of the lower (resp., upper) triangular submatrix of the Hessian need to be present in the compressed Hessian B. Harvesting solves the system in (3.11) by substitution if direct methods are applied. Again, indirect methods may result in a better compression rate. Refer to [30] for details on the combinatorial problem of minimizing l_t and l_a by graph coloring algorithms.

Example 3.26 Let the second-order adjoint model of the implementation of the SFI problem (see Example 1.2) be implemented as a subroutine

```
void t2_a1_f(int s, double** y, double** t2_y,
                    double** a1_y, double** t2_a1_y,
                    double& l, double& t2_l,
                    double& a1_l, double& t2_a1_l,
                    double** r, double** t2_r,
                    double** a1_r, double** t2_a1_r);
```

The Hessian tensor of the residual is very sparse. Its computation is complicated by the fact that we are actually dealing with a 6-tensor instead of a 3-tensor, because both y and r are implemented as matrices. Thus, the Hessian for $s = 3$ determined in Example 3.2 becomes

$$\nabla^2 r_{i_1,j_1,i_2,j_2,k_1,k_2} = \begin{cases} -h^2 \cdot \lambda \cdot e^{y_{1,1}} & \text{if } i_1 = j_1 = k_1 = 1 \text{ and } i_2 = j_2 = k_2 = 1 \\ -h^2 \cdot \lambda \cdot e^{y_{1,2}} & \text{if } i_1 = j_1 = k_1 = 1 \text{ and } i_2 = j_2 = k_2 = 2 \\ -h^2 \cdot \lambda \cdot e^{y_{2,1}} & \text{if } i_1 = j_1 = k_1 = 2 \text{ and } i_2 = j_2 = k_2 = 1 \\ -h^2 \cdot \lambda \cdot e^{y_{2,2}} & \text{if } i_1 = j_1 = k_1 = 2 \text{ and } i_2 = j_2 = k_2 = 2 \\ 0 & \text{otherwise.} \end{cases}$$

Knowledge about this sparsity pattern can be exploited by seeding and harvesting the second-order adjoint routine as shown in the following driver fragment:

```
...
  for (int i=1;i<s;i++)
    for (int j=1;j<s;j++)
      a1_r[i][j]=t2_y[i][j]=1;
  t2_a1_f(1,s,y,t2_y,a1_y,t2_a1_y,
          lambda,t2_lambda,a1_lambda,t2_a1_lambda,
          r,t2_r,a1_r,t2_a1_r);
...
```

The nonzero entries of the Hessian are returned in t2_a1_y whose entries are assumed to be initialized to zero prior to the single run of t2_a1_f.

In general, the adjoint seed matrix used for the first-order adjoint can also be applied to compress the Hessian tensor. Linearities in the underlying function yield constants in the Jacobian and further zero entries in the Hessian as in the given example. While this way of exploiting sparsity appears to yield optimal computational complexity, there is still room for improvement. The preferred approach to the computation of the Hessian of the SFI problem uses an implementation of the second-order tangent-linear model

```
void t2_t1_f(int s, double** y, double** t2_y,
                    double** t1_y, double** t2_t1_y,
                    double& l, double& t2_l,
                    double& t1_l, double& t2_t1_l,
                    double** r, double** t2_r,
                    double** t1_r, double** t2_t1_r)
```

as follows:

```
...
  for (int i=1;i<s;i++)
    for (int j=1;j<s;j++)
      t1_y[i][j]=t2_y[i][j]=1;
  t2_t1_f(s,y,t2_y,t1_y,t2_t1_y,
          lambda,t2_lambda,t1_lambda,t2_t1_lambda,
          r,t2_r,t1_r,t2_t1_r);
...
```

All nonzero entries of the Hessian are returned in t2_t1_r. A single call of t2_t1_f is performed. The generation and evaluation of the computationally more challenging adjoint code can be avoided due to the strict symmetry of the Hessian tensor of the SFI problem under arbitrary projections. ∎

3.4 Higher Derivative Code

The application of forward or reverse mode AD to any of the second derivative models yields third derivative information and so forth. In order to formalize this repeated reapplication of AD, we need to generalize the tensor notation introduced in Section 3.1.

Definition 3.27. *Consider a symmetric* $(p+1)$*-tensor* $T \in \mathbb{R}^{m \times n^p}$*, where*

$$T = (t_{j,i_1,\ldots,i_p})_{i_k=0,\ldots,n-1 \text{ for } k=1,\ldots,p}^{j=0,\ldots,m-1}$$

and $t_{j,i_1,\ldots,i_p} = t_{j,\pi(i_1,\ldots,i_p)}$ *for any permutation* π *of* $i_1,\ldots,i_p$*. A* first-order tangent-linear projection *of* T *in direction* $\mathbf{v} \in \mathbb{R}^n$ *is defined as*

$$\dot{T} \equiv \langle T, \mathbf{v} \rangle \in \mathbb{R}^{m \times n^{p-1}},$$

with $\dot{T} = (\dot{t}_{j,i_1,\ldots,i_{p-1}})_{i_k=0,\ldots,n-1 \text{ for } k=1,\ldots,p-1}^{j=0,\ldots,m-1}$ *and*

$$\dot{t}_{j,i_1,\ldots,i_{p-1}} = \langle t_{j,i_1,\ldots,i_{p-1},*}, \mathbf{v} \rangle \equiv \sum_{l=0}^{n-1} t_{j,i_1,\ldots,i_{p-1},l} \cdot v_l$$

for $i_k = 0,\ldots,n-1$ $(k = 1,\ldots,p-1)$ *and* $j = 0,\ldots,m-1$.

Higher-order tangent-linear projections *are defined recursively as*

$$\begin{aligned}
\langle T, \mathbf{v}_1, \mathbf{v}_2 \rangle &\equiv \langle \langle T, \mathbf{v}_1 \rangle, \mathbf{v}_2 \rangle \in \mathbb{R}^{m \times n^{p-2}} \\
\langle T, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3 \rangle &\equiv \langle \langle \langle T, \mathbf{v}_1 \rangle, \mathbf{v}_2 \rangle, \mathbf{v}_3 \rangle \in \mathbb{R}^{m \times n^{p-3}} \\
&\vdots \\
\langle T, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p \rangle &\equiv \langle \langle \cdots \langle \langle T, \mathbf{v}_1 \rangle, \mathbf{v}_2 \rangle, \ldots \rangle, \mathbf{v}_p \rangle \in \mathbb{R}^m.
\end{aligned}$$

First- and higher-order adjoint projections in directions in $\mathbb{R}^n$ are defined as the corresponding tangent-linear projections. Such projections appear in the context of higher-order adjoint code similar to the case considered in Section C.3.3 in the appendix.

Lemma 3.28. *Consider $T \in \mathbb{R}^{m \times n^p}$ as defined in Definition* 3.27 *and $k \leq p$. Then,*

$$\langle T, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \rangle = \langle T, \pi(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k) \rangle$$

for any permutation π of $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ and where $\mathbf{v}_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$.

Proof. The lemma follows immediately from the symmetry within the tensors as defined in Definition 3.27. □

Definition 3.29. *Consider a symmetric $(p+1)$-tensor $T \in \mathbb{R}^{m \times n^p}$ as in Definition* 3.27. *A* first-order adjoint projection *of T in direction $\mathbf{u} \in \mathbb{R}^m$ is defined as*

$$\bar{T} \equiv \langle \mathbf{u}, T \rangle \in \mathbb{R}^{n^p},$$

with $\bar{T} = (\bar{t}_{i_1,\ldots,i_p})_{i_k=0,\ldots,n-1 \text{ for } k=1,\ldots,p}$ and

$$\bar{t}_{i_1,\ldots,i_p} = \langle \mathbf{u}, t_{*,i_1,\ldots,i_p} \rangle \equiv \sum_{l=0}^{m-1} u_l \cdot t_{l,i_1,\ldots,i_p}$$

for $i_k = 0, \ldots, n-1$ $(k = 1, \ldots, p)$.

A second-order adjoint projection *of T in directions $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$ is defined as a first-order (tangent-linear or adjoint) projection in direction $\mathbf{v}$ of the first-order adjoint projection in direction $\mathbf{u}$.*

Higher-order adjoint projections *are defined recursively as*

$$\begin{aligned}
\langle \mathbf{v}_1, \mathbf{u}, T \rangle &\equiv \langle \mathbf{v}_1, \langle \mathbf{u}, T \rangle \rangle \in \mathbb{R}^{n^{p-1}} \\
\langle \mathbf{v}_2, \mathbf{v}_1, \mathbf{u}, T \rangle &\equiv \langle \mathbf{v}_2, \langle \mathbf{v}_1, \langle \mathbf{u}, T \rangle \rangle \rangle \in \mathbb{R}^{n^{p-2}} \\
&\vdots \\
\langle \mathbf{v}_p, \ldots, \mathbf{v}_1, \mathbf{u}, T \rangle &\equiv \langle \mathbf{v}_p, \langle \cdots \langle \mathbf{v}_1, \langle \mathbf{u}, T \rangle \rangle, \ldots \rangle \rangle \in \mathbb{R}.
\end{aligned}$$

Lemma 3.30. *Let $T \in \mathbb{R}^{m \times n^p}$ be a symmetric $(p+1)$-tensor as defined in Definition* 3.27, *and let $k \leq p$. Then,*

$$\langle \mathbf{v}_k, \ldots, \mathbf{v}_1, \mathbf{u}, T \rangle = \langle \pi(\mathbf{v}_k, \ldots, \mathbf{v}_1), \mathbf{u}, T \rangle$$

for any permutation π of the $\mathbf{v}_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$, and where $\mathbf{u} \in \mathbb{R}^m$.

Proof. This result follows immediately from the symmetry of T. □

Lemma 3.31. *Let $T \in \mathbb{R}^{m \times n^p}$ be defined as in Definition* 3.27, *and let $k \leq p$. Then,*

$$\langle \mathbf{u}, T, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k \rangle = \langle \mathbf{u}, T, \pi(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k) \rangle$$

for any permutation π of the $\mathbf{v}_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$, and where $\mathbf{u} \in \mathbb{R}^m$.

Proof. Again, this result follows immediately from the symmetry of T. □

Higher-order projections of symmetric tensors can be shown to be associative similarly to the arguments in Section 3.1. For example,

$$\begin{aligned}\langle \mathbf{u}_2, \mathbf{u}_1, T, \mathbf{v}_1, \mathbf{v}_2 \rangle &= \langle \mathbf{u}_2, \langle \mathbf{u}_1, \langle \langle T, \mathbf{v}_1 \rangle, \mathbf{v}_2 \rangle \rangle \rangle \\ &= \langle \langle \mathbf{u}_2, \langle \langle \mathbf{u}_1, T \rangle, \mathbf{v}_1 \rangle \rangle, \mathbf{v}_2 \rangle\end{aligned}$$

for $\mathbf{u}_1 \in \mathbb{R}^m$ and $\mathbf{u}_2, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^n$.

For $\mathbf{v} \in \mathbb{R}^n$, the expression

$$\mathbf{v}_{(j_1,\ldots,j_b)}^{(i_1,\ldots,i_f)}$$

denotes the dth derivative of $\mathbf{v}$, where $d = f + b$. The current value of the dth derivative of $\mathbf{v}$ is computed by a derivative code that resulted from the i_kth differentiation performed in forward mode for $k = 1, \ldots, f$ and where the j_lth differentiation is performed in reverse mode for $l = 1, \ldots, b$. For example, $\mathbf{v}_{(1,6)}^{(2,7)}$ represents a fourth derivative of $\mathbf{v}$ in a kth-order adjoint code ($k \geq 7$) that is obtained by a sequence of applications of forward and reverse mode AD, where the first and sixth applications are performed in reverse mode, and the second and seventh applications are performed in forward mode.

Definition 3.32. *The third derivative tensor $\nabla^3 F = \nabla^3 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n \times n}$ of a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$, where $\mathbf{y} = F(\mathbf{x})$, induces a trilinear mapping $\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$ defined by*

$$(\mathbf{u}, \mathbf{v}, \mathbf{w}) \mapsto \langle \nabla^3 F, \mathbf{u}, \mathbf{v}, \mathbf{w} \rangle.$$

The function $F^{(1,2,3)} : \mathbb{R}^{4 \cdot n} \to \mathbb{R}^m$ that is defined as

$$\mathbf{y}^{(1,2,3)} = F^{(1,2,3)}(\mathbf{x}, \mathbf{u}, \mathbf{v}, \mathbf{w}) \equiv \langle \nabla^3 F(\mathbf{x}), \mathbf{u}, \mathbf{v}, \mathbf{w} \rangle \tag{3.12}$$

is referred to as the third-order tangent-linear model *of F.*

Definition 3.33. *The third derivative tensor $\nabla^3 F = \nabla^3 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n \times n}$ of a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$, where $\mathbf{y} = F(\mathbf{x})$, induces a trilinear mapping $\mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ defined by*

$$(\mathbf{u}, \mathbf{v}, \mathbf{w}) \mapsto \langle \mathbf{u}, \nabla^3 F, \mathbf{v}, \mathbf{w} \rangle.$$

The function $F_{(1)}^{(2,3)} : \mathbb{R}^{3 \cdot n + m} \to \mathbb{R}^n$ that is defined as

$$\mathbf{x}_{(1)}^{(2,3)} = F_{(1)}^{(2,3)}(\mathbf{x}, \mathbf{u}, \mathbf{v}, \mathbf{w}) \equiv \langle \mathbf{u}, \nabla^3 F(\mathbf{x}), \mathbf{v}, \mathbf{w} \rangle \tag{3.13}$$

is referred to as the third-order adjoint model *of F.*

Symmetry of $\nabla^3 F$ implies that arbitrary combinations of applications of forward and (at least a single instance of) reverse modes yield the third-order adjoint model, that is,

$$F_{(1)}^{(2,3)} \equiv F_{(2)}^{(1,3)} \equiv F_{(3)}^{(1,2)} \equiv F_{(1,2)}^{(3)} \equiv F_{(1,3)}^{(2)} \equiv F_{(2,3)}^{(1)} \equiv F_{(1,2,3)}.$$

3.4.1 Third-Order Tangent-Linear Code

Theorem 3.34. *The application of forward mode AD to the second-order tangent-linear model yields the third-order tangent-linear model.*

Proof. The application of forward mode AD to the second-order tangent-linear model $\mathbf{y}^{(1,2)} = \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \rangle$ yields

$$\begin{aligned} \mathbf{y}^{(1,2,3)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1,3)}, \mathbf{x}^{(2)} \rangle \\ &+ \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2,3)} \rangle \\ &+ \langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle \end{aligned}$$

and hence, for $\mathbf{x}^{(1,3)} = \mathbf{x}^{(2,3)} = 0$, (3.12). □

A graphical illustration in form of the tangent-linear extension of the DAG of the second-order tangent-linear model with

$$\frac{\partial \mathbf{x}}{\partial s} = \mathbf{x}^{(3)}, \quad \frac{\partial \mathbf{x}^{(1)}}{\partial s} = \mathbf{x}^{(1,3)}, \quad \text{and} \quad \frac{\partial \mathbf{x}^{(2)}}{\partial s} = \mathbf{x}^{(2,3)}$$

can be found in Figure 3.11.

Example 3.35 The application of forward mode to the second-order tangent-linear code for (1.2) developed in Example 3.13 is straightforward. The resulting third-order tangent-linear code is about twice the size of the second-order tangent-linear code.

```
1 void t3_t2_t1_f(int n, double* x, double* t3_x,
2                 double* t2_x, double* t3_t2_x,
3                 double* t1_x, double* t3_t1_x,
4                 double* t2_t1_x, double* t3_t2_t1_x,
5                 double& y, double& t3_y,
6                 double& t2_y, double& t3_t2_y,
7                 double& t1_y, double& t3_t1_y,
8                 double& t2_t1_y, double& t3_t2_t1_y) {
9   t3_t2_t1_y=0; t2_t1_y=0;
10  t3_t1_y=0; t1_y=0;
11  t3_t2_y=0; t2_y=0;
12  t3_y=0; y=0;
13  for (int i=0;i<n;i++) {
14    t3_t2_t1_y=t3_t2_t1_y+2*(t3_t2_x[i]*t1_x[i]
15                            +t2_x[i]*t3_t1_x[i]
16                            +t3_x[i]*t2_t1_x[i]
17                            +x[i]*t3_t2_t1_x[i]);
18    t2_t1_y=t2_t1_y+2*(t2_x[i]*t1_x[i]+x[i]*t2_t1_x[i]);
19
20    t3_t1_y=t3_t1_y+2*(t3_x[i]*t1_x[i]+x[i]*t3_t1_x[i]);
21    t1_y=t1_y+2*x[i]*t1_x[i];
22    t3_t2_y=t3_t2_y+2*(t3_x[i]*t2_x[i]+x[i]*t3_t2_x[i]);
```

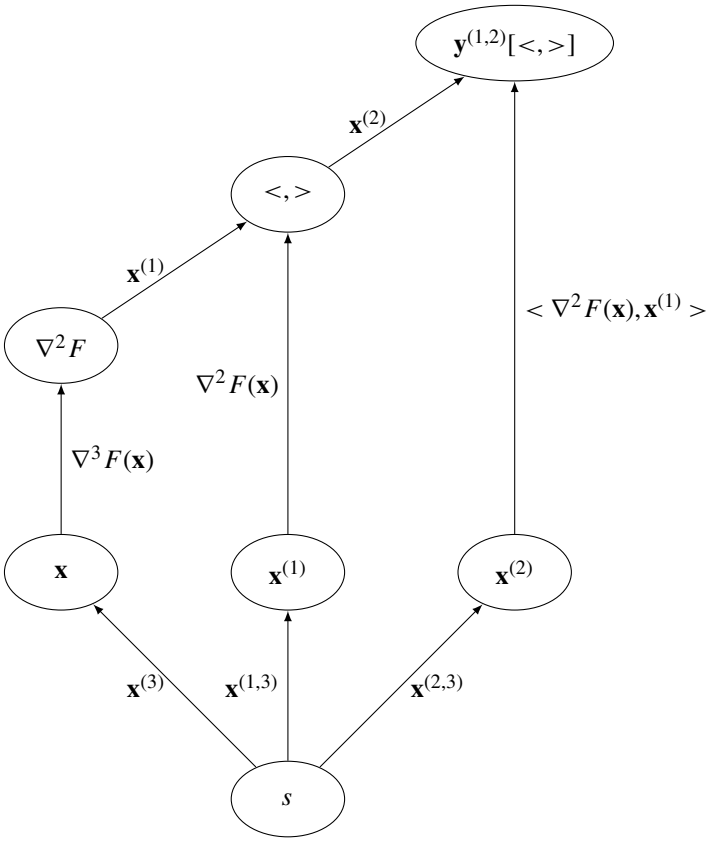


Figure 3.11. *Tangent-linear extension of the linearized DAG of the second-order tangent-linear model of* $\mathbf{y} = F(\mathbf{x})$.

```
23          t2_y=t2_y+2*x[i]*t2_x[i];
24          t3_y=t3_y+2*x[i]*t3_x[i];
25          y=y+x[i]*x[i];
26      }
27      t3_t2_t1_y=2*(t3_t2_y*t1_y+t2_y*t3_t1_y+t3_y*t2_t1_y
28                                              +y*t3_t2_t1_y);
29      t2_t1_y=2*t2_y*t1_y+2*y*t2_t1_y;
30      t3_t1_y=2*t3_y*t1_y+2*y*t3_t1_y;
31      t1_y=2*y*t1_y;
32      t3_t2_y=2*t3_y*t2_y+2*y*t3_t2_y;
33      t2_y=2*y*t2_y;
34      t3_y=2*y*t3_y;
35      y=y*y;
36  }
```

Each assignment is preceded by its tangent-linear version; for example, the tangent-linear version of the assignment in line 33 is inserted in line 32. First-order projections of $\nabla^3 F(\mathbf{x})$ in directions t1_x, t2_x, and t3_x are returned in t1_y, t2_y, and t3_y, respectively. Corresponding second-order projections are returned in t2_t1_y, t3_t2_y, and t3_t1_y.

The following driver program computes the entire third derivative tensor:

```
int main() {
  const int n=4;
  double x[n], t1_x[n], t2_x[n], t2_t1_x[n];
  double t3_x[n], t3_t1_x[n], t3_t2_x[n], t3_t2_t1_x[n];
  double y, t1_y, t2_y, t2_t1_y;
  double t3_y, t3_t1_y, t3_t2_y, t3_t2_t1_y;

  for (int j=0;j<n;j++) {
    x[j]=1;
    t2_t1_x[j]=t2_x[j]=t1_x[j]=t3_x[j]
              =t3_t2_t1_x[j]=t3_t2_x[j]=t3_t1_x[j]=0;
  }

  for (int k=0;k<n;k++) {
    t1_x[k]=1;
    for (int j=0;j<=k;j++) {
      t2_x[j]=1;
      for (int i=0;i<=j;i++) {
        t3_x[i]=1;
        t3_t2_t1_f(n,x,t3_x,t2_x,t3_t2_x,t1_x,
                   t3_t1_x,t2_t1_x,t3_t2_t1_x,
                   y,t3_y,t2_y,t3_t2_y,t1_y,
                   t3_t1_y,t2_t1_y,t3_t2_t1_y);
        cout << "H[" << k << "][" << j << "][" << i << "]="
             << t3_t2_t1_y << endl;
        t3_x[i]=0;
      }
      t2_x[j]=0;
    }
    t1_x[k]=0;
  }
  return 0;
}
```

The third partial derivatives are returned in t3_t2_t1_y. Symmetry is exploited. ∎

3.4.2 Third-Order Adjoint Code

The preferred approach to the computation of higher derivatives of multivariate scalar functions is the repeated application of forward mode AD to the adjoint code.

Theorem 3.36. *The application of forward mode AD to the second-order adjoint model yields the third-order adjoint model.*

Proof. The application of forward mode AD to the second-order adjoint model

$$\mathbf{x}_{(1)}^{(2)} = \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle$$

yields

$$\begin{aligned}\mathbf{x}_{(1)}^{(2,3)} &= \langle \mathbf{y}_{(1)}^{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \rangle \\ &+ \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle \\ &+ \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle\end{aligned}$$

and hence, for $\mathbf{y}_{(1)}^{(3)} = \mathbf{x}^{(2,3)} = 0$, (3.13). □

A graphical illustration in form of the tangent-linear extension of the DAG of the second-order adjoint model with

$$\frac{\partial \mathbf{x}}{\partial s} = \mathbf{x}^{(3)}, \quad \frac{\partial \mathbf{y}_{(1)}}{\partial s} = \mathbf{y}_{(1)}^{(3)}, \quad \text{and} \quad \frac{\partial \mathbf{x}^{(2)}}{\partial s} = \mathbf{x}^{(2,3)}$$

can be found in Figure 3.12.

Example 3.37 The application of forward mode to the second-order adjoint code for (1.2) developed in Example 3.19 does not pose any difficulties either. The resulting third-order adjoint code is about twice the size of the second-order adjoint code.

```
1  void t3_t2_a1_f(int n, double* x, double* t3_x,
2                  double* t2_x, double* t3_t2_x,
3                  double* a1_x, double* t3_a1_x,
4                  double* t2_a1_x, double* t3_t2_a1_x,
5                  double& y, double& t3_y,
6                  double& t2_y, double& t3_t2_y,
7                  double a1_y, double t3_a1_y,
8                  double t2_a1_y, double t3_t2_a1_y) {
9    t3_t2_y=0;
10   t2_y=0;
11   t3_y=0;
12   y=0;
13   for (int i=0;i<n;i++) {
14     t3_t2_y=t3_t2_y+2*(t3_x[i]*t2_x[i]+x[i]*t3_t2_x[i]);
15     t2_y=t2_y+2*x[i]*t2_x[i];
16     t3_y=t3_y+2*x[i]*t3_x[i];
17     y=y+x[i]*x[i];
18   }
19   t3_t2_required_double.push(t3_t2_y);
20   t2_required_double.push(t2_y);
21   t3_required_double.push(t3_y);
22   required_double.push(y);
23   t3_t2_y=2*(t3_y*t2_y+y*t3_t2_y);
24   t2_y=2*y*t2_y;
25   t3_y=2*y*t3_y;
```

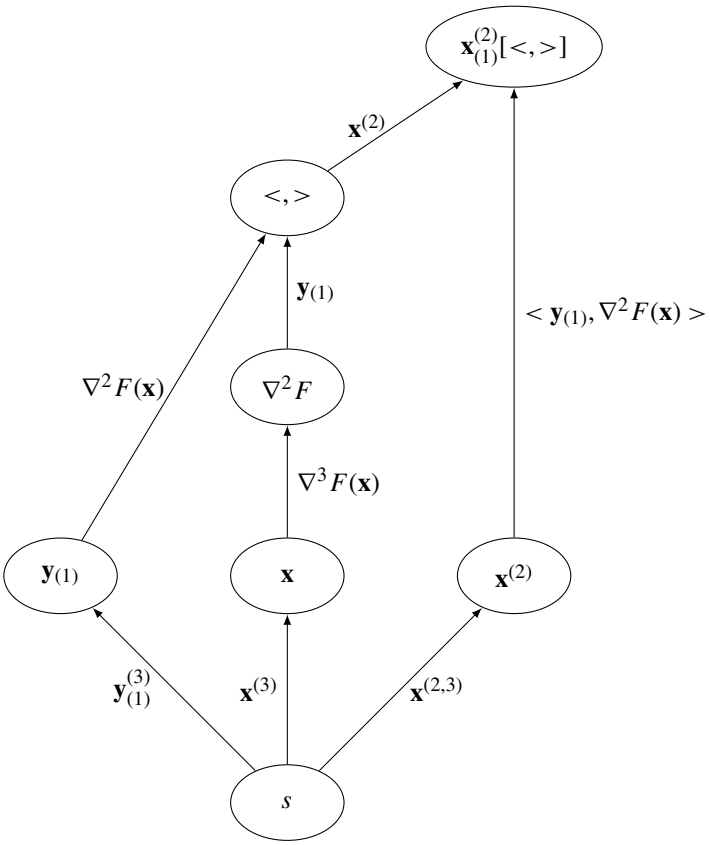


Figure 3.12. *Tangent-linear extension of the linearized DAG of the second-order adjoint model of* $\mathbf{y} = F(\mathbf{x})$.

```
26  y=y*y;
27
28  t3_t2_y=t3_t2_required_double.top();
29  t3_t2_required_double.pop();
30  t2_y=t2_required_double.top(); t2_required_double.pop();
31  t3_y=t3_required_double.top(); t3_required_double.pop();
32  y=required_double.top(); required_double.pop();
33  t3_t2_a1_y=2*(t3_t2_y*a1_y+t2_y*t3_a1_y
34                +t3_y*t2_a1_y+y*t3_t2_a1_y);
35  t2_a1_y=2*(t2_y*a1_y+y*t2_a1_y);
36  t3_a1_y=2*(t3_y*a1_y+y*t3_a1_y);
37  a1_y=2*y*a1_y;
38  for (int i=n-1;i>=0;i--) {
39    t3_t2_a1_x[i]=t3_t2_a1_x[i]+2*(t3_t2_x[i]*a1_y
40        +t2_x[i]*t3_a1_y+t3_x[i]*t2_a1_y+x[i]*t3_t2_a1_y);
41    t2_a1_x[i]=t2_a1_x[i]+2*(t2_x[i]*a1_y+x[i]*t2_a1_y);
```

```
42      t3_a1_x[i]=t3_a1_x[i]+2*(t3_x[i]*a1_y+x[i]*t3_a1_y);
43      a1_x[i]=a1_x[i]+2*x[i]*a1_y;
44    }
45  }
```

Each assignment is preceded by its tangent-linear version; for example, the tangent-linear version of the assignment in line 41 is inserted in lines 39–40. All stacks are duplicated. The respective accesses are augmented with corresponding accesses of the tangent-linear stacks.

The following driver program computes the whole third derivative tensor:

```
int main() {
  const int n=4;
  double x[n], y;
  double t3_x[n], t3_y;
  double t2_x[n], t2_y;
  double t3_t2_x[n], t3_t2_y;
  double a1_x[n], a1_y;
  double t3_a1_x[n], t3_a1_y;
  double t2_a1_x[n], t2_a1_y;
  double t3_t2_a1_x[n], t3_t2_a1_y;
  for (int i=0;i<n;i++) {
    x[i]=1; t3_x[i]=t2_x[i]=t3_t2_x[i]=0;
  }
  for (int k=0;k<n;k++) {
    t3_x[k]=1;
    for (int i=0;i<n;i++) {
      for (int j=0;j<n;j++)
        a1_x[j]=t3_a1_x[j]=t2_a1_x[j]=t3_t2_a1_x[j]=0;
      y=t3_y=t2_y=t3_t2_y=t3_a1_y=t2_a1_y=t3_t2_a1_y=0;
      a1_y=1;
      t2_x[i]=1;
      t3_t2_a1_f(n,x,t3_x,t2_x,t3_t2_x,a1_x,
                 t3_a1_x,t2_a1_x,t3_t2_a1_x,
                 y,t3_y,t2_y,t3_t2_y,a1_y,
                 t3_a1_y,t2_a1_y,t3_t2_a1_y);
      for (int j=0;j<n;j++)
        cout << "H[" << k << "][" << i << "][" << j << "]="
             << t3_t2_a1_x[j] << endl;
      t2_x[i]=0;
    }
    t3_x[k]=0;
  }
  return 0;
}
```

Refer to Table 3.4 for run-time measurements of third derivative code.

Third-order adjoint code is obtained by arbitrary combinations of forward and reverse mode AD. For example, application of reverse mode with required data stack s and result

Table 3.4. *Run times for third-order tangent-linear and adjoint code (in seconds). In order to determine the relative computational complexity $\mathcal{R}$ of the derivative code, n function evaluations are compared with n evaluations of the third-order tangent-linear code (t3_t2_t1_f) and with the same number of evaluations of the third-order adjoint code that was generated in forward-over-forward-over-reverse mode* (t3_t2_a1_f). *We observe a factor less than* 8 *when comparing the run time of a single run of the third-order tangent-linear code with that of an original function evaluation. $O(n^3)$ runs of the third-order tangent-linear code are required for the evaluation of the whole third derivative tensor or of projections thereof. Even less time is taken by a single execution of the third-order adjoint code due to more effective compiler optimization. Moreover, only $O(n^2)$ runs are required for the evaluation of the whole third derivative tensor. Second- and third-order projections thereof can even be computed with a computational complexity of $O(n)$ and $O(1)$, respectively.*

	g++ -O0			g++ -O3			$\mathcal{R}$
n	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	10^4	$2 \cdot 10^4$	$4 \cdot 10^4$	
f	0.9	3.6	13.7	0.2	0.8	3.1	1
t3_t2_t1_f	7.4	28.1	113.6	1.4	5.5	23.8	≈ 7.7
t3_t2_a1_f	8.2	31.2	129.3	1.0	4.0	16.9	≈ 5.5

checkpoint r to the second-order adjoint model in (3.6) yields reverse-over-reverse-over-forward mode. The augmented forward section becomes

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}^{(1)}_{(2)} &= \mathbf{x}^{(1)}_{(2)} + \langle \mathbf{y}^{(1)}_{(2)}, \nabla F(\mathbf{x}) \rangle \\
s[0] &= \mathbf{y}_{(2)} \\
\mathbf{y}_{(2)} &= 0 \\
s[1] &= \mathbf{y}^{(1)}_{(2)} \\
\mathbf{y}^{(1)}_{(2)} &= 0.
\end{aligned}$$

It is succeeded by the following reverse section.

$$\begin{aligned}
\mathbf{y}^{(1)}_{(2)} &= s[1] \\
\mathbf{y}^{(1)}_{(2,3)} &= 0 \\
\mathbf{y}_{(2)} &= s[0] \\
\mathbf{y}_{(2,3)} &= 0 \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}^{(1)}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}^{(1)}_{(2,3)} &= \mathbf{y}^{(1)}_{(2,3)} + \langle \mathbf{x}^{(1)}_{(2,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(2,3)} &= \mathbf{y}_{(2,3)} + \langle \mathbf{x}_{(2,3)}, \nabla F(\mathbf{x}) \rangle
\end{aligned}$$

$$
\begin{aligned}
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}^{(1)}_{(2,3)} &= \mathbf{y}^{(1)}_{(2,3)} + \langle \mathbf{x}_{(2,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}^{(1)}_{(3)} &= \mathbf{x}^{(1)}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{y}^{(1)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}^{(1)}_{(3)} &= \mathbf{x}^{(1)}_{(3)} + \langle \mathbf{y}^{(1)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(1)}_{(3)} &= 0 \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(3)} &= 0.
\end{aligned}
$$

Constant-folding, copy propagation, and substitution yield

$$
\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{x}^{(1)}_{(2)} &= \mathbf{x}^{(1)}_{(2)} + \langle \mathbf{y}^{(1)}_{(2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}^{(1)}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle + \langle \mathbf{y}^{(1)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
&\quad + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(1)}_{(2,3)} &= \langle \mathbf{x}^{(1)}_{(2,3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(2,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\
\mathbf{y}_{(2,3)} &= \langle \mathbf{x}_{(2,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}^{(1)}_{(3)} &= \mathbf{x}^{(1)}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(1)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(1)}_{(3)} &= 0 \\
\mathbf{y}_{(3)} &= 0.
\end{aligned}
$$

The entire third derivative tensor can be accumulated at the computational cost of $O(m \cdot n^2) \cdot Cost(F)$ by setting $\mathbf{x}_{(3)} = \mathbf{y}_{(2)} = \mathbf{x}^{(1)}_{(2,3)} = \mathbf{y}_{(3)} = \mathbf{y}^{(1)}_{(3)} = 0$ initially and by letting $\mathbf{x}_{(2,3)}$, $\mathbf{y}^{(1)}_{(2)}$, and $\mathbf{x}^{(1)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$, $\mathbb{R}^m$, and $\mathbb{R}^n$, respectively. Projections of $\nabla^3 F(\mathbf{x})$ can be obtained at a lower computational cost; for example,

- $\langle \mathbf{x}_{(2,3)}, \mathbf{y}^{(1)}_{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \in \mathbb{R}$ at the cost of $O(1) \cdot Cost(F)$;
- $\langle \mathbf{x}_{(2,3)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \in \mathbb{R}^m$ at the cost of $O(m) \cdot Cost(F)$ ($\mathbf{y}^{(1)}_{(2)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^m$);

- $\langle \mathbf{x}_{(2,3)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}^{m \times n}$ at the cost of $O(m \cdot n) \cdot Cost(F)$ ($\mathbf{y}_{(2)}^{(1)}$ and $\mathbf{x}^{(1)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively);

- $\langle \mathbf{y}_{(2)}^{(1)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}^{n \times n}$ at the cost of $O(n^2) \cdot Cost(F)$ ($\mathbf{x}_{(2,3)}$ and $\mathbf{x}^{(1)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$);

Moreover, the third-order adjoint code returns arbitrary projections of the second and first derivative tensors in addition to the original function value. Potential sparsity should be exploited whenever applicable.

3.4.3 Fourth and Higher Derivative Code

Projections of fourth and potentially higher derivative tensors may be required if a numerical second-order algorithm is applied to a simulation code that already contains second or higher derivatives of some underlying function. The use of AD remains straightforward.

Definition 3.38. *The fourth derivative tensor $\nabla^4 F = \nabla^4 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n \times n \times n}$ of a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$, where $\mathbf{y} = F(\mathbf{x})$, induces a quadrilinear mapping $\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$ defined by*

$$(\mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w}) \mapsto \langle \nabla^4 F, \mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w} \rangle.$$

The function $F^{(1,2,3,4)} : \mathbb{R}^{5 \cdot n} \to \mathbb{R}^m$ that is defined as

$$\mathbf{y}^{(1,2,3,4)} = F^{(1,2,3,4)}(\mathbf{x}, \mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w}) \equiv \langle \nabla^4 F(\mathbf{x}), \mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w} \rangle \tag{3.14}$$

is referred to as the fourth-order tangent-linear model *of F.*

Definition 3.39. *The fourth derivative tensor $\nabla^4 F = \nabla^4 F(\mathbf{x}) \in \mathbb{R}^{m \times n \times n \times n \times n}$ of a multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^m$, where $\mathbf{y} = F(\mathbf{x})$, induces a quadrilinear mapping $\mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ defined by*

$$(\mathbf{u}, \mathbf{t}, \mathbf{v}, \mathbf{w}) \mapsto \langle \mathbf{u}, \nabla^4 F, \mathbf{t}, \mathbf{v}, \mathbf{w} \rangle.$$

The function $F_{(1)}^{(2,3,4)} : \mathbb{R}^{4 \cdot n + m} \to \mathbb{R}^n$ that is defined as

$$\mathbf{x}_{(1)}^{(2,3,4)} = F_{(1)}^{(2,3,4)}(\mathbf{x}, \mathbf{u}, \mathbf{t}, \mathbf{v}, \mathbf{w}) \equiv \langle \mathbf{u}, \nabla^4 F(\mathbf{x}), \mathbf{t}, \mathbf{v}, \mathbf{w} \rangle \tag{3.15}$$

is referred to as the fourth-order adjoint model *of F.*

Fourth-order tangent-linear code is generated in forward-over-forward-over-forward-over-forward mode.

Theorem 3.40. *The application of forward mode AD to the third-order tangent-linear model yields the fourth-order tangent-linear model*

Proof. Application of forward mode AD to the third-order tangent-linear model $\mathbf{y}^{(1,2,3)} = \langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle$ yields

$$\begin{aligned}\mathbf{y}^{(1,2,3,4)} = &\langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1,4)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle \\ &+ \langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2,4)}, \mathbf{x}^{(3)} \rangle \\ &+ \langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(1)}, , \mathbf{x}^{(2)}, \mathbf{x}^{(3,4)} \rangle \\ &+ \langle \nabla^4 F(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle\end{aligned}$$

and hence, for $\mathbf{x}^{(1,4)} = \mathbf{x}^{(2,4)} = \mathbf{x}^{(3,4)} = 0$, (3.14). □

For $m \ll n$, fourth-order adjoint code is typically built in forward-over-forward-over-forward-over-reverse mode.

Theorem 3.41. *The application of forward mode AD to the third-order adjoint model yields the fourth-order adjoint model.*

Proof. The application of forward mode AD to the third-order adjoint model

$$\mathbf{x}_{(1)}^{(2,3)} = \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle$$

yields

$$\begin{aligned}\mathbf{x}_{(1)}^{(2,3,4)} = &\langle \mathbf{y}_{(1)}^{(4)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle \\ &+ \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,4)}, \mathbf{x}^{(3)} \rangle \\ &+ \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3,4)} \rangle \\ &+ \langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle\end{aligned}$$

and hence, for $\mathbf{y}_{(1)}^{(4)} = \mathbf{x}^{(2,4)} = \mathbf{x}^{(3,4)} = 0$, (3.15). □

Fourth-order adjoint code is obtained by arbitrary combinations of forward and reverse mode AD. For example, forward mode can be applied to the third-order adjoint model derived in Section 3.4.2. The entire fourth derivative tensor can be accumulated at a computational cost of $O(m \cdot n^3) \cdot Cost(F)$. Projections of $\nabla^4 F(\mathbf{x})$ can be obtained at a lower computational cost. The fourth-order adjoint code returns arbitrary projections of the third, second, and first derivative tensors in addition to the original function value. Again, potential sparsity should be exploited to reduce the computational cost and the memory requirement.

Fifth and higher derivative models are derived accordingly. An interesting alternative to the computation of higher partial derivatives as projections of the corresponding derivative tensors is presented in [35]. An in-depth discussion of further issues in AD for higher derivatives is beyond the scope of this introduction. Refer to [36] for additional information on this topic.

3.5 Exercises

3.5.1 Second Derivative Code

Consider the code in Listing 2.3.

1. Write second-order tangent-linear code based on the tangent-linear code that was developed in Section 2.4.1; use it to accumulate the Hessian of the dependent output y with respect to the independent input x.

2. Write second-order adjoint code based on the adjoint code that was developed in Section 2.4.1 (forward-over-reverse mode in both split and joint modes); use it to accumulate the same Hessian as in 1.

3. Write second-order adjoint code based on the tangent-linear code that was developed in Section 2.4.1 (reverse-over-forward mode in both split and joint modes); use it to accumulate the same Hessian as in 1.

3.5.2 Use of Second Derivative Models

Consider the given implementation of the extended Rosenbrock function f from Section 1.4.3.

1. Write a second-order tangent-linear code and use it to accumulate $\nabla^2 f$ with machine accuracy. Compare the numerical results with those obtained by finite difference approximation.

2. Write a second-order adjoint code in forward-over-reverse mode and use it to accumulate $\nabla^2 f$ with machine accuracy. Compare the numerical results with those obtained with the second-order tangent-linear approach.

3. Use `dco` to accumulate $\nabla^2 f$ in second-order tangent-linear and adjoint modes with machine accuracy. Compare the numerical results with those obtained from the hand-written derivative code.

4. Use the Newton algorithm and a corresponding matrix-free implementation based on the CG method for the solution of the Newton system to minimize the extended Rosenbrock function for different start values of your own choice.

5. Compare the run times for the various approaches to computing the required derivatives as well as the run times of the optimization algorithms for increasing values of n.

3.5.3 Third and Higher Derivative Models

1. Write third-order tangent-linear and adjoint versions for the code in 3.5.1. Run numerical tests to verify correctness.

2. Given $\mathbf{y} = F(\mathbf{x})$, derive the following higher derivative code and provide drivers for its use in the accumulation of the corresponding derivative tensors:

(a) third-order adjoint code in reverse-over-reverse-over-reverse mode;

(b) fourth-order adjoint code in forward-over-forward-over-forward-over-reverse mode;

(c) fourth-order adjoint code in reverse-over-forward-over-reverse-over-forward mode.

Discuss the complexity of computing various projections of the third and fourth derivative tensors.

Chapter 4

Derivative Code Compilers—An Introductory Tutorial

The following chapter serves as the basis for a one-semester lab on derivative code compilers. Through inclusion of this material, we hope to give the interested reader some insight into the automatization of derivative code generation as introduced in Chapters 2 and 3. Readers without interest in technical issues related to derivative code compilers may skip this chapter and proceed to Chapter 5, where the result in the form of a fully functional prototype derivative code compiler for a small subset of C++ is presented.

Typically, the purpose of a compiler front-end is twofold: The given source code is verified syntactically, that is, the front-end checks whether the input is a valid sequence of words from the given programming language. Moreover, the source is transformed from a pure sequence of characters into a structured representation that captures the syntactic properties of the program. A basic internal representation consists of an *abstract syntax (or parse) tree* and a *symbol table.* Various extensions and modifications are used in practice. Semantic tests are needed to verify the correctness of the given source code completely. In this book we assume that any input program is correct, both syntactically and semantically. In any case, a native compiler for the source language is required in order to be able to process the generated derivative code. The user of our basic derivative code compiler is expected to have validated the input's syntax and semantics with the help of the native compiler. Hence, we use the compiler front-end simply as a transformation engine delivering an abstract intermediate representation of the input that is then used for semantic modification. No semantic analysis is performed.

We start this chapter with a brief overview of the basic structure of a derivative code compiler in Section 4.1. Fundamental terminology required for lexical and syntax analysis is introduced in Section 4.2. Lexical analysis and its implementation by using the *scanner generator* `flex` is covered in Section 4.3 followed by syntax analysis and its implementation using the *parser generator* `bison` in Section 4.4. The logic behind parse tree construction algorithms is exploited for the single-pass syntax-directed compilation of derivative code in Section 4.5. The advantage of multipass compilation algorithms is the ability to annotate the intermediate representation with information obtained by *static program analysis*. Thus, more complex language constructs can be analyzed semantically, and potentially more efficient derivative code can be generated. The foundations of multipass source transformation in the form of an abstract intermediate representation are laid in Section 4.6.

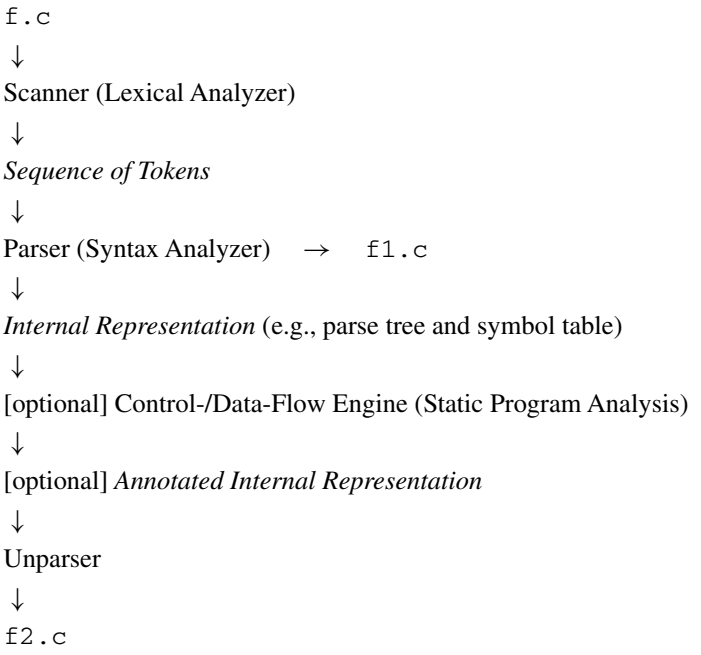


Figure 4.1. *Derivative code compiler.*

4.1 Overview

The main stages of a simple derivative code compiler are shown in Figure 4.1. The characters in a given input file (e.g. `f.c`) are grouped by the scanner into so-called *tokens* that are defined by a *regular grammar* that guides the lexical analysis as described in Section 4.3. For example, the simple product reduction

```
void f(int n, double * x, double & y) {
  int i=0;
  while (i<n)
    if (i==0)
      y=x[0];
    else
      y=y*x[i];
    i=i+1;
}
```

becomes

```
VOID N '(' INT N ',' DBL '*' N ',' DBL '&' N ')' '{'
INT N '=' C ';'
WHILE '(' N LT N ')'
IF '(' N EQ C ')'
```

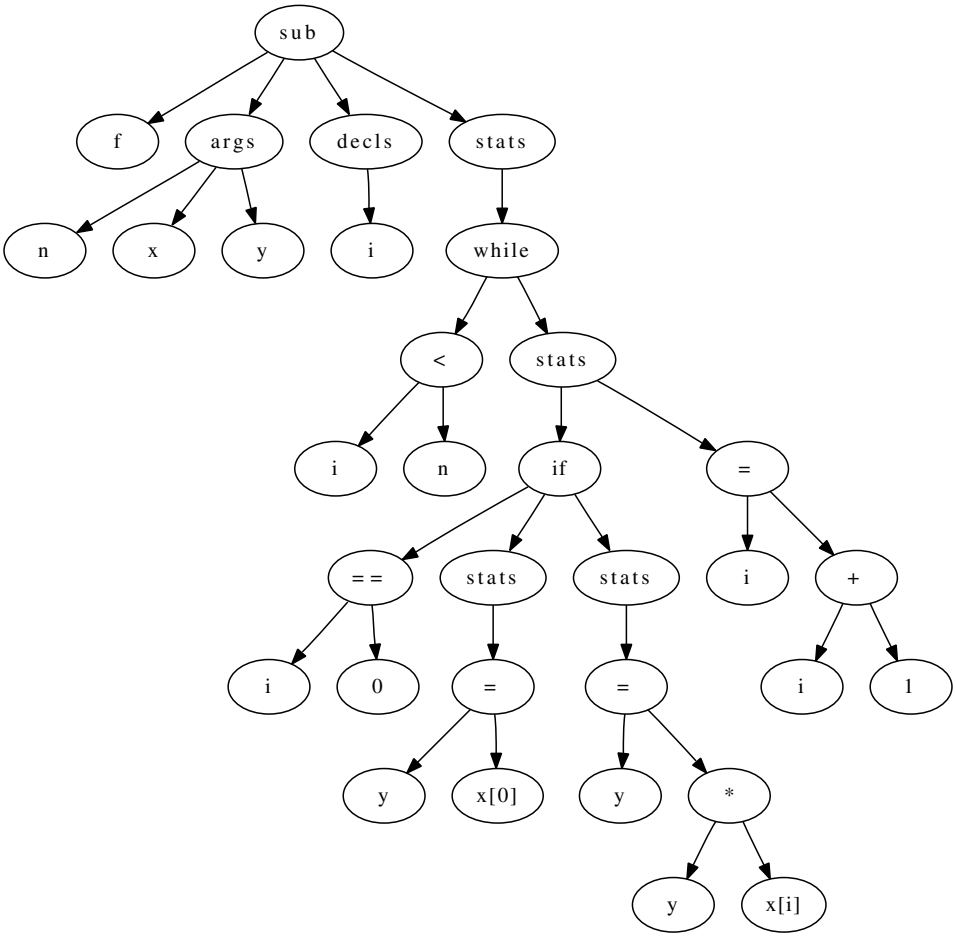


Figure 4.2. *Parse tree of product reduction.*

```
N '=' N '[' C ']' ';'
ELSE
N '=' N '*' N '[' N ']' ';'
N '=' N '+' C ';'
'}'
```

with *single-character tokens* enclosed in single quotes. Names of variables or subroutines are described syntactically by the *regular expression* `N : [a-z]|([a-z]|[0-9])*`. They always start with a lowercase letter followed by a possibly empty sequence of lowercase letters or digits. The specific names associated with tokens of type `N (f, n, x, y, i)` are stored in a *symbol table*.

A parser builds a *parse tree* as shown in Figure 4.2. At the same time, the symbol table is augmented with additional information including kinds (subroutine – 1 or variable

– 2), types (double – 1 or integer – 2), and shapes (scalar – 1 or vector – 2) of symbols.[8] For example,

Name	Kind	Type	Shape
f	1	0	0
n	2	2	1
x	2	1	2
y	2	1	1
i	2	2	1

Undefined properties are marked with 0. Access to symbol information through the parse tree can be implemented in the leaf nodes as pointers to the corresponding symbol table entries. Syntax analysis techniques are discussed in Section 4.4.

Syntax analysis is driven by a *grammar*. Instead of building a parse tree, a single-pass derivative code compiler can write derivative code as it parses the input. This *syntax-directed* approach to the semantic transformation of computer programs is discussed in Section 4.5 and is not generally applicable. It can be used to generate derivative code for the subset of C++ considered in this book. For example, parsing of the arguments (construction of the parse tree rooted by the args node in Figure 4.2) identifies both x and y as variables of type **double**. When processing the assignment y=y∗x[i] (construction of the corresponding subtree in Figure 4.2), the parser can immediately generate the tangent-linear assignment t1_y=t1_y∗x[i]+y∗t1_x[i] by applying the well-known differentiation rules (here the product rule) built on predefined prefixes (for example, t1_) to variable names for the directional derivatives. This single-pass approach is illustrated in Figure 4.3 showing the relevant part of the parse tree. We assume that the latter is built from the bottom up (following the enumeration of the vertices from 1 to 5) based, for example, on the following simplified set of syntax (also *production*) rules:

```
(P1)   a : v '=' e ';'
(P2)   e : e '*' e
(P3)     | v
(P4)   v : N
(P5)     | N '[' N ']'
```

These rules describe assignments that consist of a variable v on the left-hand side and an expression e on the right-hand side. Expressions are defined recursively as products of expressions or as single variables. Variables can be scalars (symbols) or elements of vectors, where the index is assumed to be represented by a symbol. Although this grammar is *ambiguous* (see Section 4.2), it is well-suited for the illustration of the syntax-directed generation of tangent-linear code for the assignment y=y∗x[i].

The identification of y and x[i] as variables gives the compiler access to the corresponding variable names through pointers into the symbol table. Matching variables to hold the directional derivatives are generated by adding the predefined prefix t1_ to the original variable name. The left-hand side of the assignment is represented by vertex 1. Both y and x[i] are recognized as expressions according to production rule $P3$, yielding vertices 2 and 3. *Reduction* of the product of both expressions using rule $P2$ introduces

[8] Our example is based on the syntax and semantics accepted by version 0.9 of `dcc`. In particular, all subprograms are expected to have return type **void**. Pointer arguments are always interpreted as arrays.

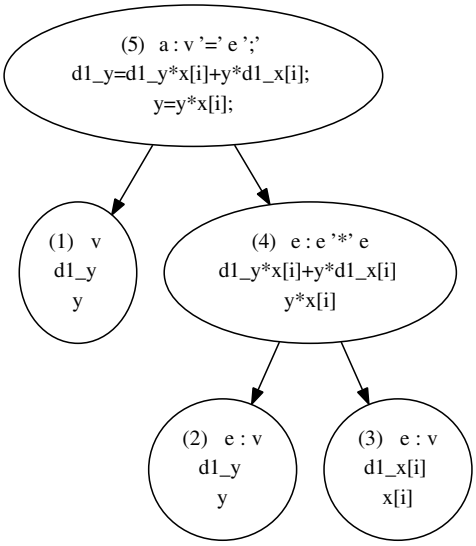


Figure 4.3. *Syntax-directed tangent-linear code.*

vertex 4. At the same time, the parsing algorithm has access to all "ingredients" of the original expression y∗x[i], as well as of its tangent-linear counterpart t1_y∗x[i]+y∗t1_x[i]. The multiplication is handled according to the product rule of differentiation. Names of all variables involved can be accessed via pointers into the symbol table stored in vertices 2 and 3. The final reduction uses rule $P1$ to concatenate the code fragments in vertices 1 and 4 as left- and right-hand sides of the tangent-linear and the original assignments.

The syntax-directed generation of tangent-linear code is discussed in detail in Section 4.5.3. More sophisticated semantic transformations such as the generation of adjoint code can be accomplished with syntax-directed translation as discussed in Section 4.5.4. While single-pass syntax-directed translation is not applicable to arbitrary languages, the underlying *attribute grammars* provide a useful formalism for the specification of parse tree augmentation and transformation algorithms. This fact will be exploited in Sections 4.5.3 and 4.5.4.

Static (performed at compile time) program analysis techniques aim to gather additional information on the properties of the program to be used for the generation of optimized (more efficient in terms of run time and memory requirement) target code. The internal representation is annotated with this information. Correctness of the results of semantic transformation is always ensured by a *conservative* approach that refuses to trade robustness for efficiency. Domain-specific data-flow analysis, for example, might ask for the assignments to be classified as *active* (a nonzero directional derivative *may* be associated with the variable on the left-hand side) or *passive*. Static data-flow analyses compute conservative estimates of such kind of information iteratively as fixed points on the *control-flow graph*. The control-flow graph of the above product reduction is shown in Figure 4.4. Walks (sequences of adjacent vertices) from the (unique) entry node (−>) to the (unique) exit (<−) represent conservative estimates of feasible sequences of assignments potentially executed by the underlying code. Tracking actual values of variables in the context of

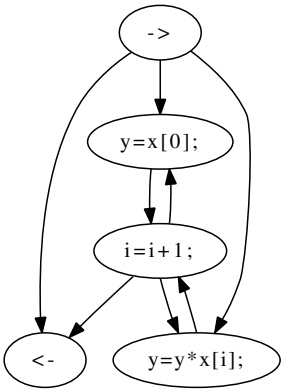


Figure 4.4. *Control flow graph of product reduction.*

abstract interpretation [20] might give a more precise representation of the flow of control. For example, the fact that y=y∗x[i] can never be the first assignment could be revealed. The corresponding edge that emanates from the entry node could be removed.

Suppose that a tangent-linear code is to be constructed that computes directional derivatives of the scalar output y with respect to the input vector x. *Activity analysis* will mark both assignments to y as active as both left-hand sides depend on some component of the independent input vector x. Moreover, they have an impact on y as the dependent output of the subroutine f. The incrementation of i is found to be passive due to its missing dependence on x. Consequently, the tangent-linear unparser modifies the signature of f and it inserts tangent-linear assignments prior to the two assignments to y leading to the following output:

```
void t1_f(int n, double * x, double * t1_x,
                 double & y, double & t1_y) {
  int i=0;
  while (i<n)
    if (i==0) {
      t1_y=t1_x[0];
      y=x[0];
    }
    else {
      t1_y=t1_y*x[i]+y*t1_x[i];
      y=y*x[i];
    }
    i=i+1;
}
```

More substantial improvements of the generated derivative code can be expected in practice. A detailed discussion of static program analysis methods specific to the domain of derivative code generation is beyond the scope of this introductory text. Refer to [37, 38] for further information on this topic. The tangent-linear and adjoint *unparsers* discussed in this book operate directly on the intermediate representation. Further static program analysis is not required.

4.2 Fundamental Concepts and Terminology

Similar to human languages, the syntax of programming languages is defined through grammars over alphabets forming words, sometimes also referred to as *sentences*. An *alphabet* Σ is a finite, nonempty set of symbols. Examples are the binary alphabet $\Sigma = \{0,1\}$, the alphabet containing all uppercase letters $\Sigma = \{A, B, \ldots\}$, or the set of all ASCII characters. *Words* (strings) are finite sequences of symbols from an alphabet Σ. The empty string ϵ has zero occurrences of symbols from Σ. For example, 010001111 is a binary word. *Languages* are all $L \subseteq \Sigma^*$, where $\Sigma^0 \equiv \epsilon$, $\Sigma^1 = \Sigma$, $\Sigma^2 = \Sigma \times \Sigma$, and so forth, and $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$. The set of all C++ programs forms a language; so does the set of all SL programs (see Section 4.3) as a subset of all C++ programs.

Definition 4.1. *A* grammar *G is a quadruple $G = (V_t, V_n, s, P)$ where*

- *V_t is a finite set of terminal symbols (also: terminals);*
- *V_n is a finite set of nonterminal symbols (also: nonterminals) such that $V_t \cap V_n = \varnothing$;*
- *$s \in V_n$ is the start symbol;*
- *P is a nonempty finite set of production rules (also: productions) of the form $u \to v$, where $u \in V_n$ and $v \in (V_t \cup V_n)^*$.*

Definition 4.2. *Consider a grammar $G = (V_t, V_n, s, P)$ and let $V = V_t \cup V_n$. A word $\sigma_2 = xvz$ over V with $x, z \subset \Sigma^*$ and $v \in V$ can be* derived *from a word $\sigma_1 = xuz$ over V with $u \in V_n$ if $(u \to v) \in P$. Derivation is denoted by $\sigma_1 \Rightarrow \sigma_2$. The relation $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$.*

Any sequence of derivations can be represented by a tree, referred to as the *abstract syntax tree* (AST) or *parse tree*. The root stands for the start symbol. The children of a node in the tree correspond to the symbols on the right-hand side of the production used to perform the respective derivation.

Definition 4.3. *The* language *$L(G) = \{\sigma \in \Sigma^* : s \Rightarrow^* \sigma\}$ that is generated by a grammar $G = (V_t, V_n, s, P)$ contains all words that can be derived from the start symbol.*

Example 4.4 Let $G = (V_t, V_n, s, P)$ with terminal symbols $V_t = \{W, O\}$, nonterminals $V_n = \{a, b, c, d\}$, start symbol $s = a$, and production rules $a \to Wb$, $b \to Oc$, $b \to Ob$, $c \to Wd$, $d \to \epsilon$. A possible derivation of $WOOOOW \in L(G)$ is the following: $a \Rightarrow^* WOb \Rightarrow^* WOOOOWd \Rightarrow WOOOOW$ as $a \Rightarrow Wb \Rightarrow WOb$ and $WOb \Rightarrow WOOb \Rightarrow WOOOc \Rightarrow WOOOOWd$. ∎

Grammars are classified according to the *Chomsky Hierarchy* [15]. Four types of grammars are distinguished; neither *Type* 0 (*unrestricted*) nor *type* 1 (*context sensitive*) grammars play a significant role in classical compiler technology. Instead, we take a closer look at type 2 and type 3 grammars. In *type* 2 or *context-free grammars*, all productions have the form $a \to v$ where $a \in V_n$ and $v \in (V_n \cup V_t)^*$. Context-free grammars form the basis for the parsing algorithms in Section 4.3. Lexical analysis is based on *type* 3 or *regular grammars.* In a (right-linear) regular grammar, all productions have the form $a \to Tb$ or $a \to T$ or $a \to \epsilon$, where $a, b \in V_n, T \in V_t$. The grammar in Example 4.4 is regular.

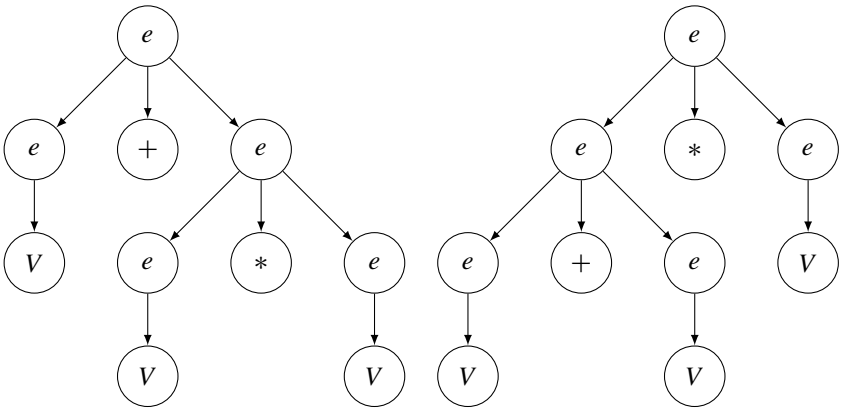


Figure 4.5. *Ambiguity leads to different ASTs.*

A grammar can be *ambiguous*, that is it can yield more than one parse tree for a given string of terminals. Consider, for example, a grammar

$$G = (\{V, L, N\}, \{e\}, e, \{e \to eLe, e \to eNe, e \to V\})$$

describing (a very limited set of) arithmetic expressions such as $a + b * c$ with linear $L = \{+, -\}$ and nonlinear binary operators $N = \{*, /\}$ applied to variables V. Two possible derivations of $VLVNV$ are

$$e \to eLe \to VLe \to VLeNe \to VLVNe \to VLVNV$$

and

$$e \to eNe \to eNV \to eLeNV \to eLVNV \to VLVNV.$$

The corresponding ASTs are shown in Figure 4.5. The first derivation is called *left-most* as the left-most nonterminal is always replaced first. In a *right-most* derivation (such as the second one) the right-most nonterminal is replaced first. Various combinations are possible when parsing larger expressions. Obviously, the second chain of derivations results in an incorrect order of evaluations of the two arithmetic operations. Numerically, the result of $a + b * c$ should be $a + (b * c)$ and not $(a + b) * c$. With both $+$ and $*$ being associative, the order of derivation of $a + b + c$ or $a * b * c$ is irrelevant in infinite precision arithmetic. Most compilers evaluate such expressions from the left as $(a + b) + c$ (or $(a * b) * c$). The same approach turns out to be numerically correct for subtraction and division.

We aim for grammars that are not ambiguous. One way to resolve ambiguity is to provide operator precedence information saying, for example, that $*$ and / have higher precedence than $+$ and $-$. Associativity can be resolved by specifying the order of evaluation, e.g. from left to right. This approach is taken in Section 4.3.

Abstract machines (also: *automata*) are used to build lexical and syntax analyzers. The two types of automata that are defined in the following form the basis for the recognition of words of a given language.

Definition 4.5. *A* Deterministic Finite Automaton *(DFA) is a quintuple*

$$A^d = (Q^d, \Sigma, \delta^d, q_0^d, F^d),$$

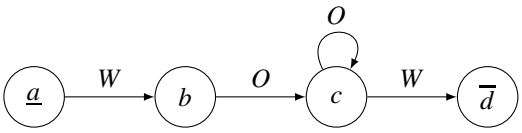


Figure 4.6. *A deterministic finite automaton.*

where

1. Q^d *is a finite set of states;*
2. Σ *is a finite alphabet (input symbols);*
3. δ^d *is a transition function* $(q_i^d, \sigma) \mapsto q_j^d$ *where* $\sigma \in \Sigma$ *and* $q_i^d, q_j^d \in Q^d$*;*
4. $q_0^d \in Q^d$ *is the start state;*
5. $vF^d \subseteq Q^d$ *is the set of final states.*

A string $s = \sigma_0\sigma_1\cdots\sigma_k$ is *accepted* by a DFA if there is a sequence of transitions $(q_{i_k}^d, \ldots, (q_{i_1}^d, (q_0^d, \sigma_0))\ldots)$ that takes the DFA from its start state to some final state.

Example 4.6 Figure 4.6 shows the relevant parts of a DFA $A^d = (Q^d, \Sigma, \delta^d, q_0^d, F^d)$ with $Q^d = \{a, b, c, d\}$, $\Sigma = \{W, O\}$, $q_0^d = a$, $F^d = \{d\}$, and where δ^d is defined as $(a, W) \mapsto b$, $(b, O) \mapsto c$, $(c, O) \mapsto c$, $(c, W) \mapsto d$. All remaining transitions lead into a dedicated error state that is not shown in Figure 4.6. The unique start state a is underlined. Overlining marks the final states (here only one, namely d). The DFA accepts nonempty sequences of O's of arbitrary length framed by two W's, for example, WOW and $WOOOOOW$. Input of W in state b results in a lexical error. ∎

Definition 4.7. *A* Nondeterministic Finite Automaton *(with ϵ-transitions; NFA) is a quintuple*

$$A^n = (Q^n, \Sigma, \delta^n, q_0^n, F^n),$$

where

1. Q^n *is a finite set of states;*
2. Σ *is a finite alphabet (input symbols);*
3. δ^n *is a transition function* $(q_i^n, \sigma) \mapsto Q_j^n$ *where* $\sigma \in \Sigma \cup \{\epsilon\}$ *and* $Q_j^n \subseteq Q^n$*;*
4. $q_0^n \in Q^n$ *is the start state;*
5. $F^n \subseteq Q^n$ *is the set of final states.*

A string $s = \sigma_0\sigma_1\cdots\sigma_k$ is accepted by a NFA if the result of some sequence of transitions $(q_{i_k}^n, \ldots, (q_{i_1}^n, (q_0^n, \sigma_0))\ldots)$ contains a final state.

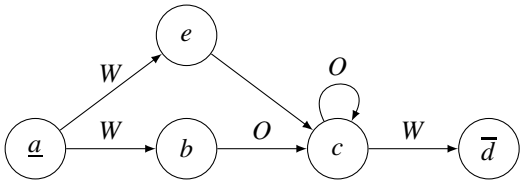


Figure 4.7. *A nondeterministic finite automaton (with an ϵ-transition (e,c)).*

Example 4.8 Figure 4.7 shows a NFA $A^n = (Q^n, \Sigma, \delta^n, q_0^n, F^n)$ with $Q^n = \{a,b,c,d,e\}$, $\Sigma = \{W,O\}$, $q_0^d = a$, $F^d = \{d\}$, and where δ^n is defined as $(a,W) \mapsto \{b,e\}$, $(e,\epsilon) \mapsto \{c\}$, $(b,O) \mapsto \{c\}, (c,O) \mapsto \{c\}, (c,W) \mapsto \{d\}$. The unlabeled arc denotes an ϵ-transition. Again, all remaining transitions lead into a dedicated error state. The NFA accepts WW in addition to all words accepted by the DFA in Figure 4.6. ∎

4.3 Lexical Analysis

Lexical analysis is performed by so-called *scanners*. It aims to group the sequence of input characters into substrings that belong to logical groups or *tokens*. The patterns to be matched by the scanner are described by regular grammars. The preferred way to specify regular grammars is via *regular expressions* (REs).

Definition 4.9. *REs are defined recursively as follows:*

- $\varnothing$ *is a RE.*
- ϵ *is a RE.*
- $\sigma \in \Sigma$ *is a RE.*
- *If a and b are REs, then $a|b$ is the RE that denotes the* union $L(a) \cup L(b) \equiv \{w | w \in L(a) \vee w \in L(b)\}$, *where $L(a)$ denotes the language defined by the RE a.*
- *If a and b are REs, then ab is the RE that denotes the* concatenation $L(a)L(b) \equiv \{vw | v \in L(a) \wedge w \in L(b)\}$.
- *If a is a RE, then $a*$ is the RE that denotes the* Kleene closure $L* \equiv \bigcup_{i=0}^{\infty} L^i$.
- *If a is a RE, then (a) is a RE that denotes the same language.*

Examples for REs are $0*|1$ or $(01)*|(10)*$. The scanner generator `flex` (see Section 4.3.4) uses an extended set of regular expressions. For example, $a^+ \equiv aa^*$, $a? \equiv \epsilon|a$, $a\{n\} \equiv \overset{(n \text{ times})}{a \cdots a}$, and $a\{n,m\} \equiv \overset{(n \text{ times})}{a \cdots a} \mid \cdots \mid \overset{m \text{ times}}{a \cdots a}$ for $n,m \in IN$, $n < m$. Refer to the `flex` manual[9] for further information.

[9] `http://flex.sourceforge.net/manual/`

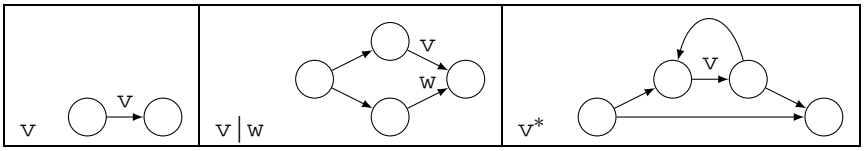


Figure 4.8. *From regular expressions* v *and* w *to nondeterministic finite automata by Thompson's construction.*

4.3.1 From RE to NFA

An NFA can be constructed for any set of REs. Figure 4.8 shows some of the basic building blocks of the recursive construction algorithm that is due to Thompson [57]. Automata that recognize the grammar symbols in a given regular expression consist of two nodes (local start and final states) connected by an arc that is labeled with the respective symbol. Concatenation vw is represented by connecting the local final state of the automaton that recognizes v with the local start state of the automaton for w via an ϵ-arc. Union and Kleene closure are constructed as shown in Figure 4.8. Unlabeled arcs are ϵ-arcs. See also Example 4.10.

4.3.2 From NFA to DFA with Subset Construction

Nondeterministic automata are often not the best choice for language recognition as they may require backtracking in order to try all possible sequences of transitions. A deterministic approach promises to be superior in most cases. Hence, we are looking for a method that allows us to transform a given NFA into a DFA. The corresponding *subset construction* algorithm [2] may result in a DFA whose size is exponential in that of the original NFA. In this case a backtracking algorithm based on the latter may be the preferred method. Most of the time, subset construction yields useful results.

We consider the construction of a DFA $A^d = (Q^d, \Sigma, \delta^d, q_0^d, F^d)$ from a NFA $A^n = (Q^n, \Sigma, \delta^n, q_0^n, F^n)$ by the subset construction algorithm. Let ϵ-closure(q) denote all states reachable from q by ϵ in A^n. Let move (q, σ) be the set of all states reachable from q by σ in A^n. The algorithm proceeds as follows:

- $q_0^d := \epsilon\text{-closure}(q_0^n)$
- $Q^d := q_0^d$
- $\forall q_i^d \in Q^d \; \forall \sigma \in \Sigma$:
  - $q_j^d := \epsilon\text{-closure}(\text{move}(q_i^d, \sigma))$
  - $Q^d = Q^d \cup q_j^d$
  - $\delta^d = \delta^d \cup ((q_i^d, \sigma) \mapsto q_j^d)$

The subset construction algorithm is a fixed-point computation. It terminates as soon as both Q^d and δ^d have reached their maximum sizes. Termination follows immediately from the fact that the number of distinct subsets of the finite set Q^n is finite. Moreover, the number

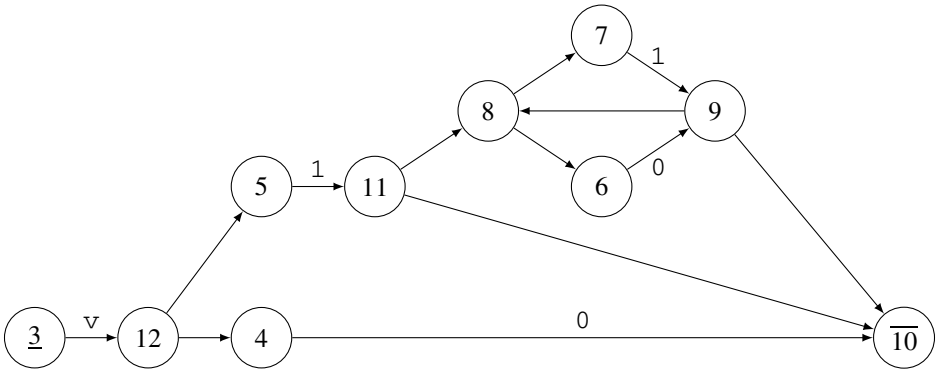


Figure 4.9. *NFA for* `v(0|1(0|1)*)`.

Table 4.1. *ϵ-closure(move(s,{v,0,1})) for $s \in Q^n$ (see NFA in Figure 4.9).*

DFA	NFA	v	0	1
$\underline{1}$	$\underline{3}$	{12,4,5}		
6	{12,4,5}		{10}	{11,10,8,6,7}
$\overline{7}$	$\{\overline{10}\}$			
$\overline{8}$	$\{11,\overline{10},8,6,7\}$		{9,10,8,6,7}	{9,10,8,6,7}
$\overline{9}$	$\{9,\overline{10},8,6,7\}$		{9,10,8,6,7}	{9,10,8,6,7}

of edges carrying distinct labels that are drawn from the finite set Σ must be finite for any source state in Q^d.

Example 4.10 The NFA obtained by Thompson's construction applied to the regular expression `v(0|1(0|1)*)` is shown in Figure 4.9. The numbering of the states corresponds to that used by the scanner generator `flex`; see Section 4.3.4. Section 4.5.3 shows only the relevant part of the NFA that is generated by `flex`. Subset construction yields the transitions in Table 4.1. The corresponding DFA is shown in Figure 4.10. Its unique start state is the ϵ-closure of the start state of the NFA (3). Three final (also *accepting*) states (7, 8, and 9) are induced by the final state of the NFA (10). ∎

In some cases, NFAs can be more expressive than their deterministic counterparts. Refer to [40] for further details.

4.3.3 Minimization of DFA

There is always a unique minimum state DFA for any regular language. It is constructed recursively by verifying if states are distinguishable by transitions on certain input symbols. In a first step, all accepting states are separated from the non-accepting states and the error state yielding three initial *groups.* A state is *distinguishable* from another state if on input of some symbol the transitions lead into different groups. Otherwise, the two states are

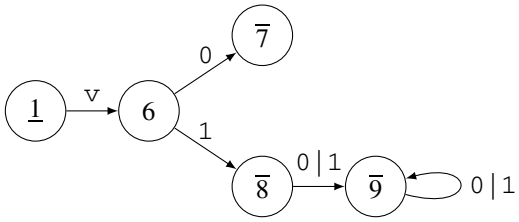


Figure 4.10. *DFA for* `v(0|1(0|1)*)`.

indistinguishable with respect to the given partitioning. This procedure is continued for as long as there are distinguishable states in one and the same group. The transitions of the resulting minimal DFA are derived from the original DFA by considering groups as states. See [2] for further details on the minimization of DFAs.

Example 4.11 In order to minimize the number of states in the DFA in Figure 4.10, we start with the partitioning

$$\{1,6\},\{7,8,9\}$$

of the non-accepting and accepting states, respectively. The third partition contains only the error state and is omitted. The two states 1 and 6 are distinguished by the input symbol v (or 0, or 1) as the transition from 1 leads to $\{1,6\}$, whereas the transition from 6 leads into the error state yielding the partitioning

$$\{1\},\{6\},\{7,8,9\}.$$

States 7 and $\{8,9\}$ are distinguishable through 0|1, both of which lead from 7 to the error state while the transition from either 8 or 9 is to $\{7,8,9\}$. The new partitioning becomes

$$\{1\},\{6\},\{7\},\{8,9\}.$$

Finally, states 8 and 9 are found to be indistinguishable as transition on v is to the error state and 0|1 lets the DFA remain in $\{8,9\}$.

The transitions of the minimal DFA are derived from the original DFA by considering partitions as states (nodes):

	v	0	1
$\underline{\{1\}}$	{6}		
$\overline{\{6\}}$		{7}	{8,9}
$\overline{\{7\}}$			
$\overline{\{8,9\}}$		{8,9}	{8,9}

The start state of the minimal DFA is the group that contains the original start state, that is, $\{1\}$. The accepting states are those groups that contain at least one accepting state from the original DFA, that is $\{7\}$ and $\{8,9\}$. A graphical representation of the minimal DFA is shown in Figure 4.11. ∎

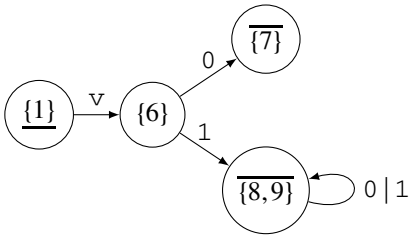


Figure 4.11. *Minimal DFA for Figure* 4.10.

4.3.4 `flex`

We start this section with a quote from the `flex` manual:[10] "`flex` is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The `flex` program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. `flex` generates as output a C source file, `lex.yy.c` by default, which defines a routine `yylex()`. This file can be compiled and linked with the `flex` run time library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code."

Listing 4.1 shows a `flex` input file for the RE `v(0|1(0|1)*)`. The file consists of three sections separated by %%. REs are used to define tokens in the first section. The second section contains rules for actions to be taken when matching tokens. In our simple example, we read strings (their respective ends marked by ''\n'') from standard input for as long as they match the definition of variables given by the regular expression in the first section of the `flex` input file. We stop the scanner as soon as a string that is not a variable is typed in. The third section contains user-defined routines—just the main function in this example.

Listing 4.1. `flex` *input file.*

```
variable        v(0|1(0|1)*)

%%

{variable}      { }
.               { return 0; }

%%

int main()
{
  yylex();
  return 0;
}
```

[10] `http://flex.sourceforge.net/manual/`

`flex` generates an NFA that contains the NFA in Figure 4.9. The corresponding DFA contains the DFA in Figure 4.10. Running `flex` with the `-T` option generates diagnostic output containing information on both automata. As an example, we consider the diagnostic output generated for Listing 4.1. It starts with an enumeration of the REs that describe the tokens to be recognized by the scanner (here only rule 1) in addition to the remaining single-character tokens (rule 2) and the special end (of string) marker ''\n'' (rule 3).

```
1       (v(0|1(0|1)*))
2       .
3       End Marker
```

Note that the scanner generated by `flex` recognizes arbitrary input. In the worst case, single characters are matched individually by using rule 2. Potentially desired error handling needs to be implemented by the user in the form of appropriate actions.

The relevant part of the NFA that corresponds to the three rules is the following:

```
***** beginning dump of nfa with start state 19
...
state #    3    118:    12,    0
state #    4     48:    10,    0
state #    5     49:    11,    0
state #    6     48:     9,    0
state #    7     49:     9,    0
state #    8    257:     6,    7
state #    9    257:     8,   10
state #   10    257:     0,    0  [1]
state #   11    257:     8,   10
state #   12    257:     4,    5
state #   13    257:     1,    3
state #   14     -1:    15,    0
state #   15    257:     0,    0  [2]
state #   16    257:    13,   14
state #   17     -2:    18,    0
state #   18    257:     0,    0  [3]
state #   19    257:    16,   17
***** end of dump
```

Each state is followed by an integer that encodes an input symbol. The next two columns contain the corresponding successor states. For example, in state 3, the input of v (ASCII code 118) leads to state 12. Zeros denote undefined states. From the start state 19, two ϵ-transitions (encoded as 257) lead into states 16 and 17, respectively. Reading the end marker (encoded as -2) in state 17 yields a transition into the accepting state 18. Acceptance is indicated by the number of the matched REs given in square brackets. All single character tokens except for v (this set is represented by -1) are accepted by state 15 that is reached from state 16 via state 14. Further ϵ-transitions take the NFA from state 16 via state 13 to state 3. Input of v (ASCII 118) leads to state 12 and hence, to states 4 and 5 via respective ϵ-transitions. Acceptance of v0 is represented by the transition from state 4 to state 10 on input of 0 (ASCII 48). Closer inspection of states 5–11 and of the corresponding transitions identifies the spanned sub-NFA as one that accepts strings described by the regular expression `v(0|1(0|1)*)`.

Subset construction yields the DFA that is listed in the diagnostic output (edited slightly for brevity) as follows:

```
state # 1:
        1       4
        2       5
        3       4
        4       4
        5       6
state # 4:
state # 5:
state # 6:
        3       7
        4       8
state # 7:
state # 8:
        3       9
        4       9
state # 9:
        3       9
        4       9
state # 4 accepts: [2]
state # 5 accepts: [3]
state # 6 accepts: [2]
state # 7 accepts: [1]
state # 8 accepts: [1]
state # 9 accepts: [1]
```

All 256 of the 8-bit characters are grouped into equivalence classes used to specify the transitions in the DFA. For example, ''\n''=2, ''0''=3, ''1''=4, ''v''=5, and the group of the remaining 252 characters is encoded as 1. Reading ''v'' in start state 1 yields a transition to state 6. From there, ''0'' and ''1'' take the DFA to states 7 and 8, respectively. State 7 accepts ''v0'' as a special case of a token described by the first RE v (0|1(0|1) $*$). State 8 accepts ''v1''. Further instances of ''0'' and ''1'' lead from state 8 to state 9 followed by leaving the DFA in state 9. The latter accepts all strings that consist of at least three characters and that can be derived from the RE v (0|1(0|1) $*$). This part of the DFA is shown in Figure 4.10. States that accept "\n" (state 5) and all remaining single character tokens (states 4 and 6) are not shown in Figure 4.10. For example, if the characters read in state 1 are either ''0'', ''1'', or any of the characters in equivalence class 1, then acceptance is due to the second rule in state 4. Moreover, if ''v'' is read and is not followed by ''0'' or ''1'', then it is accepted as a single character token in state 6.

4.4 Syntax Analysis

The purpose of syntax analysis in derivative code compilers is twofold:

1. The syntactic correctness of the given program is verified for a given syntax definition in the form of a grammar (see Sections 4.4.1 and 4.4.2). Single-pass derivative code compilers can be build for certain languages by suitable extensions of the syntax analysis algorithms (see Section 4.5).

2. An intermediate representation (IR) of the program is built (see Section 4.6). The IR is used for static program analysis as well as for semantic transformation and unparsing; that is, the generation of the desired output.

Numerous semantic tests are performed by standard compilers including, for example, type checking. In this book, we assume that all input programs are guaranteed to be both syntactically and semantically correct. Users of derivative code compilers must have access to native compilers. They can be used to verify semantic soundness of a syntactically correct input program. The derivative code compiler front-end can thus be kept simpler. The emphasis is put on the domain-specific issues instead of well-known and widely studied standard compiler problems. The development of a robust compiler for a modern programming language such as C or C++ is a highly complex and challenging project. It is clearly beyond the scope of this book. Our intention is not to present an in-depth discussion of syntax analysis techniques, thus, merely repeating material that has been available in the literature for many years. See, for example, [2]. Instead, we focus on an intuitive description of the fundamental concepts based on examples. Our goal is to provide the essential amount of understanding that enables the reader to follow the ideas behind the generation of derivative code by semantic source code transformation.

We focus on a very small subset of C++ that is still rich enough to capture the fundamental issues in derivative code compiler development and use. For example, version 0.9 of `dcc` (see Chapter 5) accepts input consisting of several subroutines containing branches and loops and with arithmetic performed on scalars as well as on multi-dimensional arrays. For the sake of brevity and conciseness of the following discussions of various parsing algorithms, it turns out to be advantageous to impose further syntactic restrictions. We consider variants of a Simple Language (SL). SL programs consist of (possibly nested) sequences of assignments, loops, and branches. All variables are assumed to be floating-point scalars. Arithmetic is performed using linear (e.g. $+$), nonlinear (e.g. $*$), and relational (e.g. $<$) operators and intrinsic functions (e.g. sin). An example of an SL program is the following:

```
if(x<y) {
  x=x*y;
  while (y<x) {
    x=sin(x+y*3);
  }
}
```

SL programs need to be wrapped into a subroutine with an appropriate signature in order to be compiled by a native C/C++ compiler. For example,

```
void f(double& x, double y) {
  // SL code goes here
}
```

As a special case, we consider sequences of assignments in SL, such as

```
x=x*y;
x=sin(x+y*3);
```

that are formally defined as follows.

Definition 4.12. *A* straight-line SL program (an SL^2 program) *is a sequence of statements described by the grammar* $G = (V_n, V_t, P, s)$ *with nonterminal symbols*

$$V_n = \left\{ \begin{array}{ll} s & \textit{(sequence of statements)} \\ a & \textit{(assignment)} \\ e & \textit{(expression)} \end{array} \right\},$$

terminal symbols

$$V_t = \left\{ \begin{array}{ll} V & \textit{(program variables)} \\ C & \textit{(constants)} \\ F & \textit{(unary intrinsic)} \\ L & \textit{(linear binary arithmetic operator)} \\ N & \textit{(nonlinear binary arithmetic operator)} \\ ; \ ) \ (= & \textit{(remaining single character tokens)} \end{array} \right\},$$

start symbol s, *and production rules*

$$P = \left\{ \begin{array}{llllll} (P1) & s:a & (P2) & s:as & (P3) & a:V=e; \\ (P4) & e:eLe & (P5) & e:eNe & (P6) & e:F(e) \\ (P7) & e:V & (P8) & e:C & & \end{array} \right\}.$$

Note that G has been made ambiguous for the purpose of illustrating certain fundamental aspects of syntax analysis. For example, the word $V = VNVLV;$ has two feasible right-most derivations

$$\begin{aligned} s \to a \to V = e; &\to V = eLe; \to V = eLV; \\ &\to V = eNeLV; \to V = eNVLV; \to V = VNVLV; \end{aligned}$$

and

$$\begin{aligned} s \to a \to V = e; &\to V = eNe; \to V = eNeLe; \\ &\to V = eNeLV; \to V = eNVLV; \to V = VNVLV; \end{aligned}$$

as previously discussed. Moreover, the missing handling of operator precedence may result in numerically incorrect code. These issues will be dealt with. Let us first recall some classical work in the field of syntax analysis, which every developer of (derivative code) compilers should be familiar with.

All context-free grammars can be converted into *Chomsky normal form* [15], where all productions have one of the following formats:

$$a:bc \qquad a:A \qquad s:\epsilon$$

with $a, b, c, s \in V_n$, $A \in V_t$ and where s is the start symbol. s is not permitted to occur on the right-hand side of a production if the production $s:\epsilon$ is present.

For context-free languages in Chomsky normal form, there is a parsing algorithm due to Cocke and Schwartz [16], Younger [62], and Kasami [43] that is built on the principles of dynamic programming [8]. Hence, the computational complexity of the so-called CYK-algorithm is cubic in the length of the input, which may be infeasible for complex code. More efficient (ideally linear in time and memory requirement) parsing algorithms have been developed. They are discussed later.

$$\begin{bmatrix} e & \underline{d} & e & \overline{c} & e \\ & n & \varnothing & \varnothing & \varnothing \\ & & e & c & \underline{e} \\ & & & l & \varnothing \\ & & & & \overline{e} \end{bmatrix}$$

Figure 4.12. *CYK-parsing of $VNVLV$.*

Transformation of a context-free grammar into Chomsky normal form requires essentially four steps:

1. The grammar is transformed into *weak Chomsky normal form* by introduction of productions $a : A$ for all $A \in V_t$ and by substitution of a for A in all right-hand sides of productions that contain A.

2. In all resulting productions, right-hand sides of length ≥ 3 are shortened recursively by introducing auxiliary nonterminals for two consecutive nonterminals and by corresponding substitution. For example, $a : bcd$ might become $\{a : ed,\ e : bc\}$.

3. All productions of the form $a : \epsilon$, where $a \neq s$, are removed as well as all occurrences of a in right-hand sides of other productions.

4. Chain rules of the form $a : b$ are eliminated by replacing all rules $b : B$ with $a : B$.

As an example, we consider a sub-grammar of the grammar defined in Definition 4.12. Let $G = (V_n, V_t, P, e)$ with $V_n = \{e\}$, $V_t = \{L, N, V\}$, and $P = \{e : eLe|eNe|V\}$. We introduce $l : L$, $n : N$, $c : el$, and $d : en$ to get

$$P_{\text{CNF}} = \{e : ce|de|V, c : el, d : en, l : L, n : N\}$$

in Chomsky normal form. Instead of stating the CYK-algorithm formally, we use this example to illustrate its rather intuitive behavior.

Consider the word $VNVLV$. The algorithm combines tabulated information about possible derivations of the $\binom{6}{2} = 15$ substrings in order of increasing length to find a derivation of the entire word. It first finds $e : V$, $l : L$, and $n : N$ as feasible productions of the five single-letter words that yield the diagonal entries of the dynamic programming table in Figure 4.12. In a next step, the four two-letter words en, ne, el, and le are considered. Two of them can be reduced to d and c using $d : en$ and $c : el$, respectively. They form the first super-diagonal in Figure 4.12. The three three-letter words VNV, NVL, and VLV are parsed either as a single-letter word followed by a two-letter word or as a two-letter word followed by a single-letter word. For VNV, the parsing algorithm attempts to reduce e and de succeeding only for the latter by using $e : de$. Similarly, VLV can be parsed as a two-letter word followed by a single-letter word by using $e : ce$. The second three-letter substring NVL cannot be reduced as neither nc nor l are right-hand sides of productions in P_{CNF}. The algorithm proceeds by considering the two four-letter words $VNVL$ and $NVLV$ as concatenations of a single- and a three-letter word, of two two-letter words, and of a three- and a single-letter word. In a final step the entire word is treated as a concatenation of words of length one and four ($V + NVLV$), two and three ($VN + VLV$), three and two ($VNV + LV$), and four and one ($VNVL + V$).

As usual in dynamic programming, the key observation is the presence of overlapping subproblems whose solutions can be tabulated and looked up in constant time. The cubic (in

the length of the input word) computational complexity is paid for with a quadratic memory requirement. The word is verified as an element of the language generated by the grammar G. It is successfully derived from the start symbol e. Ambiguity of G results in two distinct derivations marked with overset, resp., underset, bars in Figure 4.12. The corresponding alternatives are

$$\begin{aligned} e &\to ce \to cV \to elV \to eLV \to deLV \\ &\to dVLV \to enVLV \to eNVLV \to VNVLV \end{aligned}$$

and

$$\begin{aligned} e &\to de \to dce \to dcV \to delV \to deLV \\ &\to dVLV \to enVLV \to eNVLV \to VNVLV. \end{aligned}$$

4.4.1 Top-Down Parsing

Conceptually, top-down parsing is about the construction of parse trees for a sequence of tokens starting from the root and working down toward the leafs. Top-down parsers read the input from Left to right and generate a Left-most derivation. Hence, they are also referred to as LL parsers. The following discussion of top-down parsing techniques is kept very brief as our main focus is on bottom-up parsers. Still we feel that some degree of intuitive understanding of top-down parsers belongs to the "tool-box" of potential authors of derivative code compilers.

Predictive recursive descent parsers such as generated by ANTLR [51] contain a subroutine for each nonterminal symbol. They select a suitable production rule based on a lookahead of length k (LL(k) parsers) on the incoming token stream. A lookahead of $k = 1$ is sufficient for many programming languages.

Example 4.13 A basic recursive descent parser for the partial SL^2 grammar $G = (V_n, V_t, P, s)$ with $V_n = \{e\}$, $V_t = \{F, V, (,)\}$, start symbol $s = e$, and production rules $P = \{e : F(e)|V\}$, such that $F =' a'$ and $V =' b'$ requires a single recursive subroutine for the nonterminal e.

```
bool e(ifstream& i) {
  char c;
  i >> c;
  if (c=='a') {
    i >> c; if (c!='(') return false;
    if (!e(i)) return false;
    i >> c; if (c!=')') return false;
  }
  else if (c!='b') return false;
  return true;
}
```

It parses strings representing nested function calls, such as a(a(a(b)))). No lookahead is required. ∎

Left-Recursion Production rules of the form $e : e\alpha$, where $e \in V_n$ and $\alpha \in (V_n \cup V_t)^*$, prevent recursive descent parsers from terminating. Upon entry, the parsing subroutine for a calls itself recursively. Fortunately, such left-recursion can be eliminated by a simple transformation of the grammar [2]. Production rules of the form

$$e : e\alpha|\beta,$$

where $\beta \in (V_n \cup V_t)^*$, are replaced with

$$\begin{aligned} e &: \beta t \\ t &: \alpha t|\epsilon \end{aligned}$$

followed by adding the auxiliary symbol $t \notin V_n$ to the set of nonterminals. This process is repeated recursively for as along as left-recursive productions still exist. For example, in order to eliminate left-recursion from the grammar in Definition 4.12, we transform rules P4–P8, that is

$$e : eLe|eNe|F(e)|V|C,$$

first into

$$\begin{aligned} e &: eNet|F(e)t|Vt|Ct \\ t &: Let|\epsilon \end{aligned}$$

followed by

$$\begin{aligned} e &: F(e)tf|Vtf|Ctf \\ f &: Netf|\epsilon \\ t &: Let|\epsilon. \end{aligned}$$

New nonterminal symbols t and f are introduced. The corresponding recursive descent parser becomes nondeterministic. For example, the string $VNVLV$ is processed as

$$e \to Vt^1f^1 \to Vf^1 \to VNe^1t^2f^2 \to VNVt^3f^3t^2f^2$$

where superscripts enumerate the calls of the parsing routines associated with the respective nonterminal symbols. There are several feasible alternatives for deriving the missing substring LV from $t^3f^3t^2f^2$. For example, both

$$\begin{aligned} VNVt^3f^3t^2f^2 &\to VNVLe^2t^4f^3t^2f^2 \to VNVLVt^5f^4t^4f^3t^2f^2 \\ &\to VNVLVf^4t^4f^3t^2f^2 \to VNVLVt^4f^3t^2f^2 \\ &\to VNVLVf^3t^2f^2 \to VNVLVt^2f^2 \\ &\to VNVLVf^2 \to VNVLV \end{aligned}$$

and

$$\begin{aligned} VNVt^3f^3t^2f^2 &\to VNVf^3t^2f^2 \to VNVt^2f^2 \\ &\to VNVLe^2t^4f^2 \to VNVLVt^5f^4t^4f^2 \\ &\to VNVLVf^4t^4f^2 \to VNVLVt^4f^2 \\ &\to VNVLVf^2 \to VNVLV \end{aligned}$$

represent valid derivations. The former results from the greedy approach that is used, for example, in ANTLR to resolve nondeterminism. A lookahead of length one identifies L as the next token to be read. The algorithm picks the corresponding production rule $t : Let$. In general, a lookahead of length k may be required to make this choice unique.

4.4.2 Bottom-Up Parsing

Conceptually, the term bottom-up parsing refers to the construction of parse trees for a sequence of tokens starting from the leafs and working up toward the root. We consider shift-reduce parsers as a general approach to bottom-up parsing. The key decisions are about when to *reduce* and what production to apply. A *reduction* is defined as the reverse of a step in the derivation. A *handle* is a substring that matches the right-hand side of some production.

A basic shift-reduce parser uses a stack to hold grammar symbols, and it reads from an input buffer (left to right) holding the string to be parsed. The stack is empty initially. Symbols are read and pushed onto the stack (*shift* – S) until the top-most symbols on the stack form a handle. The handle is replaced by the left-hand side of the corresponding production (*reduce* – R). This process is repeated iteratively until the string has been parsed successfully (ACCEPT) or until an error has occurred (ERROR). A successful parsing procedure is characterized by an empty input buffer and a stack that contains only the start symbol. It can be shown that for any right-most derivation handles always appear on top of the stack, never inside. Shift-reduce parsing may lead to conflicts where the next action is not determined uniquely. We distinguish between *shift-reduce* and *reduce-reduce* conflicts.

Example 4.14 Consider the same context-free grammar as in Example 4.13. A shift-reduce parser processes the string $F(F(V))$ by shifting tokens from left to right onto a stack with reductions performed for handles occurring on top of the stack:

	STACK	INPUT	ACTION
1		$F(F(V))$	S
2	F	$(F(V))$	S
3	$F($	$F(V))$	S
4	$F(F$	$(V))$	S
5	$F(F($	$V))$	S
6	$F(F(V$	$))$	R $(e : V)$
7	$F(F(e$	$))$	S
8	$F(F(e)$	$)$	R $(e : F(e))$
9	$F(e$	$)$	S
10	$F(e)$		R $(e : F(e))$
11	e		ACCEPT

The addition of assignments yields an extended grammar $G = (V_n, V_t, s, P)$, where $V_n = \{a, e\}$, $V_t = \{V, F, (,), =, ;\}$, $s = a$, and

$$P = \begin{cases} a : V = e; \\ e : F(e) | V. \end{cases}$$

Shift-reduce parsing of $V = V;$ encounters two shift-reduce conflicts that can be resolved by taking a *lookahead* of one token into account.

	STACK	INPUT	ACTION
1		$V = V;$	S
2	V	$= V;$	S (or R $(e : V)$?)
3	$V =$	$V;$	S
4	$V = V$	;	R $(e : V)$ (or S?)
5	$V = e$	;	S
6	$V = e;$		R $(a : V = e;)$
7	a		ACCEPT

The lookahead token $=$ is feasible in line 2 as it follows V in the first production rule. Hence, a shift is performed instead of the potential reduction. Formal approaches to making this decision are discussed in the remainder of this section. Similarly, the lookahead token; is not feasible in line 4 yielding the reduction based on $e : V$.

Reduce-reduce conflicts can occur in languages such as Fortran, where `a(i)` can denote both an array access and a univariate function call. Additional information on the kind of tokens is required to resolve these conflicts. ∎

4.4.3 A Simple LR Language

The parser generator `bison` generates LR parsers (shift-reduce parsers that read the input from Left to right and that generate a Right-most derivation; see [2] for details) automatically for a suitable given grammar. The simple programming language SL allows us to capture many features of numerical code. Its straight-line version SL^2 that covers sequences of assignments only is used throughout this section for the derivation of a *simple LR* (SLR) parser. The parser generator `bison` can take operator precedence and information on the order of resolution of associativity into account. To illustrate these features, a modified SL grammar is considered in Section 4.4.4.

The design of a grammar that can be parsed by a given technique (e.g. SLR) is typically not an easy task. A reasonable approach follows a top-down strategy. For example, starting from the top-level structure (nonempty sequence of assignments) one descents into the definition of single assignments followed by the syntactical description of arithmetic expressions that are allowed to appear on the right-hand side. Such reasoning may result in the following grammar for SL^2.

Definition 4.15. *An* SL^2 program *is a sequence of assignments described by the grammar* $G = (V_n, V_t, P, s)$ *with nonterminal symbols*

$$V_n = \left\{ \begin{array}{llllll} s & \textit{(sequence of assignments)} & a & \textit{(assignment)} & & \\ e & \textit{(expression)} & t & \textit{(term)} & f & \textit{(factor)} \end{array} \right\},$$

terminal symbols

$$V_t = \left\{ \begin{array}{ll} V & (\textit{program variables}) \\ C & (\textit{constants}) \\ F & (\textit{unary intrinsic}) \\ L & (\textit{linear binary arithmetic operator}) \\ N & (\textit{nonlinear binary arithmetic operator}) \\ ; \;) \; (\; = & (\textit{remaining single character tokens}) \end{array} \right\},$$

start symbol s, and production rules

$$P = \left\{ \begin{array}{llllllll} (P1) & s:a & (P2) & s:as & (P3) & a:V=e; & (P4) & e:eLt \\ (P5) & e:t & (P6) & t:tNf & (P7) & t:f & (P8) & f:F(e) \\ (P9) & f:V & (P10) & f:C \end{array} \right\}.$$

Ambiguity present in Definition 4.12 is removed. Operator precedence (N over L) as well as rules for resolving associativity are built in explicitly.

So far, shift-reduce parsers have been introduced as algorithms that push symbols onto a stack followed by reducing handles on top of the stack to left-hand sides of suitable production rules. Alternatively, the states of the parsing algorithm can be described by *configurations* of production rules as follows.

A production rule with a right-hand side of length k yields $k+1$ configurations. For example, we get

$$e:.tNf \quad e:t.Nf \quad e:tN.f \quad e:tNf.$$

for production rule $P6$. The dot is used to mark how much of the right-hand side of this production rule has already been processed (is located on top of the stack). The right-hand side of the *initial configuration* of a production rule starts with the dot. For example, the configuration $e:t.Nf$ represents a state of the parser, where t is located on top of the stack and N is a feasible (ensuring syntactical correctness of the input) terminal symbol to be shifted next. Moreover, syntactical correctness of the input requires that a leading substring of the input that follows N can be reduced to f.

The *closure of a set of configurations* is a set of configurations (also: configurating set). It is built recursively by adding the initial configurations of all production rules for all nonterminal symbols that immediately follow the dot until no further new configuration can be added. For a grammar $G = (V_n, V_t, P, s)$ and a given set of items I, the closure of I is built as follows:

REPEAT
  $\forall$ [a: $\alpha.b\beta$] $\in I$
    $\forall$ b: $\gamma \in P$
      $I = I \cup$ [b: $.\gamma$]
UNTIL fixpoint reached

α, β, and γ denote arbitrary strings over $V_n \cup V_t$ while $a, b \in V_n$. For example, the closure of $f : F(.e)$ is

$$\text{Closure}(I) = \left\{ \begin{array}{l} f:F(.e) \\ e:.eLt \mid .t \\ t:.tNf \mid .f \\ f:.F(e) \mid .V \mid .C \end{array} \right\}.$$

As a first step the initial configurations of both production rules for e are added to $f : F(.e)$. We use the more compact notation $e : .eLt \mid .t$ instead of $\{e : .eLt,\ e : .t\}$. The nonterminal symbol t can be obtained by reduction using the two production rules $t : tNf$ and $t : f$. Hence, both initial configurations are added to Closure(I). Similarly, the initial configurations of the three production rules for f are added, which completes the closure operation. Termination follows immediately from the finite number of configurations over the finite set of production

rules. The closure operation yields all feasible paths to the current state of the parser. For example, if the current state is defined by the (closure of) the configuration $f : F(.e)$, then anything but a succeeding reduction to e results in a syntax error. This reduction can be obtained by reducing eLt or t to e. Recursively, this reduction must be preceded by reductions to e or t, and so forth. This closure operation captures all possible terminal symbols that are allowed to be read in the current state of the parser, for example, F, V, or C.

The configurating sets (together with a dedicated error state) define the vertices of the *characteristic automaton* (also: $LR(0)$ automaton)

$$A_{LR(0)} = (V_{LR(0)}, E_{LR(0)}).$$

The labeled edges (also: *transitions*) are defined as

$$E_{LR(0)} = \{((i, j), v) : [a : \alpha.v\beta] \in i \text{ and } [a : \alpha v.\beta] \in j\},$$

where $a \in V_n$, $\alpha, \beta \in (V_n \cup V_t)^*$, and $v \in V_n \cup V_t$. A stack is used to store the history of transitions of the characteristic automaton. To illustrate this procedure, let the characteristic automaton be in a state j defined by the corresponding configurating set. Reading a new terminal symbol B from the input results in the *forward transition* to state k defined as the closure of a configuration $[a : \alpha B.\beta] \in k$ obtained from $[a : \alpha.B\beta] \in j$. The index j of this *shift state* is pushed onto the stack. If $[a : \alpha.B\beta] \notin j$, then the transition leads into the dedicated error state and a syntax error is reported.

A *reduce state* j contains a *final configuration* $b : \gamma.$ where the dot appears at the end of the right-hand side of the production rule. When the parser reaches state j, a reduction of γ to b is performed unless j contains another configuration $a : \gamma.A\beta$ and the next terminal symbol to be read from the input (the lookahead) is A. SLR parsers perform the shift operation in this case. Reduction yields a *backward transition* to state k, where k is the $|\gamma|'$s element on the stack and where $|\gamma|$ denotes the length of the string, that is, the number of symbols in γ. The part of the parsing history that yielded the handle γ becomes obsolete. Hence, the top $|\gamma|$ elements can be removed from the stack. A forward transition from k to k' follows if k contains a configuration $[a : \alpha.b\beta] \in k$; a syntax error is reported otherwise. Conflicts that cannot be resolved using this technique identify SLR parsers as infeasible for the recognition of the language $L(G)$ that is generated by the given grammar G.

The characteristic automaton of SL^2 is shown in Figure 4.13. An auxiliary production rule

$$(P0) \quad \$accept : s\$end$$

is introduced to mark the end of the input to be parsed with $\$end$. The characteristic automaton becomes $A_{LR(0)} = (V_{LR(0)}, E_{LR(0)})$ with states

$$\begin{aligned}
V_{LR(0)} = \{ & \\
& 0 : [\ \ \$accept : .s\$end \quad s : .a \mid .as \quad a : .V = e; \ \ ], \\
& 1 : [\ \ a : V. = e; \ \ ], \\
& 2 : [\ \ \$accept : s.\$end \ \ ], \\
& 3 : [\ \ s : a. \mid a.s \quad s : .a \mid .as \quad a : .V = e; \ \ ], \\
& 4 : [\ \ a : V = .e; \quad e : .eLt \mid .t \quad t : .tNf \mid .f \quad f : .F(e) \mid .V \mid .C \ \ ], \\
& 5 : [\ \ \$accept : s\$end. \ \ ],
\end{aligned}$$

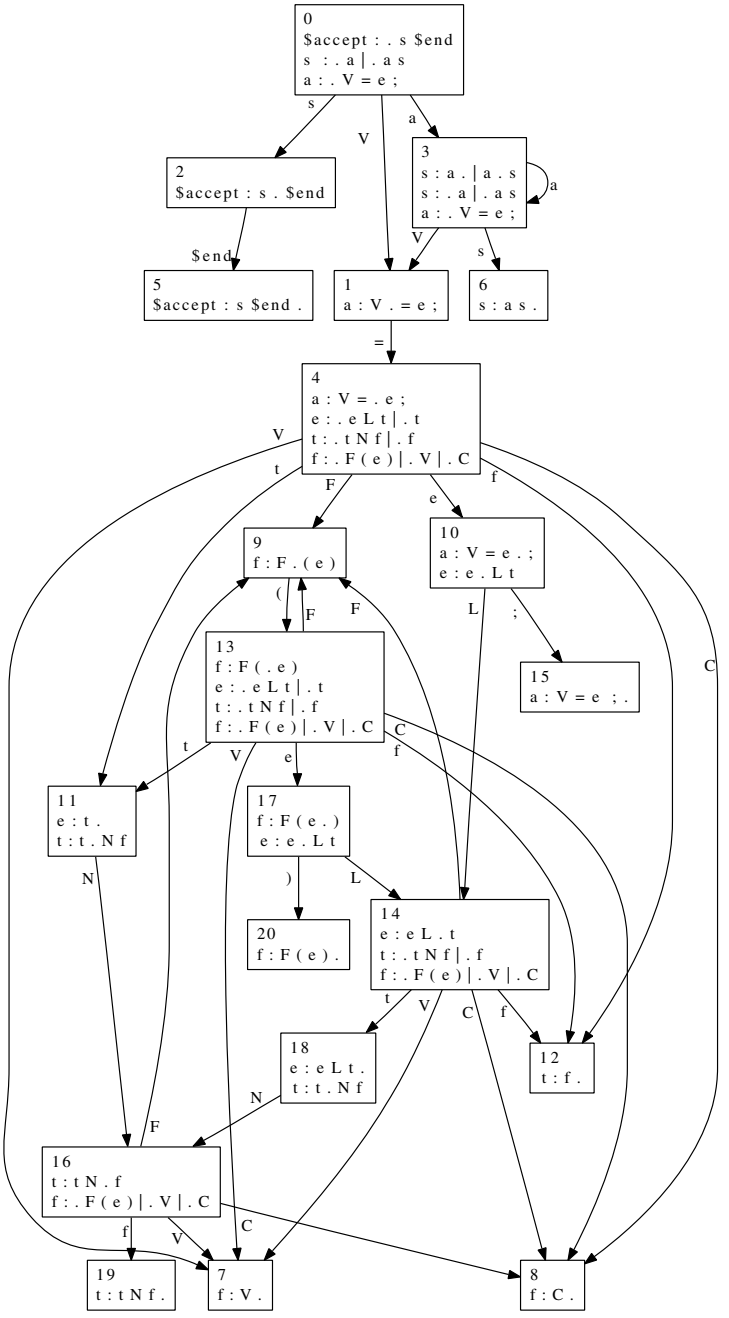


Figure 4.13. *Characteristic automaton for SL^2 programs.*

$$
\begin{array}{rl}
6: & [\quad s : as. \quad], \\
7: & [\quad f : V. \quad], \\
8: & [\quad f : C. \quad], \\
9: & [\quad f : F.(e) \quad], \\
10: & [\quad a : V = e.; \quad e : e.Lt \quad], \\
11: & [\quad e : t. \quad t : t.Nf \quad], \\
12: & [\quad t : f. \quad], \\
13: & [\quad f : F(.e) \quad e : .eLt \mid .t \quad t : .tNf \mid .f \quad f : .F(e) \mid .V \mid .C \quad], \\
14: & [\quad e : eL.t \quad t : .tNf \mid .f \quad f : .F(e) \mid .V \mid .C \quad], \\
15: & [\quad a : V = e;. \quad], \\
16: & [\quad t : tN.f \quad f : .F(e) \mid .V \mid .C \quad], \\
17: & [\quad f : F(e.) \quad e : e.Lt \quad], \\
18: & [\quad e : eLt. \quad t : t.Nf \quad], \\
19: & [\quad t : tNf. \quad], \\
20: & [\quad f : F(e). \quad]
\end{array}
$$

}

and transitions

$$
\begin{aligned}
E_{LR(0)} = \{ & \\
& [0,1,V],[0,2,s],[0,3,a],[1,4,=],[2,5,\$end],[3,1,V],[3,3,a],[3,6,s], \\
& [4,7,V],[4,8,C],[4,9,F],[4,10,e],[4,11,t],[4,12,f],[9,13,(],[10,14,L], \\
& [10,15,;],[11,16,N],[13,7,V],[13,8,C],[13,9,F],[13,11,t],[13,12,f], \\
& [13,17,e],[14,7,V],[14,8,C],[14,9,F],[14,12,f],[14,18,t],[16,7,V], \\
& [16,8,C],[16,9,F],[16,19,f],[17,14,L],[17,20,)],[18,16,N] \\
& \}
\end{aligned}
$$

The presence of shift-reduce conflicts indicates that SL^2 cannot be parsed without taking lookahead into account. For example, when reaching state 18, it is unclear whether to reduce using production rule P4 or whether to shift. Our SLR parser uses the *Follow* sets of the nonterminal symbols (the set of terminal symbols that can follow the nonterminal symbol in some derivation) to make the decision about the next action deterministic. Shift-reduce conflicts are resolved by shifting whenever there is an outgoing edge labeled with the next input symbol and whose target is not the error state. A shift is performed in state 18 if the lookahead is N. Otherwise, the handle eLt is reduced to e using production rule P4.

SLR parsing of the input string "$V = F(VNC);$" is illustrated in Table 4.2. Initially, the stack is empty in state 0 and the first token is read (shifted). Acceptance of the given string is obtained after a total of 32 steps. For example, line 5 in Table 4.2 shows the parser in state 13 after reading the first four tokens "$V = F($". State 13 is a shift state. The next token (V) is read from the input yielding the transition into state 7 while 13 is pushed onto the stack. State 7 is a reduce state. Production rule $P9$ is used to replace V by f followed by a backward transition into state 13. With the length of V being equal to one, only the top element 13 is popped from the stack. The following transition on f is from state 13 to state 12. Similar arguments yield the remaining entries in Table 4.2. The shift-reduce conflict

Table 4.2. *SLR Parsing of "$V = F(VNC);$" based on Definition* 4.15*; we show the contents of the* STACK*, the current* STATE *in the characteristic automaton, the string* PARSED *so far, the remaining* INPUT*, and the* ACTION *to be taken in each state. The parse tree can be derived by bottom-up interpretation of the reductions in the last column.*

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
5	0,1,4,9	13	$V = F(V$	$NC);$	S
6	0,1,4,9,13	7		$NC);$	R($P9$)
7	0,1,4,9	13	$V = F(f$	$NC);$	S
8	0,1,4,9,13	12		$NC);$	R($P7$)
9	0,1,4,9	13	$V = F(t$	$NC);$	S
10	0,1,4,9,13	11	$V = F(tN$	$C);$	S
11	0,1,4,9,13,11	16	$V = F(tNC$	$);$	S
12	0,1,4,9,13,11,16	8		$);$	R($P10$)
13	0,1,4,9,13,11	16	$V = F(tNf$	$);$	S
14	0,1,4,9,13,11,16	19		$);$	R($P6$)
15	0,1,4,9	13	$V = F(t$	$);$	S
16	0,1,4,9,13	11		$);$	R($P5$)
17	0,1,4,9	13	$V = F(e$	$);$	S
18	0,1,4,9,13	17	$V = F(e)$	$;$	S
19	0,1,4,9,13,17	20		$;$	R($P8$)
20	0,1	4	$V = f$	$;$	S
21	0,1,4	12		$;$	R($P7$)
22	0,1	4	$V = t$	$;$	S
23	0,1,4	11		$;$	R($P5$)
24	0,1	4	$V = e$	$;$	S
25	0,1,4	10	$V = e;$		S
26	0,1,4,10	15			R($P3$)
27		0	a		S
28	0	3			R($P1$)
29		0	s		S
30	0	2	$s\$end$		S
31	0,2	5			R($P0$)
32		0	$\$accept$		ACCEPT

in state 11 is resolved in the favor of shifting whenever the next token is N. Consequently, a shift is performed in line 10 of Table 4.2, whereas reductions take place in lines 16 and 23.

4.4.4 A Simple Operator Precedence Language

The parser generator `bison` permits the explicit specification of operator precedence and of the order of resolution of associativity. The production rules of the resulting grammars may turn out to be more intuitive. A corresponding reformulation of SL is based on the SL^2 grammar in Definition 4.12 as follows.

Definition 4.16. *An* SL program *is a sequence of statements described by an extension of the SL^2 grammar $G = (V_n, V_t, P, s)$ in Definition* 4.12*. We add nonterminal symbols*

$$V_n := V_n \cup \left\{ \begin{array}{ll} b & \textit{(branch statement)} \\ l & \textit{(loop statement)} \\ r & \textit{(result of relational operator)} \end{array} \right\},$$

terminal symbols

$$V_t := V_t \cup \left\{ \begin{array}{rl} IF & \textit{(branch keyword)} \\ WHILE & \textit{(loop keyword)} \\ R & \textit{(binary relational operator)} \\ \}\{ & \textit{(further single character tokens)} \end{array} \right\},$$

and production rules

$$P := P \cup \left\{ \begin{array}{llll} (P1a) & s:b & (P1b) & s:l \\ (P2a) & s:bs & (P2b) & s:ls \\ (P9) & b:IF(r)\{s\} & (P10) & l:WHILE(r)\{s\} \\ (P11) & r:VRV & & \end{array} \right\}.$$

The start symbols remains unchanged.

We use SL^2 programs defined by the production rules $P1$, $P2$, and $P3$–$P8$ in Definitions 4.12 and 4.16 for illustration of the parsing algorithm. The configurating sets define the vertices of the parser's characteristic automaton as shown in Figure 4.14. Ambiguity is resolved by specifying associativity and precedence of binary operators as well as by considering Follow sets as done by SLR parsers. Table 4.3 illustrates the use of the characteristic automaton for parsing "$V = F(VNC)$;".

4.4.5 Parser for SL^2 Programs with `flex` and `bison`

We use `flex` and `bison` to implement a parser for SL^2 programs. For the sake of brevity, variables and constants are restricted to lowercase letters and single digits, respectively. The `flex` source file is shown in Listing 4.2. Whitespaces are ignored (line 5). Intrinsic functions (line 12), linear operators (line 13), and nonlinear operators (line 14) are represented by a single instance each. The named tokens (F, L, N, V, and C) to be returned to the parser are encoded as integers in the file parser.tab.h. Its inclusion into lex.yy.c prior to any other automatically generated code is triggered by the corresponding preprocessor directive at the beginning of the first section of the `flex` input file (lines 1–3). The remaining unnamed single character tokens are simply forwarded to the parser (line 17). A special lexinit routine is provided to set the pointer yyin to the given source file (line 21).

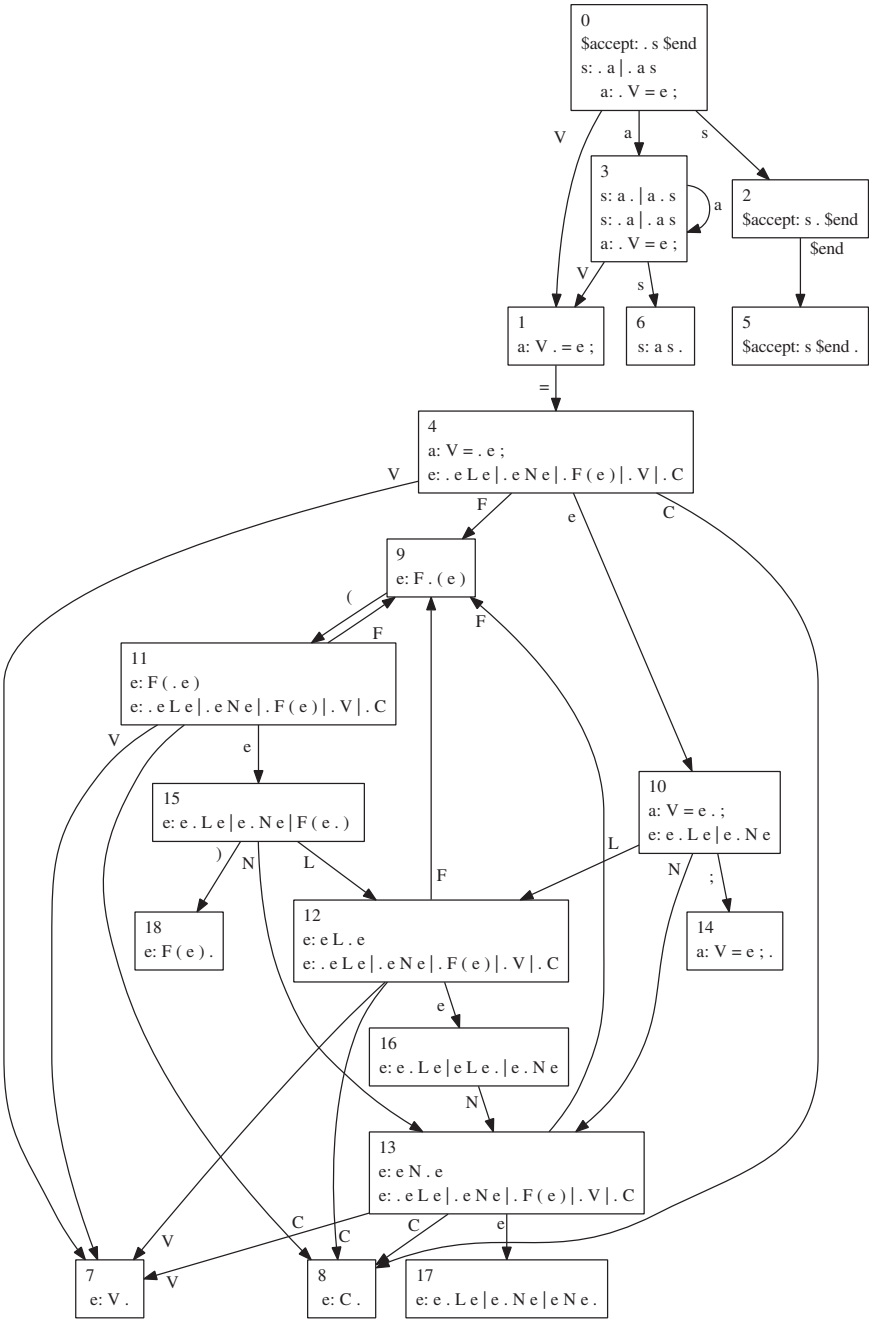


Figure 4.14. *Characteristic automaton for SL^2 programs.*

Table 4.3. *Parsing "$V = F(VNC);$" based on Definition* 4.16*. We show the contents of the* STACK*, the current* STATE *in the characteristic automaton, the string* PARSED *so far, the remaining* INPUT*, and the* ACTION *to be taken in each state. The parse tree can be derived by bottom-up interpretation of the reductions in the last column.*

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
5	0,1,4,9	11	$V = F(V$	$NC);$	S
6	0,1,4,9,11	7		$NC);$	R($P6$)
7	0,1,4,9	11	$V = F(e$	$NC);$	S
8	0,1,4,9,11	15	$V = F(eN$	$C);$	S
9	0,1,4,9,11,15	13	$V = F(eNC$	$);$	S
10	0,1,4,9,11,15,13	8		$);$	R($P7$)
11	0,1,4,9,11,15	13	$V = F(eNe$	$);$	S
12	0,1,4,9,11,15,13	17		$);$	R($P4$)
13	0,1,4,9	11	$V = F(e$	$);$	S
14	0,1,4,9,11	15	$V = F(e)$	$;$	S
15	0,1,4,9,11,15	18		$;$	R($P5$)
16	0,1	4	$V = e$	$;$	S
17	0,1,4	10	$V = e;$		S
18	0,1,4,10	14			R($P3$)
19		0	a		S
20	0	3			R($P1$)
21		0	s		S
22	0	2	$\$end$		S
23	0,2	5			R($P0$)
24		0	$\$accept$		ACCEPT

Listing 4.2. `flex` *input file.*

```
1 %{
2 #include "parser.tab.h"
3 %}
4
5 whitespace     [ \t\n]+
6 variable       [a-z]
7 constant       [0-9]
8
9 %%
10
11 {whitespace}   { }
12 "sin"          { return F; }
13 "+"            { return L; }
14 "*"            { return N; }
15 {variable}     { return V; }
16 {constant}     { return C; }
17 .              { return yytext[0]; }
```

```
18
19 %%
20
21 void lexinit(FILE *source) { yyin=source; }
```

Similar to the `flex` input file, the `bison` input file consists of three sections separated by %% that contain definitions (e.g., of tokens and rules for resolving associativity and operator precedence; see Listing 4.3), production rules (Listing 4.4), and user-defined routines (Listing 4.5), respectively. The five named tokens are defined in line 1 of Listing 4.3. Lines 3 and 4 set nonlinear operators to precede linear ones. Associativity is resolved by generating locally left-most parse trees.

Listing 4.3. *First section of the* `bison` *input file.*

```
1 %token V C F L N
2
3 %left L
4 %left N
5
6 %%
7  ...
```

Unnamed single character tokens are enclosed within single quotes inside the production rules.

Listing 4.4. *Second section of the* `bison` *input file.*

```
...

s : a
  | a s
  ;
a : V '=' e ';' ;
e : e L e
  | e N e
  | F '(' e ')'
  | V
  | C
  ;

%%
...
```

Two user-defined routines are provided. The basic error handler in line 5 of Listing 4.5 simply prints the error message that is generated by the parser to standard output. Inside the main routine, the source file is opened for read-only access (line 9) and the corresponding FILE pointer is passed on to the scanner (line 10). The parser itself is started by calling yyparse() in line 11. It calls the scanner routine yylex() to get the next token as required. Finally, the source file is closed (line 12).

Listing 4.5. *Third section of the* `bison` *input file.*

```
1 ...
2
3 #include<stdio.h>
4
5 int yyerror(char *msg) { printf("%s \n",msg); return -1; }
6
7 int main(int argc,char** argv)
8 {
9   FILE *source_file=fopen(argv[1],"r");
10  lexinit(source_file);
11  yyparse();
12  fclose(source_file);
13  return 0;
14 }
```

The dependencies within the build process are best illustrated with the following makefile [46]:

```
parse : lex.yy.c parser.tab.c
        gcc parser.tab.c lex.yy.c -lfl -o parse

parser.tab.c : parser.y
        bison -d parser.y

lex.yy.c : scanner.l
        flex scanner.l
```

The executable parse is built from the two C-files with default names lex.yy.c and parser.tab.c that are generated by `flex` and `bison`, respectively. Running `bison` with the `-d` option yields the generation of parser.tab.h that contains all token definitions to be included into lex.yy.c as described. The object code is linked with the `flex` run time support library (`-lfl`).

Similar to `flex`, the parser generator `bison` can generate diagnostic information. When run as `bison -v parser.y`, the diagnostic output is written into a file named `parser.output`. Its contents starts with a summary of the underlying augmented grammar followed by information on terminal and nonterminal symbols and rules where they appear. Production rules are enumerated as follows:

```
0 $accept: s $end

1 s: a
2  | a s

3 a: V '=' e ';'

4 e: e L e
5  | e N e
6  | F '(' e ')'
7  | V
8  | C
```

Most importantly, `bison` reports on the characteristic finite automaton that the generated parser is based on. The following output is generated for the specifications in Listing 4.4.

```
state 0

    0 $accept: . s $end

    V  shift, and go to state 1

    s  go to state 2
    a  go to state 3

...

state 3

    1 s: a .
    2  | a . s

    V  shift, and go to state 1

    $default  reduce using rule 1 (s)

    s  go to state 6
    a  go to state 3

...

state 18

    6 e: F '(' e ')' .

    $default  reduce using rule 6 (e)
```

The output has been edited for brevity as indicated by the three consecutive dots. All states list the *kernels* of their respective configurating sets while omitting the remaining production rules of their closures. Transitions that correspond to shift operations are listed as well as potential reductions and their effects. For example, in state 3 shifting requires the next token to be read to be V. The characteristic automaton moves into state 1 in this case. If the preceding reduction is to s or a, then the characteristic automaton moves into states 6 or it remains in state 3, respectively. Otherwise it attempts to reduce to s using production rule 1. A syntax error is reported if none of the previously mentioned situations occurs. `bison` can also generate a graphical representation of the characteristic automaton. Refer to `bison`'s online documentation for further up-to-date information on its diagnostic capabilities.

4.4.6 Interaction between `flex` and `bison`

The following two case studies are used to discuss the interaction between `flex` and `bison` in more detail.

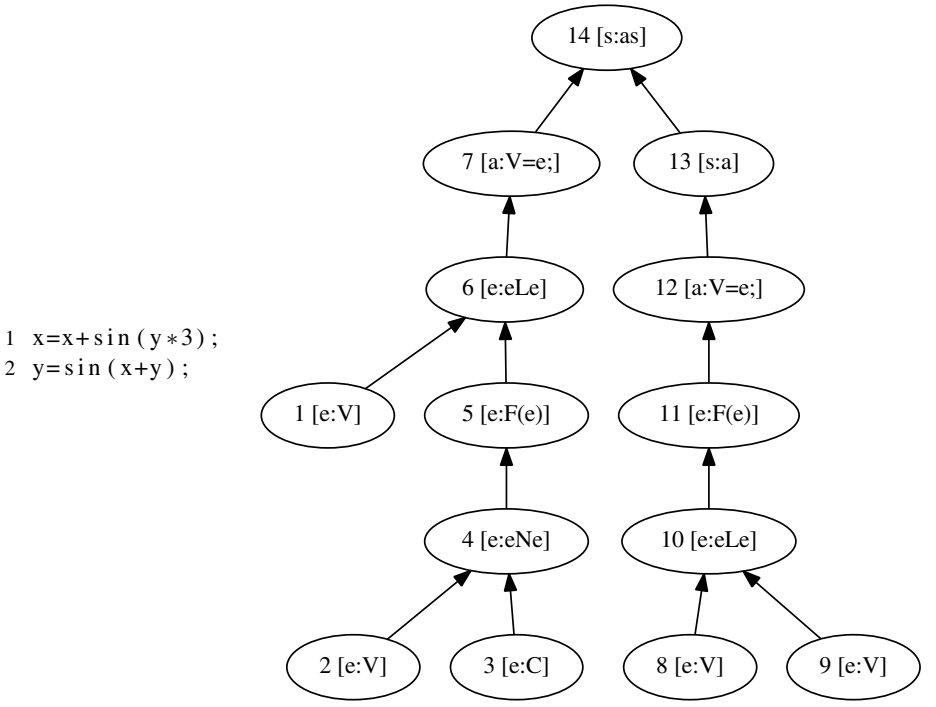


Figure 4.15. *Parse tree of SL^2 program.*

Parse Tree Printer We aim to print a structural representation of the parse tree that corresponds to derivations performed for syntactically correct input strings based on the SL^2 grammar. An example is shown in Figure 4.15. Vertices are enumerated and their associated production rules are listed. The graphical output is generated by the `graphviz` utility `dot` [27].

Listing 4.6. *First section of the* `bison` *input file.*

```
1 %{
2 int i; // vertex counter
3 #include<stdio.h>
4 %}
5
6 %token V C F L N
7
8 %left L
9 %left N
```

The `flex` input file is similar to Listing 4.2. We consider the three sections of the `bison` input file separately. Its first part contains a new section in addition to the definition of tokens (line 6 in Listing 4.6) and of rules for resolving ambiguity / shift-reduce conflicts due to operator precedence / associativity of the binary arithmetic operators (lines 8 and 9). Code enclosed within %{ and %} is copied by `bison` into the generated C-file without

modification. A global integer variable i is declared as a vertex counter in line 2. The interface to the I/O routines within the C standard library needs to be included in order to get access to printf as well as to the FILE data type (line 3).

The parse tree is built bottom-up with leafs representing expressions reduced from V or C tokens. All grammar symbols (both terminal and nonterminal) are associated with data. By default, this data is a single integer, which is exploited in the current example. Access to the symbol data is by position in the given production rule. For example, in e:eLe, the three integers that correspond to the symbols on the right-hand side are referenced as $1, $2, and $3. The data of the nonterminal symbol on the left-hand side is accessed as $$.

Listing 4.7 shows excerpts from the second part of the `bison` input file. Production rules are augmented with actions to ensure the correct enumeration of the parse tree vertices and their connection by edges in the generated `dot` output.

Listing 4.7. *Second section of the* `bison` *input file.*

```
1 s : a
2   {
3     $$=++i;
4     printf("%d [label=\"%d [s:a]\"]\n",$$,$$);
5     printf("%d->%d\n",$1,$$);
6   }
7 ...
8 a : V '=' e ';'
9   {
10    $$=++i;
11    printf("%d [label=\"%d [a:V=e;]\"]\n",$$,$$);
12    printf("%d->%d\n",$3,$$);
13  }
14  ;
15 e : e L e
16  {
17    $$=++i;
18    printf("%d [label=\"%d [e:eLe]\"]\n",$$,$$);
19    printf("%d->%d\n",$1,$$);
20    printf("%d->%d\n",$3,$$);
21  }
22 ...
23  | V
24  {
25    $$=++i;
26    printf("%d [label=\"%d [e:V]\"]\n",$$,$$);
27  }
28 ...
```

Each reduction causes the incrementation of the global vertex counter (lines 3, 10, 17, and 25). The new vertex that represents the nonterminal symbol on the left-hand side of the respective production rule is labeled with the corresponding index (lines 4, 11, 18, and 26). Indices of predecessors are accessed through their position in the right-hand side. Edges are added to the `dot` output accordingly (lines 5, 12, 19, and 20).

The third part of the `bison` input file is shown in Listing 4.8. It contains a basic version of the error handling routine yyerror and the main routine. The latter opens the file to

be parsed, provides this information to the scanner, initializes the parse tree vertex counter, and calls the parsing routine. Suitable `dot` output of the parse tree as a directed graph drawn from Bottom (leafs) to Top toward the root (sink) s is generated. The orientation is set via the rankdir attribute. Corresponding wrapper code that is written in lines 8 and 10 encloses the code generated in Listing 4.7.

Listing 4.8. `bison` *input file – Part* 3.

```
1 int yyerror(char *msg) { printf("%s \n",msg); return -1; }
2
3 int main(int argc,char** argv)
4 {
5   FILE *source_file=fopen(argv[1],"r");
6   lexinit(source_file);
7   i=0;
8   printf("digraph {\n rankdir=BT;\n");
9   yyparse();
10  printf("}\n");
11  fclose(source_file);
12  return 0;
13 }
```

Syntax-Directed Unparser We consider the single-pass generation of a verified syntactically equivalent copy of the input code as an important next step toward single-pass derivative code generation. Relevant modifications to the `flex` input file are documented by Listing 4.9 and Listing 4.10.

Listing 4.9. *First section of the* `flex` *input file.*

```
1  %{
2  #define YYSTYPE char * // needs to be defined prior to inclusion
3                         // of parser.tab.h
4  #include "parser.tab.h"
5
6  #define BUFFER_SIZE 3
7
8  #include<stdlib.h> // malloc
9  #include<string.h> // strcpy
10 void to_parser() {
11   yylval=(char*)malloc(BUFFER_SIZE*sizeof(char));
12   strcpy(yylval,yytext);
13 }
14 %}
15
16 whitespace      [ \t\n]+
17 variable        [a-z]
18 constant        [0-9]
```

Specific names of all tokens need to be passed from the scanner to the parser in order to be copied correctly to the output. Therefore, the default type of the information that is associated with all parse tree nodes needs to be changed to the C-string type **char***. The preprocessor macro YYSTYPE is redefined accordingly in line 2 prior to the inclusion of parser.tab.h that contains references to YYSTYPE. A buffer of characters of sufficient

size BUFFER_SIZE and with built-in name yylval is allocated in line 11, and it is used by the function to_parser to pass the string associated with the current token to the parser. Appropriate declarations from the C standard library need to be included (lines 8 and 9). Simplified lexical definitions of whitespaces, variables, and constants follow in lines 16–18.

The various tokens are handled in the second part of the `flex` input file. For simplicity, whitespaces are ignored in line 1 of Listing 4.10. While passing whitespaces on to the parser results in an exact copy of the input code the size of the buffer (here, set equal to 3 as none of the tokens is represented by a string of length greater than 3) becomes unpredictable. Formatting of the output is taken care of by the parser.

Listing 4.10. *Second section of the* `flex` *input file.*

```
1 {whitespace}     { }
2 "sin"            { to_parser(); return F; }
3 "cos"            { to_parser(); return F; }
4 "*"              { to_parser(); return N; }
5 "/"              { to_parser(); return N; }
6 "+"              { to_parser(); return L; }
7 "-"              { to_parser(); return L; }
8 {variable}       { to_parser(); return V; }
9 {constant}       { to_parser(); return C; }
10 .               { return yytext[0]; }
```

Several instances of the same token type are distinguished through their actual names. For example, both sin and cos are tokens of type F. Single character tokens that are not explicitly listed are simply passed on to the parser in line 10. The third section of the `flex` input file is not listed as it contains nothing but the standard **void** lexinit (FILE*) routine.

Section one of the `bison` input file is similar to Listing 4.6 except for the missing declaration of the parser tree vertex counter that is not required by the syntax-directed unparser. Its listing is omitted. The second section of the `bison` input file augments the production rules with appropriate actions for printing a syntactically equivalent copy of the input code.

Listing 4.11. *Second section of the* `bison` *input file.*

```
1 s : a
2   | a s
3   ;
4 a : V '=' { printf("%s=",$1); } e ';' { printf(";\n"); }
5   ;
6 e : e L { printf("%s",$2); } e
7   | e N { printf("%s",$2); } e
8   | F '(' { printf("%s(",$1); } e ')' { printf(")"); }
9   | V { printf("%s",$1); }
10  | C { printf("%s",$1); }
11  ;
```

In line 4 of Listing 4.11, reading the left-hand side of an assignment and the assignment operator is succeeded by the output of the string that is associated with the token V followed by '='. Bottom-up parsing of the expression e on the right-hand side yields corresponding output due to the actions associated with the production rules in lines 6–10. All assignments

are finished with a semicolon. Output of the following newline character ensures that each assignment is printed onto a new line.

The third part of the `bison` input file is similar to that in Listing 4.4.

4.5 Single-Pass Derivative Code Compilers

This section deals with the simplest possible method for generating derivative code. SL has been designed to facilitate this approach. It shows nicely the link between differentiation and compiler construction. Many statements and algorithms in this section are mostly conceptual. State-of-the-art derivative code compilers implement variants thereof in order to achieve the wanted efficiency. Our aim is to communicate fundamental concepts as the basis for a better understanding of advanced concepts and source code transformation algorithms that are described in the literature.

4.5.1 Attribute Grammars

Attribute grammars are very powerful tools for rigorous definition of source code analysis and modification algorithms. Transformation rules are associated with the syntactical structure of the language making their implementation often a straightforward extension of the parser. A single pass over the input program may suffice to perform the desired modifications of the source code. Moreover, attribute grammars describe exactly the corresponding traversal and modification pattern of the parse tree, thus making them the preferred approach to the specification of source transformation rules in multipass compilation as well.

Definition 4.17. *A* synthesized attribute *$v.s$ of the left-hand side v of a production v : $\phi(u_1,\ldots,u_k)$ is defined only in terms of its own attribute values or of attribute values of its children $u_1,\ldots,u_k$.*

An inherited attribute *$v.i$ of a symbol v on the right-hand side of a production w : $\phi(u_1,\ldots,v,\ldots,u_k)$ is defined only in terms of its own attribute values, of those of its parent w, or of attribute values of its siblings $u_1,\ldots,u_k$.*

There is no need to let inherited attributes be dependent on attributes of the children of the associated symbol as the underlying grammar can be rewritten to separate synthesized and inherited attributes. Refer to [2] for details.

Definition 4.18. *A grammar is called* S-attributed *if it contains only synthesized attributes.*

In an L-attributed *grammar the values of all inherited attributes of an instance v of a nonterminal symbol are either functions of synthesized or inherited attributes of nonterminals to the left in the given production (including the parent on the left-hand side); or they are functions of synthesized or inherited attributes of v itself. Cyclic dependencies among the attributes of v must not occur.*

Any S-attributed grammar is also L-attributed.

The enumeration of subexpressions is a key ingredient of the generation of derivative code as it allows us to decompose complex expressions into elemental functions whose local directional derivatives and adjoints can be computed in a straightforward fashion.

Example 4.20 illustrates the syntax-directed enumeration of subexpressions on the right-hand side of assignments. It is based on Example 4.19 which shows how to augment the SL^2 grammar with rules for counting subexpressions in parse trees of right-hand sides of assignments.

Example 4.19 (S-attributed Counting of Subexpressions) Without loss of generality, we consider all SL^2 programs that consist of a single assignment only. Production rules P3-P8 in Definition 4.16 (see also Definition 4.12) are augmented with actions on the synthesized attribute s that holds the number of subexpressions in the parse tree rooted by the current vertex. Any reduction to e (rules P4-P8) adds a new subexpression. Unary intrinsics increment the number of subexpressions in their single argument. Binary operators add one to the sum of the numbers of subexpressions in the two operands. We use superscripts to distinguish between instances of the same nonterminal symbol within the same production rule. The left-hand side of a production is potentially augmented with superscript l. Counters r_1 and r_2 denote the first and second occurrences of a symbol on the right-hand side of the production, respectively.

$$
\begin{array}{ll}
a : V = e; & \\
e^l : F(e^r) & \{e^l.s := e^r.s + 1\} \\
\quad : e^{r_1} L e^{r_2} & \{e^l.s := e^{r_1}.s + e^{r_2}.s + 1\} \\
\quad : e^{r_1} N e^{r_2} & \{e^l.s := e^{r_1}.s + e^{r_2}.s + 1\} \\
\quad : V & \{e^l.s := 1\} \\
\quad : C & \{e^l.s := 1\}
\end{array}
$$

The implementation with `flex` and `bison` is straightforward. The scanner is the same as in Section 4.4.5. Modifications are restricted to the first and second parts of the `bison` input file. The latter becomes

```
a : V '=' e ';' { printf("%d\n",$3); } ;
e : e L e { $$=$1+$3+1; }
  | e N e { $$=$1+$3+1; }
  | F '(' e ')' { $$=$3+1; }
  | V { $$=1; }
  | C { $$=1; }
  ;
```

requiring the addition of

```
%{ #include<stdio.h> %}
```

to the first part in order to gain access to the definition of printf. Application to "y=sin(x*2);" yields the output 4 which corresponds to the four subexpressions "x", "2", "x*2", and "sin(x*2)". ∎

Example 4.20 (L-attributed Enumeration of Subexpressions) We consider the same grammar as in the previous example. An inherited attribute i that represents the unique index of each subexpression is added to the S-attributed grammar developed in Example 4.19.

This index will be used to generate single assignment code in Section 4.5.2. The bottom-up evaluation of the synthesized attribute values is followed by a top-down sweep on the parse tree to propagate i.

$$
\begin{array}{ll}
a : V = e; & \{\ e.i := 0\ \} \\
e^l : F(e^r) & \{\ e^l.s := e^r.s + 1;\ e^r.i := e^l.i + 1\ \} \\
\quad : e^{r_1} L e^{r_2} & \{\ e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \\
 & \quad e^{r_1}.i := e^l.i + 1;\ e^{r_2}.i := e^{r_1}.i + e^{r_1}.s\ \} \\
\quad : e^{r_1} N e^{r_2} & \{\ e^l.s := e^{r_1}.s + e^{r_2}.s + 1 \\
 & \quad e^{r_1}.i := e^l.i + 1;\ e^{r_2}.i := e^{r_1}.i + e^{r_1}.s\ \} \\
\quad : V & \{\ e^l.s := 1\ \} \\
\quad : C & \{\ e^l.s := 1\ \}
\end{array}
$$

Right-hand sides of assignments receive index $i = 0$. Arguments of unary operations, as well as the first operands in binary operations, receive the index of their parent incremented by one. For the second operands in binary operations, the index of the parent needs to be increased by the number of subexpressions in the first operand.

The order of evaluation of inherited attributes (from parents to children) contradicts the direction in which bottom-up parsers build the parse tree (from children to parents). Arbitrary parse tree traversals can be implemented if a physical copy of the tree is stored. However, the single-pass evaluation of inherited attributes does not fit well into the logic of shift-reduce parser generators such as `bison`. Nevertheless, certain algorithmic tricks can enable the implementation of the desired effects.

For example, the enumeration of subexpressions in right-hand sides of assignments can be implemented using a global SAC variable counter sacvc as shown in the following partial listing of a corresponding `bison` input file:

```
1 %{ unsigned int sacvc; %}
2
3 %token V C F L N
4
5 %left L
6 %left N
7
8 %%
9
10 a : V '=' { sacvc=0; } e ';'
11 e : e L e { $$=sacvc++; }
12   | e N e { $$=sacvc++; }
13   | F '(' e ')' { $$=sacvc++; }
14   | V { $$=sacvc++; }
15   | C { $$=sacvc++; }
16   ;
17
18 %%
19 ...
```

Reduction of a new subexpression in lines 11–15 yields the incrementation of the initially zero (see line 10) global counter, thus ensuring uniqueness of the assigned index. The order of enumeration is bottom-up instead of top-down. It is irrelevant in the context of assignment-level SAC whose syntax-directed generation is discussed in the following section. ∎

4.5.2 Syntax-Directed Assignment-Level SAC

Conceptually, derivative code is based on decompositions of all assignments into SAC as in (1.4). We present an L-attributed grammar for SAC of single assignments. The enumeration of the SAC variables is top-down on the parse tree starting with v_0 as in Example 4.20.

Three attributes are associated with each nonterminal symbol v. The synthesized attribute $v.s$ contains the number of subexpressions in the subtree with root v as in Example 4.19. The inherited attribute $v.i$ contains the corresponding unique SAC variable index as in Example 4.20. The SAC associated with the subtree with root v is synthesized in $v.c$. The scanner is expected to store the sequence of characters that correspond to a token T in $T.c$—that is, $V.c$ contains the variable name, $C.c$ the value (as a sequence of characters) of a constant, $O.c$ the operator (e.g. $+$, $-$, $*$, or $/$), and $F.c$ the name of the associated elemental function (e.g. sin, cos, log, etc.).

Again, we use superscripts to distinguish between occurrences of the same symbol v in a production. When synthesizing the SAC in attribute c, we use an overloaded $+$ operator[11] to concatenate sequences of strings and integers. For example, the result of ''v'' $+ e^l.i$ is the string ''v5'' if $e^l.i = 5$.

$$
\begin{array}{lll}
(P3) & a : V = e; & e.i = 0 \\
 & & a.c = e.c \\
 & & \quad + V.c + \text{“=v0;”} \\
(P4) & e^l : e^{r_1} L e^{r_2} & e^l.s = e^{r_1}.s + e^{r_2}.s + 1 \\
 & & e^{r_1}.i = e^l.i + 1 \\
 & & e^{r_2}.i = e^{r_1}.i + e^{r_1}.s \\
 & & e^l.c = e^{r_1}.c + e^{r_2}.c \\
 & & \quad + \text{“v”} + e^l.i + \text{“=v”} + e^{r_1}.i + L.c + \text{“v”} + e^{r_2}.i + \text{“;”} \\
(P5) & e^l : e^{r_1} N e^{r_2} & e^l.s = e^{r_1}.s + e^{r_2}.s + 1 \\
 & & e^{r_1}.i = e^l.i + 1 \\
 & & e^{r_2}.i = e^{r_1}.i + e^{r_1}.s \\
 & & e^l.c = e^{r_1}.c + e^{r_2}.c \\
 & & \quad + \text{“v”} + e^l.i + \text{“=v”} + e^{r_1}.i + N.c + \text{“v”} + e^{r_2}.i + \text{“;”}
\end{array}
$$

[11] The semantics of the operator $+$ is modified according to the principles of operator overloading in, for example, C++.

Shift-reduce conflicts are resolved by specifying the order of evaluation for associativity and operator precedence as discussed in Section 4.4.4.

$$
\begin{array}{lll}
(P6) & e^l : F(e^r) & e^l.s = e^r.s + 1 \\
 & & e^r.i = e^l.i + 1 \\
 & & e^l.c = e^r.c \\
 & & \quad + \text{“v”} + e^l.i + \text{“=”} + F.c + \text{“(v”} + e^r.i + \text{“);”} \\
(P7) & e : V & e.s = 1 \\
 & & e.c = \text{“v”} + e.i + \text{“=”} + V.c + \text{“;”} \\
(P8) & e : C & e.s = 1 \\
 & & e.c = \text{“v”} + e.i + \text{“=”} + C.c + \text{“;”}
\end{array}
$$

Table 4.4 illustrates the attribute grammar's use in a syntax-directed assignment-level SAC generator for SL^2 programs. The sequence of tokens in the assignment “y=sin(x∗2);” is parsed, and its SAC is synthesized in the c attributes of the nonterminals. Subexpressions are enumerated top-down on the parse tree as shown in Example 4.20. The SAC of subtrees with roots that represent nonterminals on the right-hand side of the production are followed by the contribution of the production itself. An annotated representation of the parse tree is shown in Figure 4.16. The local contributions to the value of the c attribute can be unparsed immediately. An explicit construction of the parse tree is not necessary as shown in the following proof-of-concept implementation.

Implementation

We use `flex` and `bison` to build assignment-level SACs for SL^2 programs. The corresponding `flex` and `bison` input files are shown in Listings 4.12 and 4.13, respectively. An extension to SL programs is a straightforward exercise.

The single-pass SAC generator is based on Examples 4.19 and 4.20. It uses the structured type ptNodeType that is defined in a file `ast.h` to associate the two attributes i and c with the nodes in the parse tree.

```
#define BUFFER_SIZE 100000

typedef struct {
  int i;
  char* c;
} ptNodeType;

#define YYSTYPE ptNodeType
```

The `bison` preprocessor macro YYSTYPE is set to ptNodeType for this purpose. A sufficiently large buffer of characters is required to store the SAC. For simplicity, we define its size statically in `ast.h`.

The `flex` input file includes `ast.h` prior to `parser.tab.h`. Otherwise, it is similar to Listings 4.9 and 4.10.

Table 4.4. *Syntax-directed assignment-level SAC for* $y = \sin(x * 2)$.

i	PARSED	ACTION	\$\$.i	\$\$.c	Comment
0	V	S			
	...				
11	$V = F(V$	S			
7		R($P7$)	2	$\boxed{v_2 = x;}$	
	...				
13	V=F(eNC	S			
8		R($P8$)	3	$\boxed{v_3 = 2;}$	
13	V=F(eNe	S			
14		R($P5$)		$v_2 = x;$	$< \ldots = e^{r_1}.c$
				$v_3 = 2;$	$< \ldots = e^{r_2}.c$
			1	$\boxed{v_1 = v_2 * v_3;}$	$N.c =$ "$*$"
					$e^{r_1}.i = 2, e^{r_2}.i = 3$
11	V=F(e	S			
15	V=F(e)	S			
18		R($P6$)		$v_2 = x;$	$<$
				$v_3 = 2;$	$<$
				$v_1 = v_2 * v_3;$	$< \ldots = e^{r}.c$
			0	$\boxed{v_0 = \sin(v_1);}$	$F.c =$ "sin", $e^{r}.i = 1$
4	V=e	S			
10	V=e;	S			
14		R($P3$)		$v_2 = x;$	$<$
				$v_3 = 2;$	$<$
				$v_1 = v_2 * v_3;$	$<$
				$v_0 = \sin(v_1);$	$< \ldots = e.c$
				$\boxed{y = v_0;}$	$V.c =$ "y", $e.i = 0$
	...				
0	\$accept	ACCEPT			

Listing 4.12. `flex` *input file.*

```
%{
#include "ast.h"
#include "parser.tab.h"

#include<stdlib.h> // malloc
#include<string.h> // strcpy

void to_parser() {
  yylval.c=(char*)malloc(BUFFER_SIZE*sizeof(char));
  strcpy(yylval.c,yytext);
}

% }

whitespace      [ \t\n]+
```

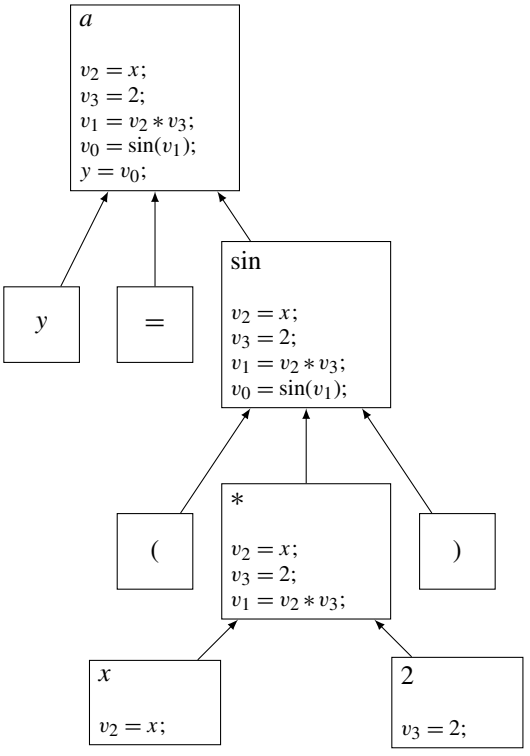


Figure 4.16. *SAC-augmented parse tree for* $y = \sin(x * 2)$.

```
variable        [a-z]
constant        [0-9]

%%

{whitespace}    { }
"sin"           { to_parser(); return F; }
"+"             { to_parser(); return L; }
"*"             { to_parser(); return N; }
{variable}      { to_parser(); return V; }
{constant}      { to_parser(); return C; }
.               { return yytext[0]; }

%%

void lexinit(FILE *source) { yyin=source; }
```

The `bison` input file does not yield many surprises. It uses routines get_buffer and free_buffer to manage the memory that is required during the synthesis of the SAC. Individual SAC variables are enumerated as in Example 4.20.

Listing 4.13. `bison` *input file.*

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "ast.h"

unsigned int sacvc;

void get_buffer(YYSTYPE* v) {
  v->c=malloc(BUFFER_SIZE*sizeof(char));
}
void free_buffer(YYSTYPE* v) {
  if (v->c) free(v->c);
}
%}

%token V C F L N

%left L
%left N

%%

sl2_program : s {
    printf("%s",$1.c);
    free_buffer(&$1);
  };
s : a
  | a s {
    get_buffer(&$$);
    sprintf($$.c,"%s%s",$1.c,$2.c);
    free_buffer(&$1); free_buffer(&$2);
  };
a : V '=' { sacvc=0; } e ';' {
    get_buffer(&$$);
    sprintf($$.c,"%s%s=v%d;\n",$4.c,$1.c,$4.j);
    free_buffer(&$1); free_buffer(&$4);
  };
e : e L e {
    $$.j=sacvc++;
    get_buffer(&$$);
    sprintf($$.c,"%s%sv%d=v%d%sv%d;\n",
                  $1.c,$3.c,$$.j,$1.j,$2.c,$3.j);
    free_buffer(&$1);
  }
  | e N e {
    // same as above
  }
```

```
  | F '(' e ')' {
    $$.j=sacvc++;
    get_buffer(&$$);
    sprintf($$.c,"%sv%d=sin(v%d);\n",
                  $3.c,$$.j,$3.j);
    free_buffer(&$3);
  }
  | V {
    $$.j=sacvc++;
    get_buffer(&$$);
    sprintf($$.c,"v%d=%s;\n",$$.j,$1.c);
    free_buffer(&$1);
  }
  | C {
    // same as above
  };

%%

int yyerror(char *msg) {printf("ERROR: %s \n",msg); return -1;}

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file); yyparse(); fclose(source_file);
  return 0;
}
```

Actions in the `bison` input file can be associated with reduce as well as with shift operations. For example, the rule

```
a : V "=" { sacvc=0; } e ";"
```

causes the variable sacvc to be initialized immediately after reading "=" from the input. The C library function sprintf is used to implement the overloaded + operator in the associated attribute grammar. The SAC statements are printed into the corresponding buffers. Local buffers are freed as soon as they are no longer required.

Example 4.21 Application of the syntax-directed assignment-level SAC compiler to the SL^2 program

```
x=x*y;
x=sin(x*y+3);
```

yields

```
v0=x; v1=y; v2=v0*v1; x=v2;
v0=x; v1=y; v2=v0*v1; v3=3; v4=v2+v3; v5=sin(v4); x=v5;
```

All variables are assumed to be scalar floating-point variables. Again, the numbering of the subexpressions on the right-hand side of assignments does not match the values of the corresponding inherited attribute in the associated attribute grammar. This difference is

irrelevant in the present context. All that is needed is uniqueness, which is guaranteed by the global counter mechanism. ∎

4.5.3 Syntax-Directed Tangent-Linear Code

Tangent-linear code is generated conceptually by attaching (directional) derivative components to each floating-point variable followed by differentiating all SAC assignments. According to (2.1), each SAC assignment $v_j = \varphi_j(v_i)_{i \prec j}$ is preceded by code for computing the inner product of the partial derivative of v_j with respect to all SAC variables v_i, $i \prec j$, on the right-hand side with the vector $(v_i^{(1)})_{i \prec j}$ of directional derivatives of these SAC variables. We use the underscore character to denote the directional derivative of a variable v, that is $\mathrm{v\_} \equiv v^{(1)}$.

The attributes are the same as in Section 4.5.2. The synthesized attribute c now contains the sequence of assignment-level SAC statements, each of them augmented with the corresponding elementary tangent-linear assignments. The resulting L-attributed grammar for tangent-linear versions of single assignments is the following:

$$
\begin{array}{lll}
(P3) & a : V = e; & e.i = 0 \\
 & & a.c = e.c \\
 & & \quad + V.c + \text{“\_=v0\_;”} \\
 & & \quad + V.c + \text{“=v0;”}
\end{array}
$$

For $y = v_0$, we get $y^{(1)} = \frac{\partial y}{\partial v_0} \cdot v_0^{(1)} = v_0$.

Linear $(P4)$ and nonlinear $(P5)$ operators can be described by a single rule $P4/5$. The differences are restricted to the expressions for the local partial derivatives.

$$
\begin{array}{lll}
(P4/5) & e^l : e^{r_1} O e^{r_2} & e^l.s = e^{r_1}.s + e^{r_2}.s + 1 \\
 & & e^{r_1}.i = e^l.i + 1 \\
 & & e^{r_2}.i = e^{r_1}.i + e^{r_1}.s \\
 & & e^l.c = e^{r_1}.c + e^{r_2}.c \\
 & & \quad + \text{“v”} + e^l.i + \text{“\_=”} + \partial_{e^{r_1}.i} O + \text{“*v”} + e^{r_2}.i + \text{“\_+”} \\
 & & \quad + \partial_{e^{r_2}.i} O + \text{“*v”} + e^{r_1}.i + \text{“\_;”} \\
 & & \quad + \text{“v”} + e^l.i + \text{“=v”} + e^{r_1}.i + O.c + \text{“v”} + e^{r_2}.i + \text{“;”}
\end{array}
$$

where $O \in \{L, N\}$ and the local partial derivatives are

$$\partial_{e^{r_1}.i} L := 1,$$

$$\partial_{e^{r_2}.i} L := \begin{cases} \text{“1”} & \text{if } L.c = \text{“+”}, \\ \text{“}-1\text{”} & \text{if } L.c = \text{“}-\text{”}, \end{cases}$$

$$\partial_{e^{r_1}.i} N := \begin{cases} \text{“v”} + e^{r_2}.i & \text{if } N.c = \text{“}*\text{”}, \\ \text{“1/v”} + e^{r_2}.i & \text{if } N.c = \text{“/”}, \end{cases}$$

and

$$\partial_{e^{r_2}.i} N := \begin{cases} \text{“v”} + e^{r_1}.i & \text{if } N.c = \text{“}*\text{”}, \\ \text{“}-\text{”} + \partial_{e^{r_1}.i} N + \text{“}*\text{”} + \partial_{e^{r_1}.i} N + \text{“}*\text{v”} + e^{r_1}.i & \text{if } N.c = \text{“/”}. \end{cases}$$

As before, shift-reduce conflicts are resolved by specifying the order of evaluation for associativity and operator precedence.

$$\begin{array}{lll} (P6) & e^l : F(e^r) & e^l.s = e^r.s + 1 \\ & & e^r.i = e^l.i + 1 \\ & & e^l.c = e^r.c \\ & & \quad + \text{“v”} + e^l.i + \text{“\_=”} + \partial_{e^r.i} F + \text{“}*\text{ v”} + e^r.i + \text{“\_;”} \\ & & \quad + \text{“v”} + e^l.i + \text{“=”} + F.c + \text{“(v”} + e^r.i + \text{“);”} \end{array}$$

where

$$\partial_{e^r.i} F := \begin{cases} \text{“cos(v”} + e^r.i + \text{“)”} & \text{if } F.c = \text{“sin”} \\ \text{“}-\text{sin(v”} + e^r.i + \text{“)”} & \text{if } F.c = \text{“cos”} \\ \text{“exp(v”} + e^r.i + \text{“)”} & \text{if } F.c = \text{“exp”} \\ \vdots \quad \text{etc.} \end{cases}$$

$$\begin{array}{lll} (P7) & e : V & e.s = 1 \\ & & e.c = \text{“v”} + e.i + \text{“\_=”} + V.c + \text{“\_;”} \\ & & \quad + \text{“v”} + e.i + \text{“=”} + V.c + \text{“;”} \\ (P8) & e : C & e.s = 1 \\ & & e.c = \text{“v”} + e.i + \text{“\_=0;”} \\ & & \quad + \text{“v”} + e.i + \text{“=”} + C.c + \text{“;”} \end{array}$$

Certain production rules of the SL grammar are omitted as the flow of control in the tangent-linear code is the same as in the original code. The actions associated with rules $P9$–$P11$ are simple unparsing steps. Rules $P1$–$P2$ yield a simple concatenation of tangent-linear code of sequences of statements.

The use of the attribute grammar in a syntax-directed tangent-linear code compiler is illustrated in Table 4.5 for the assignment “$y = \sin(x * 2)$;”. The derivation of the corresponding annotated parse tree is straightforward. A proof-of-concept implementation based on Listings 4.12 and 4.13 is left as an exercise.

Example 4.22 Application of the syntax-directed tangent-linear code compiler to the SL program

Table 4.5. *Syntax-directed tangent-linear code for* $y = \sin(x * 2)$; *set* $v_i^{(1)} \equiv$ v*i_ to establish the link with the code that is generated by the syntax-directed tangent-linear code compiler.*

i	PARSED	ACTION	\$\$.i	\$\$.c
0	V	S		
	...			
11	$V = F(V$	S		
7		R($P7$)	1	$\boxed{v_2^{(1)} = x^{(1)};\ v_2 = x;}$
	...			
13	V=F(eNC	S		
8		R($P8$)	2	$\boxed{v_3^{(1)} = 0;\ v_3 = 2;}$
13	V=F(eNe	S		
14		R($P5$)		$v_2^{(1)} = x^{(1)};\ v_2 = x;$ $v_3^{(1)} = 0;\ v_3 = 2;$
			3	$\boxed{v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ v_1 = v_2 * v_3;}$
11	V=F(e	S		
15	V=F(e)	S		
18		R($P6$)		$v_2^{(1)} = x^{(1)};\ v_2 = x;$ $v_3^{(1)} = 0;\ v_3 = 2;$ $v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ v_1 = v_2 * v_3;$
			4	$\boxed{v_0^{(1)} = \cos(v_1) * v_1^{(1)};\ v_0 = \sin(v_1);}$
4	V=e	S		
10	V=e;	S		
14		R($P3$)		$v_2^{(1)} = x^{(1)};\ v_2 = x;$ $v_3^{(1)} = 0;\ v_3 = 2;$ $v_1^{(1)} = v_2^{(1)} * v_3 + v_2 * v_3^{(1)};\ v_1 = v_2 * v_3;$ $v_0^{(1)} = \cos(v_1) * v_1^{(1)};\ v_0 = \sin(v_1);$ $\boxed{y^{(1)} = v_0^{(1)};\ y = v_0;}$
	...			
0	\$accept	ACCEPT		

```
if(x<y) {
  x=sin(x);
  while (y<x) { x=sin(x*3); }
  y=4*x+y;
}
```

yields

```
if (x<y) {
  v0_=x_; v0=x;
  v1_=cos(v0)*v0_; v1=sin(v0);
  x_=v1_; x=v1;
  while (y<x) {
```

```
    v0_=x_; v0=x;
    v1_=0; v1=3;
    v2_=v0_*v1+v0*v1_; v2=v0*v1;
    v3_=cos(v2)*v2_; v3=sin(v2);
    x_=v3_; x=v3;
  }
  v0_=0; v0=4;
  v1_=x_; v1=x;
  v2_=v0_*v1+v0*v1_; v2=v0*v1;
  v3_=y_; v3=y;
  v4_=v2_+v3_; v4=v2+v3;
  y_=v4_; y=v4;
}
```

All variables are assumed to be scalar floating-point variables. The flow of control remains unchanged. Each assignment is simply augmented with its tangent-linear code. As in Example 4.21, the numbering of the subexpressions is bottom-up instead of top-down due to the replacement of the inherited attribute i by a global counter.

In order to run this code, it must be wrapped into a function

```
void f_(double& x, double& x_, double& y, double& y_) {
  double v0,v1,v2,v3,v4;
  double v0_,v1_,v2_,v3_,v4_;

  // generated code goes here
}
```

that includes appropriate declarations of the SAC variables and of their tangent-linear versions. ∎

4.5.4 Syntax-Directed Adjoint Code

While the syntax-directed generation of assignment-level SACs as well as of tangent-linear code is straightforward, a similar approach to the generation of adjoint code is not obvious. A possible solution for SL programs is discussed next.

Five attributes are associated with each symbol. The semantics of the integer attributes i and s are similar to Section 4.5.3. A third integer attribute k serves as an enumerator of the assignments in the input code. It is used for the reversal of the flow of control. There are two text attributes to hold the forward (c^f) and backward (c^b) sections of the adjoint code. The text vector c^b has length α, where α denotes the number of assignment statements in the input code. The whole adjoint code is synthesized into $\$accept.c^f$ during a successful compilation. The complete augmented forward code $s.c^f$ is followed by the reverse loop over the adjoints of all assignments that are executed in the forward section.

The chosen approach to data (and hence control flow) reversal is meant to resemble (2.7) as closely as possible. We use control and required data stacks, accessed by push_c / pop_c and push_d / pop_d, respectively, to store all required information. This method is likely to fail for large-scale numerical simulations due to excessive memory requirement as previously discussed. Nevertheless, every potential expert derivative code compiler writer should realize that adjoint code is always semantically equivalent to this basic version.

$$
\begin{array}{lll}
(P0) & \$accept: & \\
 & & s.k = 0 \\
 & s\$end & \\
 & & \$accept.c^f = s.c^f \\
 & & \quad + \text{“}\textbf{int}\ i;\text{”} \\
 & & \quad + \text{“}\textbf{while}(pop\_c(i))\{\text{”} \\
 & & \quad + \quad \text{“}\textbf{if}(i == 1)\{\text{”} \\
 & & \quad + \quad\ \ s.c_1^b \\
 & & \quad + \quad \text{“}\}\textbf{else if}(i == 2)\{\text{”} \\
 & & \quad + \quad\ \ s.c_2^b \\
 & & \quad\ \ \vdots \\
 & & \quad + \quad \text{“}\}\textbf{else if}(i ==\text{”} + s.k + \text{“})\{\text{”} \\
 & & \quad + \quad\ \ s.c_{s.k}^b \\
 & & \quad + \quad \text{“}\}\text{”}
\end{array}
$$

$$
\begin{array}{lll}
(P1) & s: & \\
 & & a.k = s.k + 1 \\
 & a & \\
 & & s.k = a.k; \quad s.c^f = a.c^f; \quad s.c^b = a.c^b
\end{array}
$$

The vector assignment $s.c^\nu = a.c^\nu$ is defined as $s.c_i^\nu = a.c_i^\nu$ for $i = 1, \ldots, \alpha$ and $\nu \in \{f, b\}$. We present the production rules together with their associated actions similar to a corresponding implementation in `bison`. For example, the attribute k of a is set prior to parsing the assignment itself. The attribute k as well as the forward and backward code of s are synthesized at the time of the reduction to s. The value of the inherited assignment counter k is propagated top-down through sequences of statements that are described by rules $P1$, $P1a, \ldots, P2b$. It is incremented whenever a new assignment is parsed; see rules $P1$ and $P2$.

$$
\begin{array}{lll}
(P1a) & s: & \\
 & & b.k = s.k \\
 & b & \\
 & & s.k = b.k; \quad s.c^f = b.c^f; \quad s.c^b = b.c^b
\end{array}
$$

$$
\begin{array}{lll}
(P1b) & s: & \\
 & & l.k = s.k \\
 & l & \\
 & & s.k = l.k; \quad s.c^f = l.c^f; \quad s.c^b = l.c^b
\end{array}
$$

In production rules $P2$, $P2a$, and $P2b$ that describe sequences of statements with more than one element, the value of the assignment counter is passed from left to right prior to

processing the respective nonterminal symbol. Its value is returned to the left-hand side of the production rule at the time of reduction. Moreover, forward and backward sections of adjoint sequences of statements are built as concatenations of the respective code fragments that are associated with their children. The order is reversed in the synthesis of the reverse section.

$$
\begin{array}{lll}
(P2) & s^l: & \\
 & & a.k = s^l.k + 1 \\
 & a & \\
 & & s^r.k = a.k \\
 & s^r & \\
 & & s^l.k = s^r.k \\
 & & s^l.c^f = a.c^f + s^r.c^f; \quad s^l.c^b = s^r.c^b + a.c^b
\end{array}
$$

Again, the vector sum $s^\mu.c^\nu = s^\mu.c^\nu + a.c^\nu$ is elemental, that is, $s^\mu.c^\nu_i = s^\mu.c^\nu_i + a.c^\nu_i$ for $i = 1,\ldots,\alpha$, $\mu \in \{l,r\}$, and $\nu \in \{f,b\}$.

$$
\begin{array}{lll}
(P2a) & s^l: & \\
 & & b.k = s^l.k \\
 & b & \\
 & & s^r.k = b.k \\
 & s^r & \\
 & & s^l.k = s^r.k \\
 & & s^l.c^f = b.c^f + s^r.c^f; \quad s^l.c^b = s^r.c^b + b.c^b
\end{array}
$$

$$
\begin{array}{lll}
(P2b) & s^l: & \\
 & & l.k = s^l.k \\
 & l & \\
 & & s^r.k = l.k \\
 & s^r & \\
 & & s^l.k = s^r.k \\
 & & s^l.c^f = l.c^f + s^r.c^f; \quad s^l.c^b = s^r.c^b + l.c^b
\end{array}
$$

Assignment-level SAC is built as in Section 4.5.3. The root of the syntax tree of the expression of the right-hand side has fixed SAC variable index 0. Variable names are stored in $V.c^f$ by the scanner. The assignment of v0 to the variable on the left-hand side is preceded by push statements for storing the unique number $a.k$ of the assignment and the current value of its left-hand side $V.c^f$ on appropriately typed stacks. While, due to missing static data-flow analysis, it cannot be decided if this value is needed by the reverse section, this conservative approach ensures correctness of the adjoint code. The storage of the unique identifier of the assignment is necessary for the correct reversal of the flow of

control. Alternative approaches to control-flow reversal are discussed in the literature; see Chapter 2.

$(P3)$ a :

$$e.k = a.k; \quad e.i = 0$$

$$V = e;$$

$$\begin{aligned} a.c^f = {} & e.c^f + \text{"push\_c("} + a.k + \text{");"} \\ & + \text{"push\_d("} + V.c^f + \text{");"} \\ & + V.c^f + \text{"=v0;"} \end{aligned}$$

The adjoint assignment is built according to Adjoint Code Generation Rule 2; see Section 2.2.1. Incrementation of adjoint SAC variables, such as v0, is not necessary as values of SAC variables are read exactly once. The adjoint of the program variable on the left-hand side of the assignment is set equal to zero, followed by the execution of the adjoint code that corresponds to the SAC of the right-hand side of the assignment.

$$\begin{aligned} a.c^b_{a.k} = {} & \text{"pop\_d("} + V.c^f + \text{");"} \\ & + \text{"v0\_="} + V.c^f + \text{"\_;"} \\ & + V.c^f + \text{"\_=0;"} \\ & + e.c^b_{a.k} \end{aligned}$$

Again, linear and nonlinear operations are treated similarly with differences restricted to the local partial derivatives. Both the attributes for enumeration of subexpressions (i) and for identifying assignments uniquely (k) are propagated top-down. The left-hand sides of all SAC statements are stored on the data stack prior to being overwritten. Their values are recovered before the execution of the corresponding adjoint assignments in the reverse section. Note that this approach yields a larger memory requirement than the code that results from Adjoint Code Generation Rule 4. There, the storage of overwritten values is restricted to program variables, and assignment-level incomplete SAC is built within the reverse section to ensure access to arguments of the local partial derivatives. The corresponding modification of the attribute grammar is straightforward and hence left as an exercise.

$(P4/5)$ e^l :

$$e^{r_1}.i = e^l.i + 1; \quad e^{r_i}.k = e^l.k \ \text{ for } i = 1,2$$

$$e^{r_1}$$

$$e^{r_2}.i = e^{r_1}.i + e^{r_1}.s + 1$$

$$Oe^{r_2}$$

$$\begin{aligned} e^l.s &= e^{r_1}.s + e^{r_2}.s + 1 \\ e^l.c^f &= e^{r_1}.c^f + e^{r_2}.c^f \\ &\quad + \text{“push\_d(v”} + e^l.i + \text{“);”} \\ &\quad + \text{“v”} + e^l.i \\ &\quad + \text{“=v”} + e^{r_1}.i + O.c^f + \text{“v”} + e^{r_2}.i + \text{“;”} \end{aligned}$$

$$\begin{aligned} e^l.c^b_{e.k} = &\text{“pop\_d(v”} + e^l.i + \text{“);”} \\ &+ \text{“v”} + e^{r_2}.i + \text{“\_=”} + O_{e^{r_2}.i} + \text{“*v”} + e^l.i + \text{“\_;”} \\ &+ \text{“v”} + e^{r_1}.i + \text{“\_=”} + O_{e^{r_1}.i} + \text{“*v”} + e^l.i + \text{“\_;”} \\ &+ e^{r_2}.c^b_{e.k} + e^{r_1}.c^b_{e.k} \end{aligned}$$

where $O \in \{L, N\}$. As in Section 4.5.3 $O_{e^{r_1}.i}$ denotes the partial derivative of operation O with respect to the SAC variable that holds the value of the expression e^{r_1} (similarly e^{r_2}). Shift-reduce conflicts are resolved by specifying the order of evaluation for associativity and operator precedence.

A feasible treatment of unary intrinsics follows immediately from the previous discussion.

$$\begin{aligned} (P6) \qquad & e^l : \\ & e^r.i = e^l.i + 1; \quad e^r.k = e^l.k \\ F(e^r) \quad & \\ & e^l.s = e^r.s + 1 \\ & e^l.c^f = e^r.c^f \\ & \quad + \text{“push\_d(v”} + e^l.i + \text{“);”} \\ & \quad + \text{“v”} + e^l.i + \text{“=”} + F.c^f + \text{“(v”} + e^r.i + \text{“);”} \\ & e^l.c^b_{e.k} = \text{“pop\_d(v”} + e^l.i + \text{“);”} \\ & \quad + \text{“v”} + e^r.i + \text{“\_=”} + F_{e^r.i} + \text{“*v”} + e^l.i + \text{“\_;”} \\ & \quad + e^r.c^b_{e.k} \end{aligned}$$

F is an arbitrary unary function, such as sin or exp. $F_{e^r.i}$ denotes the partial derivative of F with respect to the SAC variable that holds the value of the expression e^r.

Assignments of values of program variables to SAC variables do not yield any surprises in the forward section.

$$\begin{aligned} (P7) \quad & e : V \quad e.s = 1 \\ & a.c^f = \text{“push\_d(v”} + e.i + \text{“);”} \\ & \quad + \text{“v”} + e.i + \text{“=”} + V.c^f + \text{“;”} \end{aligned}$$

In the reverse section, the adjoint program variable needs to be incremented according to Adjoint Code Generation Rule 2.

$$\begin{aligned} a.c^b_{e.k} = &\text{“pop\_d(v”} + e.i + \text{“);”} \\ &+ V.c^f + \text{“\_+=v”} + e.i + \text{“\_;”} \end{aligned}$$

The assignment of constant values to SAC variables makes a corresponding adjoint assignment obsolete. Still, the value of the overwritten SAC variable needs to be stored in the forward section, and it must be recovered in the reverse section in order to ensure correctness of the overall adjoint code.

$$
\begin{aligned}
(P8)\quad e:C \quad e.s = 1 \\
a.c^f &= \text{“push\_d(v”} + e.i + \text{“);”} \\
&\quad + \text{“v”} + e.i + \text{“=”} + C.c^f + \text{“;”} \\
a.c^b_{e.k} &= \text{“pop\_d(v”} + e.i + \text{“);”}
\end{aligned}
$$

Control-flow statements such as branches and loops as well as the associated conditions are simply unparsed in the forward section. They have no impact on the reverse section due to the chosen conservative control-flow reversal method.

$$
\begin{aligned}
(P9)\quad b:IF(r) \\
s.k &= b.k \\
\{s\} \\
b.k &= s.k \\
b.c^f &= \text{“}\mathbf{if}\text{”} + \text{“(”} + r.c^f + \text{“)”} + \text{“\{”} s.c^f + \text{“\}”} \\
b.c^b &= s.c^b
\end{aligned}
$$

$$
\begin{aligned}
(P10)\quad l:WHILE(r) \\
s.k &= l.k \\
\{s\} \\
l.k &= s.k \\
l.c^f &= \text{“}\mathbf{while}\text{”} + \text{“(”} + r.c^f + \text{“)”} + \text{“\{”} s.c^f + \text{“\}”} \\
l.c^b &= s.c^b
\end{aligned}
$$

$$
\begin{aligned}
(P11)\quad r:V^{r_1}RV^{r_2} \\
r.c^f &= V^{r_1}.c^f + R.c^f + V^{r_2}.c^f
\end{aligned}
$$

Table 4.6 illustrates the syntax-directed synthesis of the forward and reverse sections of the adjoint code. A proof-of-concept implementation based on Listings 4.12 and 4.13 is left as an exercise.

Example 4.23 Application of the syntax-directed adjoint code compiler to

```
t=0;
while (x<t) {
  if (x<y) {
    x=y+1;
  }
  x=sin(x*y);
}
```

yields

```
push_c(0);
push_d(v1); v1=0;
push_d(t); t=v1;
```

Table 4.6. *Syntax-Directed Adjoint SAC for* $y = \sin(x * 2)$; *set* $v_{i(1)} \equiv$ v*i*_ *to establish the link with the code that is generated by the syntax-directed adjoint code compiler.*

i	$\$\$.c^f$	$\$\$.c^b$
0		
...		
11		
7	$push(v_2);\ v_2 = x;$	$pop(v_2);\ x_{(1)}\text{+=}v_{2(1)};$
13		
8	$push(v_3);\ v_3 = 2;$	$pop(v_3);$
13		
14	$push(v_2);\ v_2 = x;$	$pop(v_1);\ v_{2(1)} = v_3 * v_{1(1)};\ v_{3(1)} = v_2 * v_{1(1)};$
	$push(v_3);\ v_3 = 2;$	$pop(v_3);$
	$push(v_1);\ v_1 = v_2 * v_3;$	$pop(v_2);\ x_{(1)}\text{+=}v_{2(1)};$
11		
15		
18	$push(v_2);\ v_2 = x;$	$pop(v_0);\ v_{1(1)} = \cos(v_1) * v_{0(1)};$
	$push(v_3);\ v_3 = 2;$	$pop(v_1);\ v_{2(1)} = v_3 * v_{1(1)};\ v_{3(1)} = v_2 * v_{1(1)};$
	$push(v_1);\ v_1 = v_2 * v_3;$	$pop(v_3);$
	$push(v_0);\ v_0 = \sin(v_1);$	$pop(v_2);\ x_{(1)}\text{+=}v_{2(1)};$
4		
10		
14	$push(v_2);\ v_2 = x;$	$pop(y);\ v_{0(1)} = y_{(1)};\ y_{(1)} = 0;$
	$push(v_3);\ v_3 = 2;$	$pop(v_0);\ v_{1(1)} = \cos(v_1) * v_{0(1)};$
	$push(v_1);\ v_1 = v_2 * v_3;$	$pop(v_1);\ v_{2(1)} = v_3 * v_{1(1)};\ v_{3(1)} = v_2 * v_{1(1)};$
	$push(v_0);\ v_0 = \sin(v_1);$	$pop(v_3);$
	$push(y);\ y = v_0;$	$pop(v_2);\ x_{(1)}\text{+=}v_{2(1)};$
...		
0		

```
while (x<t) {
  if (x<y) {
    push_c(1);
    push_d(v1); v1=y;
    push_d(v2); v2=1;
    push_d(v3); v3=v1+v2;
    push_d(x); x=v3;
  }
  push_c(2);
  push_d(v1); v1=x;
  push_d(v2); v2=y;
  push_d(v3); v3=v1*v2;
  push_d(v4); v4=sin(v3);
  push_d(x); x=v4;
}
```

```
int i_;
while (pop_c(i_)) {
  if (i_==0) {
    pop_d(t); v1_=t_; t_=0;
    pop_d(v1);
  }
  else if (i_==1) {
    pop_d(x); v3_=x_; x_=0;
    pop_d(v3); v1_=v3_; v2_=v3_;
    pop_d(v2);
    pop_d(v1); y_+=v1_;
  }
  else if (i_==2) {
    pop_d(x); v4_=x_; x_=0;
    pop_d(v4); v3_=cos(v3)*v4_;
    pop_d(v3); v1_=v3_*v2; v2_=v3_*v1;
    pop_d(v2); y_+=v2_;
    pop_d(v1); x_+=v1_;
  }
}
```

Appropriate implementations of the stack access routines must be supplied. Again, the generated code must be wrapped into an appropriate function in order to run it. This wrapper must declare all program and SAC variables as well as their respective adjoints. ∎

4.6 Toward Multipass Derivative Code Compilers

Our prototype derivative code compiler `dcc` to be presented in Chapter 5 uses `flex` and `bison` to build an internal representation of the input program in the form of a parse tree and of various symbol tables. Hence, the current chapter concludes with a brief discussion of how this very basic internal representation is built. Further static program analysis can be performed based on this internal representation as outlined in Section 4.1. Domain-specific data-flow analyses, such as activity and TBR analyses [38], are performed by the "production version" of `dcc`.

A minor extension of the SL syntax is required to illustrate the construction of an internal representation in `dcc`. We consider *explicitly typed SL code* featuring variable declarations with optional initializations in addition to the previously defined syntax of SL. An example is shown in Listing 4.14.

Listing 4.14. *Simple extended SL code.*

```
double x,y,a,p;
int n,i=0;

p=y+1;
while (i<n) {
  if (i==0)
    a=sin(x*y);
  y=a*p;
  i=i+1;
}
```

There are four (double precision) floating-point variables x,y,a, and p and two integer variables n and i. The value of i is assumed to be equal to zero at the beginning of the SL code fragment.

4.6.1 Symbol Table

Symbols are described by regular expressions that are recognized by the scanner. Associated parse tree leaf nodes are generated that contain a reference (pointer) to the corresponding symbol table entries. The entire procedure is illustrated by the following fragments from the `flex` input file.

```
1 %{
2 #include "parse_tree.hpp"
3 #include "parser.tab.h"
4 ...
5 %}
6
7 ...
8 symbol              [a-z]
9 ...
10
11 %%
12 ...
13
14 {symbol} {
15    yylval=new parse_tree_vertex_symbol(SYMBOL_PTV, yytext);
16    return V;
17 }
18 ...
19
20 %%
21 ...
```

A new leaf node is added to the parse tree in line 15. Parse tree vertices that are referenced through yylval are declared as of type parse_tree_vertex * in the file `parse_tree.hpp` that is included in line 2.

```
class parse_tree_vertex {
  unsigned short type;
  list<parse_tree_vertex*> succ;
  ...
};

class parse_tree_vertex_named : public parse_tree_vertex {
  string name;
  ...
};

class parse_tree_vertex_symbol : public parse_tree_vertex {
  symbol* sym;
```

```
  ...
};

#define YYSTYPE parse_tree_vertex*
```

A specialization parse_tree_vertex_symbol of parse_tree_vertex contains a pointer to a symbol table entry that stores the name of the corresponding symbol as well as its data type.

```
class symbol {
  string name;
  int type;
  ...
};

class symbol_table {
  list<symbol*> tab;
  symbol* insert(string);
  ...
};
```

When calling the constructor of parse_tree_vertex_symbol in line 15 with the string yytext as its second argument, the name of the new symbol is set to yytext while its type is left undefined. Types of variables can only be determined after parsing the associated declaration (see Section 4.6.2). If a symbol with the same name as in yytext already exists, then the address of the existing entry is returned. The type of the newly generated parse tree vertex is set to SYMBOL_PTV. Token identifiers to be returned to the parser (for example, V returned in line 16) are defined in the file `parser.tab.h` included in line 3. After lexical analysis the parse tree consists of a number of leaf nodes that represent symbols referenced via pointers into the symbol table.

4.6.2 Parse Tree

The parser that is generated by `bison` based on the code fragments listed below performs two main tasks. It sets the types of all variables while parsing the respective declarations (see line 13) and it inserts new parse tree vertices when reducing a handle to the left-hand side of the associated production (lines 20–26).

```
 1 %{
 2 ...
 3 #include "parse_tree.hpp"
 4 ...
 5 extern parse_tree_vertex* pt_root;
 6 %}
 7 ..
 8 %%
 9
10 s1 : d s { pt_root=$2; } ;
11 d :
12    ...
13    | FLOAT V ";" d { $2->symbol_type()=FLOAT_ST; }
```

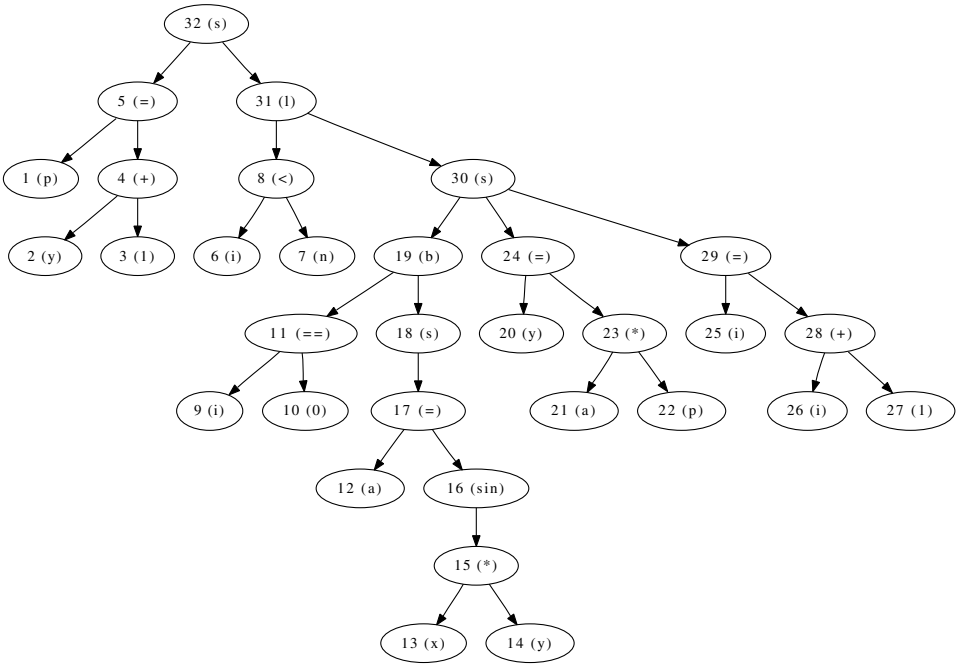


Figure 4.17. *Parse tree example for SL code.*

```
14    ;
15  ...
16 s  :  a
17  ...
18 a  :  V "=" e ";"
19  ...
20 e  :   e N e
21    {
22       if ($2->get_name()=="*")
23         $$=new parse_tree_vertex(MULTIPLICATION_PTV);
24       $$->succ.push_back($1); $$->succ.push_back($3);
25       delete $2;
26    }
27  ...
28 %%
29  ...
```

The parse tree is synthesized bottom-up by inserting new typed vertices including references to their children. A global pointer to the unique root of the parse tree is stored once the whole input program is parsed (line 10). An example parse tree is shown in Figure 4.17 for the SL code in Listing 4.14. The implementation of tangent-linear and adjoint code unparsers is reasonably straightforward. Precise descriptions are given by the attribute grammars in Section 4.5.3 and Section 4.5.4.

This chapter can only be a first step toward a comprehensive discussion of issues in derivative code compiler construction. As mentioned previously, a large number of technical challenges are caused by the various advanced syntactical and semantic concepts of modern programming languages. It is up to the users of these languages to decide which features are absolutely necessary in the context of numerical simulation software development. Existing semantic source transformation tools for numerical code rarely support entire language standards. Failure to apply these tools to a given code is often due to rather basic incompatibilities that could be avoided if code and tool development took place in parallel. Communication among both sides is crucial.

4.7 Exercises

4.7.1 Lexical Analysis

Derive DFAs for recognizing the languages that are defined by the following regular expressions

1. `0|1+(0|1)*`
2. `0+|1(0|1)+`

Implement scanners for these languages with `flex` and `gcc`. Compare the NFAs and DFAs derived by yourself with the ones that are generated by `flex`.

4.7.2 Syntax Analysis

1. Use the parser for SL^2 to parse the assignment "$y = \sin(x) + x * 2$;" as shown in Table 4.3. Draw the parse tree.
2. Extend SL^2 and its parser to include the ternary *fused-multiply-add* operation, defined as $y = \texttt{fma}(a,b,c) \equiv a * b + c$. Derive the characteristic automaton.
3. Use `flex` and `bison` to implement a parser for SL programs that prints a syntactically equivalent copy of the input code.

4.7.3 Single-Pass Derivative Code Compilers

1. Use `flex` and `bison` to implement a single-pass tangent-linear code compiler for SL^2 programs. Extend it to SL.
2. Use `flex` and `bison` to implement a single-pass adjoint code compiler for SL^2 programs. Extend it to SL.

4.7.4 Toward Multipass Derivative Code Compilers

Use `flex` and `bison` to implement a compiler that generates an intermediate representation for explicitly typed SL programs in the form of a parse tree and a symbol table. Implement an unparser.

Chapter 5

dcc—A Prototype Derivative Code Compiler

This last chapter combines the material presented in the previous chapters to form the prototype derivative code compiler `dcc`. Version 0.9 of `dcc` can be used to verify the given examples as well as to run more complex experiments. It serves as an introductory case study for more mature derivative code compilers.

5.1 Functionality

`dcc` generates jth derivative code by arbitrary combinations of tangent-linear or adjoint modes. It takes a possibly preprocessed $(j-1)$th derivative code generated by itself as input. The original (0th derivative) code is expected to be written in a well-defined subset of C++ that is intentionally kept small. Still the accepted syntax and semantics are rich enough to be able to illustrate the topics discussed in the previous chapters. See Appendix B for a summary of the syntax accepted by version 0.9 of `dcc`.

`dcc` operates on implementations of multivariate vector functions

$$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}),$$

as subroutines

```
void f(int n, int m, double *x, double *y).
```

Its results vary depending on whether certain inputs and outputs are *aliased* (represented by the same program variable) or not. Hence, the two cases

$$\mathbf{y} = F(\mathbf{x}) \quad \text{and} \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

($\mathbf{x}$ and $\mathbf{y}$ unaliased) are considered separately. For $\mathbf{y} = F(\mathbf{x})$ and $\mathbf{x}$ and $\mathbf{y}$ not aliased the generated derivative code behaves similar to what has been presented Chapters 2 and 3.

5.1.1 Tangent-Linear Code by dcc

The tangent-linear version

$$F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m : \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$$

of the given implementation of $\mathbf{y} = F(\mathbf{x})$ computes

$$\begin{aligned} \mathbf{y}^{(1)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(1)} \rangle \\ \mathbf{y} &= F(\mathbf{x}). \end{aligned}$$

For a given implementation of F as

```
void f(int n, int m, double *x, double *y),
```

dcc generates a tangent-linear subroutine with the following signature:

```
void t1_f(int n, int m, double *x, double *t1_x,
                        double *y, double *t1_y).
```

All superscripts of the tangent-linear subroutine and variable names are replaced with the prefix t1_, that is, $\mathbf{v}^{(1)} \equiv$ t1_v.

For

$$F : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^m \times \mathbb{R}^p : \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

we obtain

$$\begin{aligned} F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^p \times \mathbb{R}^p : \\ \begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \\ \mathbf{z} \\ \mathbf{z}^{(1)} \end{pmatrix} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{z}, \mathbf{z}^{(1)}), \end{aligned}$$

where

$$\begin{aligned} \begin{pmatrix} \mathbf{y}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} &= \left\langle \nabla F(\mathbf{x}, \mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle \\ \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} &= F(\mathbf{x}, \mathbf{z}). \end{aligned}$$

For a given implementation of F as

```
void f(int n, int p, int m, double *x, double *z, double *y),
```

dcc generates a tangent-linear subroutine with the following signature:

```
void t1_f(int n, int m, int p, double *x, double *t1_x,
                               double *z, double *t1_z,
                               double *y, double *t1_y).
```

The Jacobian at point

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{z} \end{pmatrix} \in \mathbb{R}^{n+p}$$

is computed by letting the input vector

$$\begin{pmatrix} \mathrm{t1\_x} \\ \mathrm{t1\_z} \end{pmatrix}$$

range over the Cartesian basis vectors in $\mathbb{R}^{n+p}$. Potential sparsity of the Jacobian should be exploited. Details of the generated tangent-linear code will be discussed in Section 5.3.

5.1.2 Adjoint Code by `dcc`

Due to missing data flow analysis, version 0.9 of `dcc` cannot decide if a value that is overwritten within the forward section of the adjoint code is required by the reverse section.[12] Conservatively, it stores all overwritten values on appropriately typed required data stacks. Hence, the straightforward application of reverse mode with data flow reversal stack s to $\mathbf{y} = F(\mathbf{x})$ yields

$$\begin{aligned} s[0] = \mathbf{y};\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y} = s[0];\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(1)} &= 0. \end{aligned}$$

The adjoint code generated by `dcc` does not return the correct function value $\mathbf{y}$ of F. It rather restores the (possibly undefined) input value of $\mathbf{y}$. To return the correct function value, code for storing a result checkpoint r must be provided by the user to save the value of $\mathbf{y}$ after the augmented forward sweep followed by recovering it after the reverse sweep:

$$\begin{aligned} s[0] &= \mathbf{y};\ \mathbf{y} = F(\mathbf{x}) \\ r[0] &= \mathbf{y} \\ \mathbf{y} &= s[0];\ \mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(1)} &= 0 \\ \mathbf{y} &= r[0]. \end{aligned}$$

Result checkpointing in `dcc` will be discussed in further detail in Section 5.4.2. For the remainder of this section we assume that the adjoint

$$\begin{aligned} F_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m &\to \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^m : \\ \begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{y} \\ \mathbf{y}_{(1)} \end{pmatrix} &= F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)}) \end{aligned}$$

[12] Version 1.0 features both activity and TBR analyses [38] in addition to a richer accepted syntax and various other source transformation techniques. The development of `dcc` is driven by ongoing collaborative research projects. Its focus is on advanced AD source transformation algorithms for the given applications rather than on coverage of the whole C/C++ standards.

of an implementation of $\mathbf{y} = F(\mathbf{x})$ features result checkpointing and hence computes

$$\begin{aligned} \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{y}_{(1)} &= 0. \end{aligned}$$

For the given implementation of F as

```
void f(int n, int m, double *x, double *y),
```

dcc generates an adjoint subroutine with the following signature:

```
void a1_f(int a1_mode, int n, int m,
          double *x, double *a1_x,
          double *y, double *a1_y).
```

All subscripts of the adjoint subroutine and variable names are replaced with the prefix a1_, that is, $\mathbf{v}_{(1)} \equiv$ a1_v. The integer parameter a1_mode selects between various modes required in the context of interprocedural adjoint code. Details will be discussed in Section 5.3.

The adjoint of $F(\mathbf{x}, \mathbf{z})$ becomes

$$F_{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \to \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m :$$
$$\begin{pmatrix} \mathbf{x} \\ \mathbf{x}_{(1)} \\ \mathbf{z} \\ \mathbf{z}_{(1)} \\ \mathbf{y} \\ \mathbf{y}_{(1)} \end{pmatrix} = F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{z}, \mathbf{z}_{(1)}, \mathbf{y}_{(1)}),$$

where

$$\begin{aligned} \begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} &= \begin{pmatrix} \mathbf{x}_{(1)} \\ 0 \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle \\ \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} &= F(\mathbf{x}, \mathbf{z}) \\ \mathbf{y}_{(1)} &= 0. \end{aligned} \tag{5.1}$$

The input value of $\mathbf{z}_{(1)}$ is overwritten instead of incremented because $\mathbf{z}$ is both an input and an output of $F(\mathbf{x}, \mathbf{z})$. Correctness of (5.1) follows immediately from the decomposition

$$\begin{aligned} \begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix} &= F(\mathbf{x}, \mathbf{z}) \\ \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} &= \begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix} \end{aligned}$$

using an auxiliary variable $\mathbf{v} \in \mathbb{R}^{m+p}$. Decomposition ensures that local inputs and outputs are mutually unaliased. Application of incremental reverse mode with required data stack s and result checkpoint r to the decomposed function yields

[augmented forward section]

$$\begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$
$$s[0] = \mathbf{z}; \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix}$$
$$r[0] = \mathbf{y}; \; r[1] = \mathbf{z}$$

[reverse section]

$$\mathbf{v}^{\mathbf{y}}_{(1)} = 0; \; \mathbf{v}^{\mathbf{z}}_{(1)} = 0$$
$$\mathbf{z} = s[0]; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(1)} \\ \mathbf{v}^{\mathbf{z}}_{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(1)} \\ \mathbf{v}^{\mathbf{z}}_{(1)} \end{pmatrix} + \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix}; \quad \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} = 0$$
$$\begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(1)} \\ \mathbf{v}^{\mathbf{z}}_{(1)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(1)} \\ \mathbf{v}^{\mathbf{z}}_{(1)} \end{pmatrix} = 0$$
$$\mathbf{y} = r[0]; \; \mathbf{z} = r[1],$$

which is easily simplified to (5.1).

For the given implementation of F as

```
void f(int n, int p, int m, double *x, double *z, double *y),
```

`dcc` generates an adjoint subroutine with the following signature:

```
void a1_f(int a1_mode, int n, int p, int m,
                       double *x, double *a1_x,
                       double *z, double *a1_z,
                       double *y, double *a1_y);
```

The Jacobian is computed by setting a1_mode = 1 and a1_x = 0 followed by letting the input vector

$$\begin{pmatrix} \text{a1\_y} \\ \text{a1\_z} \end{pmatrix}$$

range over the Cartesian basis vectors in $\mathbb{R}^{m+p}$. Potential sparsity of the Jacobian should be exploited.

5.1.3 Second-Order Tangent-Linear Code by `dcc`

`dcc` behaves exactly as described in Section 3.2 when applied in forward mode to the tangent-linear code $F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$. Application of `dcc` in forward mode to a tangent-linear code

$$\begin{pmatrix} \mathbf{y}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} = \left\langle \nabla F(\mathbf{x}, \mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle$$
$$\begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

that implements $F^{(1)}(\mathbf{x},\mathbf{x}^{(1)},\mathbf{z},\mathbf{z}^{(1)})$ yields

$$\begin{aligned} F^{(1,2)} &: \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \\ &\to \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p : \\ &\begin{pmatrix} \mathbf{y} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1,2)} \\ \mathbf{z} \\ \mathbf{z}^{(2)} \\ \mathbf{z}^{(1)} \\ \mathbf{z}^{(1,2)} \end{pmatrix} = F^{(1,2)}(\mathbf{x},\mathbf{x}^{(2)},\mathbf{x}^{(1)},\mathbf{x}^{(1,2)},\mathbf{z},\mathbf{z}^{(2)},\mathbf{z}^{(1)},\mathbf{z}^{(1,2)}), \end{aligned}$$

where

$$\begin{aligned} \begin{pmatrix} \mathbf{y}^{(1,2)} \\ \mathbf{z}^{(1,2)} \end{pmatrix} &= \left\langle \nabla F(\mathbf{x},\mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1,2)} \\ \mathbf{z}^{(1,2)} \end{pmatrix} \right\rangle + \left\langle \nabla^2 F(\mathbf{x},\mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix}, \begin{pmatrix} \mathbf{x}^{(2)} \\ \mathbf{z}^{(2)} \end{pmatrix} \right\rangle \\ \begin{pmatrix} \mathbf{y}^{(2)} \\ \mathbf{z}^{(2)} \end{pmatrix} &= \left\langle \nabla F(\mathbf{x},\mathbf{z}), \begin{pmatrix} \mathbf{x}^{(2)} \\ \mathbf{z}^{(2)} \end{pmatrix} \right\rangle \\ \begin{pmatrix} \mathbf{y}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} &= \left\langle \nabla F(\mathbf{x},\mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle \\ \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} &= F(\mathbf{x},\mathbf{z}). \end{aligned}$$

For the given tangent-linear subroutine

```
void t1_f(int n, int p, int m, double *x, double *t1_x,
                               double *z, double *t1_z,
                               double *y, double *t1_y),
```

dcc generates a second-order tangent-linear subroutine with the following signature:

```
void t2_t1_f(int n, int p, int m,
   double *x, double *t2_x, double *t1_x, double *t2_t1_x,
   double *z, double *t2_z, double *t1_z, double *t2_t1_z,
   double *y, double *t2_y, double *t1_y, double *t2_t1_y).
```

Superscripts of the second-order tangent-linear subroutine and variable names are replaced with the prefixes t2_ and t1_, that is, $\mathbf{v}^{(1,2)} \equiv$ t2_t1_v. The Hessian at point

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{z} \end{pmatrix} \in \mathbb{R}^{n+p}$$

is accumulated by setting t2_t1_x$[i] = 0$ for $i = 0,\ldots,n-1$ and t2_t1_z$[j] = 0$ for $j = 0,\ldots,p-1$ on input and by letting the input vectors

$$\begin{pmatrix} \mathrm{t1\_x} \\ \mathrm{t1\_z} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \mathrm{t2\_x} \\ \mathrm{t2\_z} \end{pmatrix}$$

range independently over the Cartesian basis vectors in $\mathbb{R}^{n+p}$.

5.1.4 Second-Order Adjoint Code by `dcc`

`dcc` supports all three modes for generating second-order adjoint code. Its output is such that an arbitrary number of reapplications are possible after some minor preprocessing.

Forward-over-Reverse Mode

`dcc` behaves exactly as described in Section 3.3 when applied in forward mode to the adjoint code $F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$. Application of `dcc` in forward mode to an adjoint code

$$\begin{aligned}\begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}_{(1)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle\\ \begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} &= F(\mathbf{x}, \mathbf{z})\\ \mathbf{y}_{(1)} &= 0\end{aligned}$$

that implements $F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{z}, \mathbf{z}_{(1)}, \mathbf{y}_{(1)})$ yields

$$\begin{aligned}F^{(2)}_{(1)} : &\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m\\ &\to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m :\end{aligned}$$

$$\begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{x}^{(2)}_{(1)}\\ \mathbf{z}\\ \mathbf{z}^{(2)}\\ \mathbf{z}_{(1)}\\ \mathbf{z}^{(2)}_{(1)}\\ \mathbf{y}\\ \mathbf{y}^{(2)}\\ \mathbf{y}_{(1)}\\ \mathbf{y}^{(2)}_{(1)}\end{pmatrix} = F^{(2)}_{(1)}(\mathbf{x}, \mathbf{x}^{(2)}, \mathbf{x}_{(1)}, \mathbf{x}^{(2)}_{(1)}, \mathbf{z}, \mathbf{z}^{(2)}, \mathbf{z}_{(1)}, \mathbf{z}^{(2)}_{(1)}, \mathbf{y}_{(1)}, \mathbf{y}^{(2)}_{(1)}),$$

where

$$\begin{aligned}\begin{pmatrix}\mathbf{x}^{(2)}_{(1)}\\ \mathbf{z}^{(2)}_{(1)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}^{(2)}_{(1)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}^{(2)}_{(1)}\\ \mathbf{z}^{(2)}_{(1)}\end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle + \left\langle \begin{pmatrix}\mathbf{y}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix}, \nabla^2 F(\mathbf{x}, \mathbf{z}), \begin{pmatrix}\mathbf{x}^{(2)}\\ \mathbf{z}^{(2)}\end{pmatrix} \right\rangle\\ \begin{pmatrix}\mathbf{y}^{(2)}\\ \mathbf{z}^{(2)}\end{pmatrix} &= \left\langle \nabla F(\mathbf{x}), \begin{pmatrix}\mathbf{x}^{(2)}\\ \mathbf{z}^{(2)}\end{pmatrix} \right\rangle\\ \mathbf{y}^{(2)}_{(1)} &= 0\\ \begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}_{(1)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle\\ \begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} &= F(\mathbf{x}, \mathbf{z})\\ \mathbf{y}_{(1)} &= 0.\end{aligned}$$

For the given adjoint subroutine

```
void a1_f(int a1_mode, int n, int p, int m,
                       double *x, double *a1_x,
                       double *z, double *a1_z,
                       double *y, double *a1_y),
```

dcc generates a second-order adjoint subroutine with the following signature:

```
void t2_a1_f(int a1_mode, int n, int p, int m,
   double *x, double *t2_x, double *a1_x, double *t2_a1_x,
   double *z, double *t2_z, double *a1_z, double *t2_a1_z,
   double *y, double *t2_y, double *a1_y, double *t2_a1_y).
```

Super- and subscripts of the second-order adjoint subroutine and variable names are replaced with the prefixes t2_ and a1_, respectively; that is, $\mathbf{v}_{(1)}^{(2)} \equiv$ t2_a1_v. The Hessian at point

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{z} \end{pmatrix} \in \mathbb{R}^{n+p}$$

is accumulated by setting t2_a1_x$[i] = 0$ for $i = 0,\ldots,n-1$, t2_a1_y$[j] = 0$ for $j = 0,\ldots,m-1$, and t2_a1_z$[k] = 0$ for $k = 0,\ldots,p-1$ on input and by letting the input vectors

$$\begin{pmatrix} \mathrm{a1\_y} \\ \mathrm{a1\_z} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \mathrm{t2\_x} \\ \mathrm{t2\_z} \end{pmatrix}$$

range independently over the Cartesian basis vectors in $\mathbb{R}^{m+p}$ and $\mathbb{R}^{n+p}$, respectively.

Reverse-over-Forward Mode

dcc behaves exactly as described in Section 3.3 when applied in reverse mode to the tangent-linear code $F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$. Application of dcc in reverse mode with required data stack s and result checkpoint r to a tangent-linear code

$$\begin{pmatrix} \mathbf{y}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} = \left\langle \nabla F(\mathbf{x}, \mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle$$
$$\begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

that implements $F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}, \mathbf{z}, \mathbf{z}^{(1)})$ yields

$$\begin{aligned} F_{(2)}^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m \\ \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m : \end{aligned}$$

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{x}^{(1)}_{(2)} \\ \mathbf{z} \\ \mathbf{z}_{(2)} \\ \mathbf{z}^{(1)} \\ \mathbf{z}^{(1)}_{(2)} \\ \mathbf{y} \\ \mathbf{y}_{(2)} \\ \mathbf{y}^{(1)} \\ \mathbf{y}^{(1)}_{(2)} \end{pmatrix} = F^{(1)}_{(2)}(\mathbf{x}, \mathbf{x}_{(2)}, \mathbf{x}^{(1)}, \mathbf{x}^{(1)}_{(2)}, \mathbf{z}, \mathbf{z}_{(2)}, \mathbf{z}^{(1)}, \mathbf{z}^{(1)}_{(2)}, \mathbf{y}_{(2)}, \mathbf{y}^{(1)}_{(2)}),$$

where

[augmented forward section]

$$\begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix} = \left\langle \nabla F(\mathbf{x}, \mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle$$

$$s[0] = \mathbf{z}^{(1)}; \quad \begin{pmatrix} \mathbf{y}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

$$s[1] = \mathbf{z}; \quad \begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}} \\ \mathbf{v}^{\mathbf{z}} \end{pmatrix}$$

$$r[0] = \mathbf{y};\ r[1] = \mathbf{z};\ r[2] = \mathbf{y}^{(1)};\ r[3] = \mathbf{z}^{(1)}$$

[reverse section]

$$\mathbf{v}^{\mathbf{y}}_{(2)} = 0;\ \mathbf{v}^{\mathbf{z}}_{(2)} = 0$$

$$\mathbf{z} = s[1]; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} + \begin{pmatrix} \mathbf{y}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix}; \quad \begin{pmatrix} \mathbf{y}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = 0$$

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = 0$$

$$\mathbf{z}^{(1)} = s[0]; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} + \begin{pmatrix} \mathbf{y}^{(1)}_{(2)} \\ \mathbf{z}^{(1)}_{(2)} \end{pmatrix}; \quad \begin{pmatrix} \mathbf{y}^{(1)}_{(2)} \\ \mathbf{z}^{(1)}_{(2)} \end{pmatrix} = 0$$

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \nabla^2 F(\mathbf{x}, \mathbf{z}), \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{z}^{(1)} \end{pmatrix} \right\rangle$$

$$\begin{pmatrix} \mathbf{x}^{(1)}_{(2)} \\ \mathbf{z}^{(1)}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^{(1)}_{(2)} \\ \mathbf{z}^{(1)}_{(2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = 0$$

$$\mathbf{y} = r[0];\ \mathbf{z} = r[1];\ \mathbf{y}^{(1)} = r[2];\ \mathbf{z}^{(1)} = r[3]$$

which is easily simplified to

$$\begin{aligned}
\begin{pmatrix}\mathbf{x}_{(2)}\\ \mathbf{z}_{(2)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}_{(2)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}_{(2)}\\ \mathbf{z}_{(2)}\end{pmatrix}, \nabla F(\mathbf{x},\mathbf{z}) \right\rangle + \left\langle \begin{pmatrix}\mathbf{y}^{(1)}_{(2)}\\ \mathbf{z}^{(1)}_{(2)}\end{pmatrix}, \nabla^2 F(\mathbf{x},\mathbf{z}), \begin{pmatrix}\mathbf{x}^{(1)}\\ \mathbf{z}^{(1)}\end{pmatrix} \right\rangle \\
\begin{pmatrix}\mathbf{y}^{(1)}\\ \mathbf{z}^{(1)}\end{pmatrix} &= \left\langle \nabla F(\mathbf{x},\mathbf{z}), \begin{pmatrix}\mathbf{x}^{(1)}\\ \mathbf{z}^{(1)}\end{pmatrix} \right\rangle \\
\begin{pmatrix}\mathbf{x}^{(1)}_{(2)}\\ \mathbf{z}^{(1)}_{(2)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}^{(1)}_{(2)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}^{(1)}_{(2)}\\ \mathbf{z}^{(1)}_{(2)}\end{pmatrix}, \nabla F(\mathbf{x},\mathbf{z}) \right\rangle \\
\begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} &= F(\mathbf{x},\mathbf{z}) \\
\mathbf{y}_{(2)} &= 0 \\
\mathbf{y}^{(1)}_{(2)} &= 0.
\end{aligned}$$

For the given tangent-linear subroutine

```
void t1_f(int n, int p, int m, double *x, double *t1_x,
                               double *z, double *t1_z,
                               double *y, double *t1_y),
```

dcc generates a second-order adjoint subroutine with the following signature:

```
void a2_t1_f(int a2_mode, int n, int p, int m,
   double *x, double *a2_x, double *t1_x, double *a2_t1_x,
   double *z, double *a2_z, double *t1_z, double *a2_t1_z,
   double *y, double *a2_y, double *t1_y, double *a2_t1_y).
```

Super- and subscripts of second-order adjoint subroutine and variable names are replaced with the prefixes t1_ and a2_, respectively; that is, $\mathbf{v}^{(1)}_{(2)} \equiv$ a2_t1_v. The Hessian at point

$$\begin{pmatrix}\mathrm{x}\\ \mathrm{z}\end{pmatrix} \in \mathbb{R}^{n+p}$$

is accumulated by setting a2_x[i] = 0 for $i = 0,\ldots,n-1$, a2_y[j] = 0 for $j = 0,\ldots,m-1$, and a2_z[k] = 0 for $k = 0,\ldots,p-1$ on input and by letting the input vectors

$$\begin{pmatrix}\text{a2\_t1\_y}\\ \text{a2\_t1\_z}\end{pmatrix} \quad \text{and} \quad \begin{pmatrix}\text{t1\_x}\\ \text{t1\_z}\end{pmatrix}$$

range independently over the Cartesian basis vectors in $\mathbb{R}^{m+p}$ and $\mathbb{R}^{n+p}$, respectively.

Reverse-over-Reverse Mode

While reverse-over-reverse mode has no relevance for practical applications its discussion is useful as it provides deeper insight into adjoint code in general. dcc behaves exactly as described in Section 3.3 when applied in reverse mode to the adjoint code $F_{(1)}(\mathbf{x},\mathbf{x}_{(1)},\mathbf{y}_{(1)})$.

Application of `dcc` in reverse mode with required data stack s and result checkpoint r to the adjoint code

$$\begin{aligned}\begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}_{(1)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix}, \nabla F(\mathbf{x})\right\rangle\\ \begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} &= F(\mathbf{x},\mathbf{z})\\ \mathbf{y}_{(1)} &= 0\end{aligned}$$

yields

$$\begin{aligned}F_{(1,2)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m \to\\ \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^m :\end{aligned}$$

$$\begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{x}_{(2)}\\ \mathbf{z}\\ \mathbf{z}_{(1)}\\ \mathbf{z}_{(2)}\\ \mathbf{z}_{(1,2)}\\ \mathbf{y}\\ \mathbf{y}_{(1)}\\ \mathbf{y}_{(2)}\\ \mathbf{y}_{(1,2)}\end{pmatrix} = F_{(1,2)}(\mathbf{x},\mathbf{x}_{(2)},\mathbf{x}_{(1)},\mathbf{x}_{(1,2)},\mathbf{z},\mathbf{z}_{(2)},\mathbf{z}_{(1)},\mathbf{z}_{(1,2)},\mathbf{y}_{(1)},\mathbf{y}_{(2)}),$$

where

[augmented forward section]

$$\begin{aligned}\begin{pmatrix}\mathbf{v}^{\mathbf{x}}\\ \mathbf{v}^{\mathbf{z}}\end{pmatrix} &= \begin{pmatrix}\mathbf{x}_{(1)}\\ 0\end{pmatrix} + \left\langle \begin{pmatrix}\mathbf{y}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix}, \nabla F(\mathbf{x}.\mathbf{z})\right\rangle\\ s[0] &= \mathbf{z}_{(1)}; \quad \begin{pmatrix}\mathbf{x}_{(1)}\\ \mathbf{z}_{(1)}\end{pmatrix} = \begin{pmatrix}\mathbf{v}^{\mathbf{x}}\\ \mathbf{v}^{\mathbf{z}}\end{pmatrix}\\ \begin{pmatrix}\mathbf{v}^{\mathbf{y}}\\ \mathbf{v}^{\mathbf{z}}\end{pmatrix} &= F(\mathbf{x},\mathbf{z})\\ s[1] &= \mathbf{z}; \quad \begin{pmatrix}\mathbf{y}\\ \mathbf{z}\end{pmatrix} = \begin{pmatrix}\mathbf{v}^{\mathbf{y}}\\ \mathbf{v}^{\mathbf{z}}\end{pmatrix}\\ s[2] &= \mathbf{y}_{(1)}; \quad \mathbf{y}_{(1)} = 0\\ r[0] &= \mathbf{x}_{(1)};\ r[1] = \mathbf{z}_{(1)};\ r[2] = \mathbf{y};\ r[3] = \mathbf{z};\ r[4] = \mathbf{y}_{(1)}\end{aligned}$$

[reverse section]

$$\begin{aligned}\mathbf{v}^{\mathbf{x}}_{(2)} &= 0;\ \mathbf{v}^{\mathbf{y}}_{(2)} = 0;\ \mathbf{v}^{\mathbf{z}}_{(2)} = 0\\ \mathbf{y}_{(1)} &= s[2]; \quad \mathbf{y}_{(1,2)} = 0\end{aligned}$$

$$\mathbf{z} = s[1]; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} + \begin{pmatrix} \mathbf{y}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix}; \quad \begin{pmatrix} \mathbf{y}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = 0$$

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{y}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = 0$$

$$\mathbf{z}_{(1)} = s[0]; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{x}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{v}^{\mathbf{x}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} + \begin{pmatrix} \mathbf{x}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix}; \quad \begin{pmatrix} \mathbf{x}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix} = 0$$

$$\mathbf{x}_{(1,2)} = \mathbf{x}_{(1,2)} + \mathbf{v}^{\mathbf{x}}_{(2)}$$

$$\begin{pmatrix} \mathbf{y}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix} = \begin{pmatrix} \mathbf{y}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{x}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \nabla F(\mathbf{x}, \mathbf{z}) \right\rangle$$

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{v}^{\mathbf{x}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix}, \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix}, \nabla^2 F(\mathbf{x}, \mathbf{z}) \right\rangle; \quad \begin{pmatrix} \mathbf{v}^{\mathbf{x}}_{(2)} \\ \mathbf{v}^{\mathbf{z}}_{(2)} \end{pmatrix} = 0$$

$$\mathbf{x}_{(1)} = r[0];\ \mathbf{z}_{(1)} = r[1];\ \mathbf{y} = r[2];\ \mathbf{z} = r[3];\ \mathbf{y}_{(1)} = r[4]$$

and hence

$$\begin{pmatrix} \mathbf{x}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(2)} \\ 0 \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{y}_{(2)} \\ \mathbf{z}_{(2)} \end{pmatrix}, \nabla F(\mathbf{x}) \right\rangle + \left\langle \begin{pmatrix} \mathbf{x}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix}, \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix}, \nabla^2 F(\mathbf{x}, \mathbf{z}) \right\rangle$$

$$\begin{pmatrix} \mathbf{y}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix} = \left\langle \begin{pmatrix} \mathbf{x}_{(1,2)} \\ \mathbf{z}_{(1,2)} \end{pmatrix}, \nabla F(\mathbf{x}) \right\rangle$$

$$\mathbf{y}_{(2)} = 0$$

$$\begin{pmatrix} \mathbf{x}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{(1)} \\ 0 \end{pmatrix} + \left\langle \begin{pmatrix} \mathbf{y}_{(1)} \\ \mathbf{z}_{(1)} \end{pmatrix}, \nabla F(\mathbf{x}) \right\rangle$$

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{z} \end{pmatrix} = F(\mathbf{x}, \mathbf{z})$$

$$\mathbf{y}_{(1)} = 0.$$

For the given adjoint subroutine

```
void a1_f(int a1_mode, int n, int p, int m,
                       double *x, double *a1_x,
                       double *z, double *a1_z,
                       double *y, double *a1_y),
```

dcc generates a second-order adjoint subroutine with the following signature:

```
void a2_a1_f(int a2_mode, int a1_mode, int n, int p, int m,
   double *x, double *a2_x, double *a1_x, double *a2_a1_x,
   double *z, double *a2_z, double *a1_z, double *a2_a1_z,
   double *y, double *a2_y, double *a1_y, double *a2_a1_y).
```

Subscripts of second-order adjoint subroutine and variable names are replaced with the prefixes a2_ and a1_, respectively; that is, $\mathbf{v}_{(1,2)} \equiv$ a2_a1_v. The Hessian at point

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{z} \end{pmatrix} \in \mathbb{R}^{n+p}$$

is accumulated by setting a2_x[i] = 0 for $i = 0,\ldots,n-1$, a2_y[j] = 0 for $j = 0,\ldots,m-1$, and a2_z[k] = 0 for $k = 0,\ldots,p-1$ on input and by letting the input vectors

$$\begin{pmatrix} \text{a1\_y} \\ \text{a1\_z} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \text{a2\_a1\_x} \\ \text{a2\_a1\_z} \end{pmatrix}$$

range independently over the Cartesian basis vectors in $\mathbb{R}^{m+p}$ and $\mathbb{R}^{n+p}$, respectively.

5.1.5 Higher Derivative Code by dcc

Repeated application of dcc to previously generated derivatives code yields tangent-linear and adjoint code of arbitrary order. The derivation of higher derivative code for $F(\mathbf{x},\mathbf{z})$ is purely mechanical and thus left as an exercise.

5.2 Installation of dcc

The compiler has been tested on various Linux platforms. Its installation files come as a compressed tar archive file dcc-0.9.tar.gz. It is unpacked into a subdirectory ./dcc-0.9, e.g., by running

```
tar -xzvf dcc-0.9.tar.gz.
```

To build the compiler enter the subdirectory ./dcc-0.9 and type

```
./configure --prefix=$(INSTALL_DIR)
make
make check
make install
```

The executable dcc can be found in $(INSTALL_DIR)/bin.

make check runs the compiler in both supported modes (tangent-linear and adjoint) on test input code stored in subdirectories of ./dcc-0.9/src/tests. The generated output is verified against a reference. An error message is generated for anything but identical matches.

5.3 Use of dcc

Let the original source code reside in a file named f.c in subdirectory $(SRC_DIR) and let the top-level directory of the dcc installation be $(DCC_DIR).

A first-order tangent-linear code is built in $(SRC_DIR) by typing

```
$(DCC_DIR)/dcc f.c 1 1.
```

The name of the source file f.c is followed by two command-line parameters for setting tangent-linear mode (1) and the order of the derivative (1). The generated code is stored in a file named t1_f.c.

A first-order adjoint code is built in $(SRC_DIR) by typing

```
$(DCC_DIR)/dcc f.c 2 1.
```

The first-order (third command-line parameter set to 1) adjoint (second command-line parameter set to 2) version of the code in f.c is stored in a file named a1_f.c.

Higher derivative code can be obtained by reapplying dcc to a previously generated derivative code in either tangent-linear or adjoint mode. Reapplication of dcc to a previously generated adjoint code a1_f.c requires running the C preprocessor on a1_f.c first as described in Section 5.4.4. For example, the second-order adjoint code t2_a1_f.c results from running

```
$(DCC_DIR)/dcc a1_f.c 1 2
```

on the preprocessed version of a1_f.c. A third derivative code can be generated, for example, by running

```
$(DCC_DIR)/dcc t2_a1_f.c 2 3.
```

The result is stored in a3_t2_a1_f.c. While reapplication of dcc in adjoint mode to a previously generated first- or higher-order adjoint model is feasible, this feature is less likely to be used in practice for reasons outlined in Chapter 3. A third-order adjoint model is best generated by running

```
$(DCC_DIR)/dcc t2_a1_f.c 1 3.
```

Nevertheless, repeated code transformations in adjoint mode have been found to be enlightening ingredients of our lecture / tutorial on "Computational Differentiation."

5.4 Intraprocedural Derivative Code by dcc

Consider a file f.c with the following content

```
// a very simple input code
void f(double& x, double& y) {
  y=sin(x);
}
```

dcc expects all **double** parameters to be passed by reference. Call by value is supported for integer parameters only. Single-line comments are not preserved in the output code. We use this trivial input code to take a closer look at the result of the semantic transformations performed by dcc. Larger inputs result in tangent-linear and adjoint code whose listing becomes unreasonable due to excessive length.

5.4.1 Tangent-Linear Code

The name t1_f of the tangent-linear routine is generated by prepending the prefix t1_ to the name of the original routine. The original parameter list is augmented with dummy variables holding directional derivatives of all **double** parameters. Both x and y receive respective tangent-linear versions t1_x and t1_y in line 1 of the following code listing.

```
1 void t1_f(double& x, double& t1_x, double& y, double& t1_y)
2 {
3   double v1_0=0;
```

```
4     double t1_v1_0=0;
5     double v1_1=0;
6     double t1_v1_1=0;
7     t1_v1_0=t1_x;
8     v1_0=x;
9     t1_v1_1=cos(v1_0)*t1_v1_0;
10    v1_1=sin(v1_0);
11    t1_y=t1_v1_1;
12    y=v1_1;
13 }
```

The original assignment is decomposed into the SAC (see Section 2.1.1) listed in lines 8, 10, and 12. Two auxiliary SAC variables v1_0 and v1_1 are declared in lines 3 and 5. dcc expects a separate declaration for each variable as well as its initialization with some constant (e.g. 0). Tangent-linear versions of both auxiliary variables are declared and initialized in lines 4 and 6. All three SAC statements are augmented with local tangent-linear models (lines 7, 9, and 11).

Auxiliary variable names are built from the base string v by appending the order of differentiation (1) and a unique counter (0, 1, ...) separated by an underscore. Potential name clashes with variables present in the original source code could be avoided by a more sophisticated naming strategy. Version 0.9 of dcc does not support such a mechanism. Its source code would need to be edited in order to replace the base string v with some alternative. The native C++ compiler can be expected to eliminate most auxiliary variables as the result of copy propagation.

A driver program/function must be supplied by the user, for example,

```
1 #include<fstream>
2 #include<cmath>
3 using namespace std;
4
5 #include "t1_f.c"
6
7 int main() {
8   ofstream t1_out("t1.out");
9   double x=1, t1_x=1;
10  double y, t1_y;
11  t1_f(x, t1_x, y, t1_y);
12  t1_out << y << " " << t1_y << endl;
13  return 0;
14 }
```

It computes the partial derivative of the output y with respect to the input x at point x=1. Relevant parts of the C++ standard library are used for file i/o (fstream) and to provide an implementation for the intrinsic sine function (cmath). Global use of the std namespace is crucial as dcc does neither accept nor generate namespace prefixes such as std::. The file t1_f.c is included into the driver in line 5 in order to make these preprocessor settings applicable to the tangent-linear output of dcc. Both the values of x and of its directional derivative t1_x are set to one at the time of their declaration in line 9, followed by declarations of the outputs y and t1_y and the call of the tangent-linear function t1_f in lines 10 and 11,

respectively. The results are written into the file t1.out for later validation. Optimistically, zero is returned to indicate an error-free execution of the driver program.

5.4.2 Adjoint Code

The adjoint routine a1_f has been edited slightly by removing parts without relevance to the intraprocedural case. Its signature is left unchanged despite the fact that the integer input parameter a1_mode could also be omitted in this situation.

```
 1 int cs[10];
 2 int csc=0;
 3 double fds[10];
 4 int fdsc=0;
 5 int ids[10];
 6 int idsc=0;
 7 #include "declare_checkpoints.inc"
 8
 9 void a1_f(int a1_mode, double& x, double& a1_x,
10                        double& y, double& a1_y)
11 {
12    double v1_0=0;
13    double a1_v1_0=0;
14    double v1_1=0;
15    double a1_v1_1=0;
16    if (a1_mode==1) {
17       cs[csc]=0; csc=csc+1;
18       fds[fdsc]=y; fdsc=fdsc+1; y=sin(x);
19 #include "f_store_results.inc"
20       while (csc>0) {
21         csc=csc-1;
22         if (cs[csc]==0) {
23           fdsc=fdsc-1; y=fds[fdsc];
24           v1_0=x;
25           v1_1=sin(v1_0);
26           a1_v1_1=a1_y; a1_y=0;
27           a1_v1_0=cos(v1_0)*a1_v1_1;
28           a1_x=a1_x+a1_v1_0;
29         }
30       }
31 #include "f_store_results.inc"
32    }
33 }
```

The adjoint function needs to be called in *first-order adjoint calling mode* a1_mode=1 to invoke the propagation of adjoints from the adjoint output a1_y to the adjoint input a1_x. Further calling modes will be added when considering call tree reversal in the interprocedural case in Section 5.6.

An augmented version of the original code enumerates basic blocks in the order of their execution (line 17; see Adjoint Code Generation Rule 5) and it saves left-hand sides of assignments before they get overwritten (line 18; see Adjoint Code Generation

Rule 3). Three global stacks are declared for this purpose with default sizes set to 10 to be adapted by the user. The sizes of both the control flow stack (cs) and the required floating-point data stack (fds) can be reduced to 1 in the given example. Counter variables csc and fdsc are declared as references to the tops of the respective stacks. Missing integer assignments make the required integer data stack (ids) in line 5, as well as its counter variable idsc in line 6, obsolete. Code for allocating memory required for the potential storage of argument and/or result checkpoints needs to be provided by the user in a file named `declare_checkpoints.inc`. In version 0.9 of `dcc`, all memory required for the data-flow reversal is allocated globally. Related issues such as thread safety of the generated adjoint code are the subject of ongoing research and development.

The reverse section of the adjoint code (lines 20 to 30) runs the adjoint basic blocks in reverse order driven by their indices retrieved one by one from the top of the control stack (lines 20 to 22). Processing of the original assignments within a basic block in reverse order starts with the recovery of the original value of the variable on the left-hand side of the assignment (line 23). An incomplete version of the assignment's SAC (without storage of the value of the right-hand side expression in the variable on the left-hand side of the original assignment; lines 24 and 25; see Adjoint Code Generation Rule 4) is built to ensure availability of all arguments of local partial derivatives potentially needed by the adjoint SAC (lines 26 to 28). The corresponding auxiliary SAC variables and their adjoints are declared in lines 12 to 15 (see Adjoint Code Generation Rule 1). `dcc` expects all local variables to be initialized, e.g., to zero. Adjoints of variables declared in the original code are incremented (line 28) while adjoints of (single-use) auxiliary variables are overwritten (lines 26 and 27). Adjoints of left-hand sides of assignments are set to zero after their use by the corresponding adjoint SAC statement (line 26; see Adjoint Code Generation Rule 2).

The user is given the opportunity to ensure the return of the correct original function value through provision of three appropriate files to be included into the adjoint code. By default, the data flow reversal mechanism restores the input values of all parameters. For example, one could store y (rescp=y;) in `f_store_results.inc` and recover it (y= rescp;) in `f_restore_results.inc` in addition to the declaration and initialization of the checkpoint (**double** rescp=0;) in `declare_checkpoints.inc`. Automation of this kind of *checkpointing* is impossible if arrays are passed as pointer parameters due to missing size information in C/C++.

The determination of sufficiently large stack sizes may turn out to be not entirely trivial. For given values of the inputs, one could check the maxima of the stack counters csc, fdsc, and idsc by insertion of

```
cout << csc << " " << fdsc << " " << ids << endl;
```

in between the augmented forward and reverse sections of the adjoint code (right before or after line 19).

Again, a driver program/function needs to be supplied by the user. For our simple example, it looks very similar to the tangent-linear driver discussed in Section 5.4.1.

```
1 #include<fstream>
2 #include<cmath>
3 using namespace std;
4
5 #include "a1_f.c"
6
```

```
 7 int main() {
 8   ofstream a1_out("a1.out");
 9   double x=1, a1_x=0;
10   double y, a1_y=1;
11   a1_f(1,x, a1_x, y, a1_y);
12   a1_out << y << " " << a1_x << endl;
13   return 0;
14 }
```

To compute the partial derivative of y with respect to x at point x = 1, the value a1_y of the adjoint of the output is set to one while the adjoint a1_x of the input needs to be initialized to zero. The correct calling mode (1) is passed to the adjoint function a1_f in line 11. In line 12, the result a1_x is written to a file for later validation. Compilation of this driver followed by linking with the C++ standard library yields a program whose execution generates the same output as the tangent-linear driver in Section 5.4.1. A typical correctness check comes in the form of a comparison of the results obtained from the tangent-linear and adjoint code, for example running

```
diff t1.out a1.out.
```

5.4.3 Second-Order Tangent-Linear Code

Application of `dcc` to `t1_f.c` in tangent-linear mode as

```
$(DCC_DIR)/dcc t1_f.c 1 2
```

yields a second-order tangent-linear code.[13] To compute the second partial derivative of the output y with respect to the input x at point x = 1, the values t1_x and t2_x of the total derivative of the input are set to one while the second total derivative t2_t1_x is set to zero.

```
 1 #include<fstream>
 2 #include<cmath>
 3 using namespace std;
 4
 5 #include "t2_t1_f.c"
 6
 7 int main() {
 8   ofstream t2t1_out("t2t1.out");
 9   double x=1,t1_x=1, t2_x=1, t2_t1_x=0
10   double y, t1_y, t2_y, t2_t1_y;
11   t2_t1_f(x, t2_x, t1_x, t2_t1_x, y, t2_y, t1_y, t2_t1_y);
12   t2t1_out << t2_t1_y << endl;
13   return 0;
14 }
```

The result t2_t1_y is written to the file `t2t1.out` for later validation, for example, by comparison with the second derivative generated by a second-order adjoint code to be discussed in the next section.

[13] Listings of second and higher derivative codes are omitted due to their considerable lengths. The reader is encouraged to generate them with `dcc`.

5.4.4 Second-Order Adjoint Code

We consider all three combinations of tangent-linear and adjoint modes to obtain second-order adjoint code with dcc.

Forward-over-Reverse Mode

A second-order adjoint code is obtained by application of dcc to a preprocessed version of a1_f.c in tangent-linear mode as

```
$(DCC_DIR)/dcc a1_f.c 1 2.
```

The C preprocessor needs to be called with the -P (inhibit generation of line markers) option to resolve all **#include** statements. Its output corresponds to the syntax accepted by dcc. As a result, the code associated with checkpointing (declarations, read and write accesses) is inlined. No argument checkpointing code is required for this simple example.

To compute the second partial derivative of the output y with respect to the input x at point x = 1, the values a1_y and t2_x are set to one while the second derivatives t2_a1_x and t2_a1_y need to be set to zero.

```
 1 #include<fstream>
 2 #include<cmath>
 3 using namespace std;
 4
 5 #include "t2_a1_f.c"
 6
 7 int main() {
 8   ofstream t2a1_out("t2a1.out");
 9   double x=1, a1_x=0, t2_x=1, t2_a1_x=0;
10   double y, a1_y=1, t2_y, t2_a1_y=0;
11   t2_a1_f(1,x, t2_x, a1_x, t2_a1_x, y, t2_y, a1_y, t2_a1_y);
12   t2a1_out << t2_a1_x << endl;
13   return 0;
14 }
```

Both y and t2_y are pure outputs and thus do not need to be initialized. The result t2_a1_x is written to the file t2a1.out for comparison with the previously generated t2t1.out.

In addition to the second-order adjoint projection t2_a1_x of the Hessian, the second-order adjoint code also computes tangent-linear and adjoint projections of the Jacobian. For the given scalar case we obtain the value of the first derivative of y with respect to x both in t2_y (tangent-linear projection) and a1_x (adjoint projection). Hence, changing both lines 12 in the previously listed drivers for the second-order tangent-linear and second-order adjoint code to

```
t2t1_out << y << t2_y << t1_y << t2_t1_y << endl;
```

and

```
t2a1_out << y << a1_x << t2_y << t2_a1_x << endl;
```

respectively, yields identical outputs

```
0.841471 0.540302 0.540302 -0.841471.
```

Spaces have been inserted for improved readability.

Reverse-over-Forward Mode

A second-order adjoint code is obtained by application of `dcc` to `t1_f.c` in adjoint mode as

```
$(DCC_DIR)/dcc t1_f.c 2 2.
```

To compute the second partial derivative of the output y with respect to the input x at point x=1, the values t1_x and a2_t1_y are set to one while the first-order adjoints a2_x and a2_y need to be set to zero.

```
#include<fstream>
#include<cmath>
using namespace std;

#include "a2_t1_f.c"

int main() {
  ofstream a2t1_out("a2t1.out");
  double x=1, t1_x=1, a2_x=0, a2_t1_x=0;
  double y, t1_y, a2_y=0, a2_t1_y=1;
  a2_t1_f(1,x, a2_x, t1_x, a2_t1_x, y, a2_y, t1_y, a2_t1_y);
  a2t1_out << a2_x << endl;
  return 0;
}
```

Both y and t1_y are pure outputs and thus do not need to be initialized. The result a2_x is written to the file `a2t1.out` for comparison with the previously generated `t2t1.out`.

In addition to the second-order adjoint projection a2_x of the Hessian the second-order adjoint code also computes tangent-linear and adjoint projections of the Jacobian. For the given scalar case we obtain the value of the first derivative of y with respect to x both in t1_y (tangent-linear projection in direction t1_x) and a2_t1_x (adjoint projection in direction a2_t1_y). Proper initialization of a2_t1_x to zero is crucial in this case.

Reverse-over-Reverse Mode

As a third alternative, second-order adjoint code is obtained by application of `dcc` to a preprocessed version of `a1_f.c` in adjoint mode as

```
$(DCC_DIR)/dcc a1_f.c 2 2.
```

The names of all global variables need to be modified in `a1_f.c` to avoid name clashes with the global variables generated by the second application of adjoint mode. This step is not automatic when working with version 0.9 of `dcc`. The user needs to change the source code in `a1_f.c` manually. This restriction is mostly irrelevant as second-order adjoint code is unlikely to be generated in reverse-over-reverse mode in practice anyway.

To compute the second partial derivative of y with respect to x at point x=1, both a1_y and a2_a1_x are set to one while a2_x and a2_y need to be initialized to zero.

```
#include<fstream>
#include<cmath>
using namespace std;
```

```
#include "a2_a1_f.c"

int main() {
  {
    ofstream a1_out("a2a1.out");
    double x=1, a2_x=0, a1_x=0, a2_a1_x=1;
    double y, a2_y=0, a1_y=1, a2_a1_y;
    a2_a1_f(1,1,x, a2_x, a1_x, a2_a1_x, y, a2_y, a1_y, a2_a1_y);
    a1_out << y << a1_x << a2_a1_y << a2_x << endl;
  }
  return 0;
}
```

Both y and a2_a1_y are pure outputs and can thus be initialized arbitrarily. The result a2_x is written to the file `a2a1.out` for comparison with the previously generated `t2t1.out`.

In addition to the second-order adjoint projection a2_x of the Hessian, the second-order adjoint code also computes adjoint projections of the Jacobian. For the given scalar case, we obtain the value of the first derivative of y with respect to x both in a1_x (adjoint projection in direction a1_y) and a2_a1_y (adjoint projection in direction a2_a1_x). Initialization of a1_x to zero is crucial in this case.

5.4.5 Higher Derivative Code

Higher derivative code is generated by repeated application of `dcc` to its own (preprocessed) output. Listings become rather lengthy even for the simplest code.

The third-order tangent-linear subroutine has the following signature:

```
t3_t2_t1_f(x, t3_x, t2_x, t3_t2_x, t1_x, t3_t1_x, t2_t1_x,
    t3_t2_t1_x, y, t3_y, t2_y, t3_t2_y, t1_y, t3_t1_y, t2_t1_y,
    t3_t2_t1_y);
```

Initialization of t1_x, t2_x, t3_x to one while all second- and third-order directional derivatives are set equal to zero yields the first (partial) derivative of y with respect to x in t1_y, t2_y, and t3_y, respectively, the second derivative in t2_t1_y, t3_t1_y, and t3_t2_y, respectively, and the third derivative in t3_t2_t1_y.

To obtain the same derivative information, the third-order adjoint routine obtained by running `dcc` in forward over forward over reverse mode is called as follows:

```
t3_t2_a1_f(1,x, t3_x, t2_x, t3_t2_x, a1_x, t3_a1_x, t2_a1_x,
    t3_t2_a1_x, y, t3_y, t2_y, t3_t2_y, a1_y, t3_a1_y, t2_a1_y,
    t3_t2_a1_y);
```

a1_y, t2_x, and t3_x need to be initialized to one while the remaining second- and third-order directional derivatives and adjoints are set equal to zero. The first derivative is returned in a1_x, t2_y, and t3_y, respectively, the second derivative in t2_a1_x, t3_a1_x, and t3_t2_y, respectively, and the third derivative in t3_t2_a1_x.

We leave the generation and use of fourth and higher derivative code to the reader.

5.5 Run Time of Derivative Code by dcc

The following implementation of (1.2) is accepted by dcc:

```
void f(int n, double *x, double &y) {
  int i=0;
  y=0;
  while (i<n) {
    y=y+x[i]*x[i];
    i=i+1;
  }
  y=y*y;
}
```

Table 5.1 quantifies the performance of the corresponding first and second derivative code generated by dcc on our reference platform. We compare the run times of n executions of the respective derivative code for $n = 2 \cdot 10^4$. Thus, we are able to quantify easily the computational complexity $\mathcal{R}$ of the derivative code relative to the cost of an original function evaluation. For example, the ratio between the run time of a single evaluation of the first-order adjoint code and the run time of a single function evaluation is $4.4/0.8 = 5.5$. We show the numbers of lines of code (loc) in the second column. Optimization of the native C++ compiler is switched off (-O0) or full optimization is applied (-O3).

The results document the superiority of the hand-written derivative code discussed in Chapters 2 and 3. A single execution of the adjoint code generated by dcc takes about five times the time of an original function evaluation. Much better performance can be observed for the tangent-linear code. However, only a single execution of the adjoint code is required to compute the gradient entries as opposed to n executions of the tangent-linear code.

The second-order tangent-linear code can be optimized very effectively by the native C++ compiler performing copy propagation and elimination of common subexpressions. The optimized second-order adjoint code generated in forward-over-reverse mode is only about 50 percent more expensive than the first-order adjoint code. Compiler optimization turns out to be less effective if reverse-over-forward mode is used. This lack is mostly due to all auxiliary variables getting pushed onto the global required data stack within the augmented forward section.

Missing native compiler optimizations decrease the performance of the generated code significantly. A second-order adjoint code generated in forward-over-reverse mode out-performs the one generated in reverse-over-forward mode. A second-order adjoint code generated in reverse-over-reverse mode turns out to be not competitive.

Table 5.1. *Run time of first and second derivative code generated by* dcc *(in seconds).*

	loc	-O0	-O3
f	10	3.6	0.8
t1_f	41	11.1	0.9
a1_f	80	23.9	4.4
t2_t1_f	177	37.7	2.1
t2_a1_f	320	71.4	6.0
a2_t1_f	236	80.8	15.3
a2_a1_f	453	181.9	73.0

We encourage the reader to run similar tests on their favorite computer architectures. Experience shows that the actual run time of (derivative) code depends significantly on the given platform consisting of the hardware, the optimizing native C++ compiler, and the implementation of the C++ standard library and other libraries used. Typically, there is plenty of room for improving automatically generated derivative code either by postprocessing or by adaptation of the source code transformation algorithms to the given platform. Pragmatically, the extent to which such optimizations pay off depends on the context. Derivative code compilers can be tuned for given applications depending on their relevance. Automatically generated derivative code can be tuned (semi-)manually for speed and memory requirement if the resulting code is used extensively over a long period of time.

5.6 Interprocedural Derivative Code by dcc

For the generation of interprocedural derivative code, dcc expects all subroutines to be provided in a single file; for example,

```
void g(double& x) {
  x=sin(x);
}

void f(double& x, double& y) {
  g(x);
  y=sqrt(x);
}.
```

This code implements a univariate vector function $x \mapsto (x, y)$.

5.6.1 Tangent-Linear Code

The generated tangent-linear code does not yield any surprises.

```
 1 void t1_g(double& x, double& t1_x)
 2 {
 3     double v1_0=0;
 4     double t1_v1_0=0;
 5     double v1_1=0;
 6     double t1_v1_1=0;
 7     t1_v1_0=t1_x;
 8     v1_0=x;
 9     t1_v1_1=cos(v1_0)*t1_v1_0;
10     v1_1=sin(v1_0);
11     t1_x=t1_v1_1;
12     x=v1_1;
13 }
14 void t1_f(double& x, double& t1_x,
15            double& y, double& t1_y)
16 {
17     double v1_0=0;
18     double t1_v1_0=0;
```

```
19    double v1_1=0;
20    double t1_v1_1=0;
21    t1_g(x,t1_x);
22    t1_v1_0=t1_x;
23    v1_0=x;
24    t1_v1_1=1/(2*sqrt(v1_0))*t1_v1_0;
25    v1_1=sqrt(v1_0);
26    t1_y=t1_v1_1;
27    y=v1_1;
28 }
```

The original call of g is replaced by its tangent-linear version t1_g in line 21. Copy propagation (elimination of auxiliary variables) and the elimination of common subexpressions (for example, sqrt(v1_0) in lines 24 and 25) is again left to the native C++ compiler.

5.6.2 Adjoint Code

dcc generates fully joint call tree reversals. Due to the considerable length of the automatically generated interprocedural adjoint code, we split the listing into three parts.

The global declarations of required data and control stacks are followed by two **#include** C-preprocessor directives in lines 7 and 8 of the following listing.

```
1 int cs[10];
2 int csc=0;
3 double fds[10];
4 int fdsc=0;
5 int ids[10];
6 int idsc=0;
7 #include "declare_checkpoints.inc"
8 #include "f.c"
```

It is the user's responsibility to declare sufficiently large stacks. Moreover, name clashes with variables declared in the original program must be avoided. The preset sizes (here 10) need to be adapted accordingly. The file declare_checkpoints.inc is extended with variable declarations required for the implementation of the subroutine argument checkpointing scheme in joint call tree reversal mode. For example,

```
double rescp=0;
double argcp=0;
```

allocates memory for storing the input value x of g that is needed for running g "out of context" in joint call tree reversal mode. As in Section 5.4.2 these declarations need to be supplied by the user since the problem of generating correct checkpoints for C++-code is statically undecidable. Sizes of vector arguments passed as pointers are generally unknown due to missing array descriptors. While the scalar case could be treated automatically it is probably not worth the effort since in numerical simulation code handled by dcc most subroutine arguments are arrays. The inclusion of the original code in line 8 is necessary as g is called within the augmented forward section of the adjoint version a1_f of f.

Adjoint subroutines can be called in three modes selected by setting the integer parameter a1_mode. The prefix a1 indicates the order of differentiation. For example, if a

third-order adjoint code is generated by reverse-over-forward-over-forward mode, then the name of this parameter becomes a3_mode. Let us first take a closer look at a1_g.

```
1 void a1_g(int a1_mode, double& x, double& a1_x) {
2    double v1_0=0;
3    double a1_v1_0=0;
4    double v1_1=0;
5    double a1_v1_1=0;
6    int save_csc=0;
7    save_csc=csc;
8    if (a1_mode==1) {
9       // augmented forward section
10      cs[csc]=0; csc=csc+1;
11      fds[fdsc]=x; fdsc=fdsc+1; x=sin(x);
12 #include "g_store_results.inc"
13      // reverse section
14      while (csc>save_csc) {
15        csc=csc-1;
16        if (cs[csc]==0) {
17          fdsc=fdsc-1; x=fds[fdsc];
18          v1_0=x;
19          v1_1=sin(v1_0);
20          a1_v1_1=a1_x; a1_x=0;
21          a1_v1_0=cos(v1_0)*a1_v1_1;
22          a1_x=a1_x+a1_v1_0;
23        }
24      }
25 #include "g_restore_results.inc"
26   }
27   if (a1_mode==2) {
28 #include "g_store_inputs.inc"
29      a1_mode=a1_mode;
30   }
31   if (a1_mode==3) {
32 #include "g_restore_inputs.inc"
33      a1_mode=a1_mode;
34   }
35 }
```

The three modes are represented by three **if** statements in lines 8, 27, and 31. If the adjoint subroutine is called with a1_mode set equal to one, then an adjoint code that is similar to the one discussed in Section 5.4.2 is executed. Note that the control flow reversal uses an additional auxiliary variable save_csc to store the state of the control stack counter (csc) in line 7, followed by stepping through the local csc − save_csc adjoint basic blocks in lines 14–24. Code for storage and recovery of g's results needs to be supplied by the user.

The two remaining adjoint calling modes invoke user-supplied code for storage (a1_mode==2) and recovery (a1_mode==3) of the subroutine's inputs. Two dummy assignments are generated in lines 29 and 33 to ensure correct syntax of the adjoint code even if no argument checkpointing code is provided, that is, if both files `g_store_inputs.inc` and `g_restore_inputs.inc` are left empty. These dummy assignments are eliminated

by the optimizing native C++ compiler. In the current example the input value x of g is saved by

```
argcp=x
```

and restored by

```
x=argcp.
```

Not saving x results in incorrect adjoints as its input value is overwritten by the call of g in line 13 of the following listing of a1_f. The adjoint a1_g would hence be called with the wrong input value for x in line 27.

```
 1 void a1_f(int a1_mode, double& x, double& a1_x,
 2                        double& y, double& a1_y) {
 3    double v1_0=0;
 4    double a1_v1_0=0;
 5    double v1_1=0;
 6    double a1_v1_1=0;
 7    int save_csc=0;
 8    save_csc=csc;
 9    if (a1_mode==1) {
10       // augmented forward section
11       cs[csc]=0; csc=csc+1;
12       a1_g(2,x,a1_x);
13       g(x);
14       fds[fdsc]=y; fdsc=fdsc+1; y=sqrt(x);
15 #include "f_store_results.inc"
16       // reverse section
17       while (csc>save_csc) {
18         csc=csc-1;
19         if (cs[csc]==0) {
20           fdsc=fdsc-1; y=fds[fdsc];
21           v1_0=x;
22           v1_1=sqrt(v1_0);
23           a1_v1_1=a1_y; a1_y=0;
24           a1_v1_0=1/(2*sqrt(v1_0))*a1_v1_1;
25           a1_x=a1_x+a1_v1_0;
26           a1_g(3,x,a1_x);
27           a1_g(1,x,a1_x);
28         }
29       }
30 #include "f_restore_results.inc"
31    }
32    if (a1_mode==2) {
33 #include "f_store_inputs.inc"
34       a1_mode=a1_mode;
35    }
36    if (a1_mode==3) {
37 #include "f_restore_inputs.inc"
38       a1_mode=a1_mode;
39    }
40 }
```

Apart from the treatment of the subroutine call in lines 12, 13, 26, and 27 the adjoint version of f is structurally similar to a1_g. Subroutine calls are preceded by the storage of their argument checkpoints within the augmented forward section (lines 12 and 13). In the reverse section, the correct arguments are restored (line 26) before the adjoint subroutine is executed (line 27). The correct result y of f is preserved by the user-provided code for storing

```
rescp=y
```

in `f_store_results.inc` and restoring

```
y=rescp
```

in `f_restore_results.inc`. Arguments of f need not be stored and recovered as f is never executed "out of context."

5.6.3 Second-Order Tangent-Linear Code

The second-order tangent-linear code contains the two second-order tangent-linear subroutines t2_t1_g and t2_t1_f adding up to a total of 123 lines of code. The call of g inside of f (of t1_g inside of t1_f) is replaced by dcc by a call to t2_t1_g inside of t2_t1_f:

```
void t2_t1_g(double& x, double& t2_x,
             double& t1_x, double& t2_t1_x) {
  ...
}

void t2_t1_f(double& x, double& t2_x,
             double& t1_x, double& t2_t1_x,
             double& y, double& t2_y,
             double& t1_y, double& t2_t1_y) {
  double v1_0=0;
  double t2_v1_0=0;
  ...
  t2_t1_g(x,t2_x,t1_x,t2_t1_x);
  ...
}
```

5.6.4 Second-Order Adjoint Code

We focus on second-order adjoint code generated in forward-over-reverse mode where reverse mode is applied to the original code followed by a forward mode transformation of the preprocessed adjoint code. As discussed in Section 2.3, the adjoint of the call tree

```
|_ f
   |_ g
```

is given by

```
|_ a1_f(RECORD)
|      |_ a1_g(STORE_INPUTS)
|      |_ g
```

```
|_ a1_f(ADJOIN)
      |_ a1_g(RESTORE_INPUTS)
      |_ a1_g(RECORD)
      |_ a1_g(ADJOIN).
```

The inputs to g are stored within the augmented forward section of f and the original g is executed. Once the propagation of adjoints within the reverse section of f reaches the point where the adjoint a1_g of g must be evaluated, the original inputs to g are restored and the augmented forward section of g is executed, followed by its reverse section. The adjoint results of a1_g are passed into the remainder of the reverse section of f.

The call tree of a second-order adjoint code constructed in forward-over-reverse mode becomes

```
|_ t2_a1_f(RECORD)
|      |_ t2_a1_g(STORE_INPUTS)
|      |_ t2_g
|_ t2_a1_f(ADJOIN)
      |_ t2_a1_g(RESTORE_INPUTS)
      |_ t2_a1_g(RECORD)
      |_ t2_a1_g(ADJOIN).
```

If a1_f returns the correct result y of f in addition to the first-order adjoint a1_x, then t2_a1_f returns both values as well as the first-order directional derivative t2_y and the second-order adjoint t2_a1_x.

5.6.5 Higher Derivative Code

The repeated application of dcc in tangent-linear mode to a given tangent-linear or adjoint code does not yield any further difficulties. Readers are encouraged to run corresponding experiments individually.

5.7 Toward Reality

So far, all variables have been scalars. This is rarely the case in reality. Hence, we consider the following implementation of (1.2).

```
void g(int n, double * x, double & y)
{
  int i=1;
  y=x[0]*x[0];
  while (i<n) {
    x[i]=y+x[i]*x[i];
    y=x[i];
    i=i+1;
  }
}

void f(int n, double * x, double & y)
{
  g(n,x,y);
  y=y*y;
}
```

While after calling f the output y still contains the desired function value, side effects have been added to illustrate certain complications that arise frequently in real world situations. In fact, f (together with g) is an implementation of the multivariate vector function $F : \mathbb{R}^n \to \mathbb{R}^{n+1}$ defined as

$$\mathbf{y} = F(\mathbf{x}) = \begin{pmatrix} x_0 \\ x_0^2 + x_1^2 \\ \vdots \\ \sum_{i=0}^{n-1} x_i^2 \\ \left(\sum_{i=0}^{n-1} x_i^2\right)^2 \end{pmatrix}.$$

Only the first component of the input vector x remains unchanged.

5.7.1 Tangent-Linear Code

The tangent-linear code in `t1_f.c` is called n times (for example, $n = 10$) by the following driver program in order to accumulate the n entries of the gradient of (1.2) one by one:

```
1 #include "t1_f.c"
2 const int n=10;
3
4 int main() {
5   double x[n], y, g[n], t1_x[n], t1_y;
6   for (int i=0;i<n;i++) {
7     init(x);
8     zero(t1_x); t1_x[i]=1;
9     t1_f(n,x,t1_x,y,t1_y);
10    g[i]=t1_y;
11  }
12  print(g,"t1.out"); return 0;
13 }
```

The input vector x needs to be reinitialized correctly prior to each call of t1_f as it is overwritten inside of t1_f. We call a subroutine init for this purpose in line 7. Similarly, the vector of directional derivatives of x as an output of f needs to be reset to the correct Cartesian basis vector in line 8 before getting passed as an input to the next call of t1_f in line 9. All elements of t1_x are reinitialized to zero by the subroutine zero except for the ith entry that is set to one. Thus, a single gradient entry is computed during each loop iteration. All entries are collected in the vector g in line 10 and the whole gradient g is printed to the file `t1.out` by calling the function print.

5.7.2 Adjoint Code

A single run of the adjoint code in `a1_f.c` is required to accumulate the n entries of the gradient. The following corresponding driver program does not yield any surprises:

```
1 #include "a1_f.c"
2 const int n=10;
3
4 int main() {
```

```
5   double x[n], y, a1_x[n], a1_y;
6   init(x); zero(a1_x); a1_y=1;
7   a1_f(1,n,x,a1_x,y,a1_y);
8   print(a1_x,"a1.out");
9   return 0;
10 }
```

For the adjoint a1_x of x to contain the gradient after running a1_f, it must be initialized to zero. This initialization is performed explicitly in line 6. An incorrectly initialized vector a1_x makes the incremental adjoint code increment the wrong values. The adjoint a1_y of the scalar output y of f is set to one followed by running a1_f. Again, the gradient is written to a file for subsequent verification.

Further user intervention is required to make this adjoint computation of the gradient a success. Correct argument checkpoints need to be defined and implemented manually. Joint call tree reversal results in a1_g being called out of context within the reverse section of a1_f. A checkpoint is needed to ensure correct input values for x. For example, the user may declare

```
double argcp_g[10];
int argcp_g_c;
```

in `a1_declare_checkpoints.inc` in order to be able to supply

```
argcp_g_c=0;
while (argcp_g_c<n) {
  argcp_g[argcp_g_c]=x[argcp_g_c];
  argcp_g_c=argcp_g_c+1;
}
```

in `g_store_inputs.inc` and

```
argcp_g_c=0;
while (argcp_g_c<n) {
  x[argcp_g_c]=argcp_g[argcp_g_c];
  argcp_g_c=argcp_g_c+1;
}
```

in `g_restore_inputs.inc`. When g is called as part of the augmented forward section of a1_f, it is preceded by the saving of its input arguments:

```
...
a1_g(2,n,x,a1_x,y,a1_y);
g(n,x,y);
... .
```

Execution of the adjoint version of g at the end of the reverse section of a1_f is preceded by the recovery of the correct input values:

```
...
a1_g(3,n,x,a1_x,y,a1_y);
a1_g(1,n,x,a1_x,y,a1_y);
... .
```

The remaining `.inc` files may remain empty unless the user wants the correct result to be returned in y. Corresponding result checkpoints need to be declared and implemented in this case. See also Section 5.6.2.

5.7.3 Second-Order Tangent-Linear Code

The main conceptual difference between the second-order tangent-linear driver routine in Section 5.4.3 and the following is the need for $O(n^2)$ runs of the second-order tangent-linear routine when computing the whole Hessian. Overwriting of x in g makes the proper reinitialization of all first- and second-order directional derivatives of x, as well as of x itself (line 9 of the listing below), prior to each call of the second-order tangent-linear routine t2_t1_f crucial.

```
1 #include "t2_t1_f.c"
2 const int n=10;
3
4 int main() {
5   double x[n], y, H[n][n];
6   double t1_x[n], t2_x[n], t2_t1_x[n], t1_y, t2_y, t2_t1_y;
7   for (int j=0;j<n;j++) {
8     for (int i=0;i<=j;i++) {
9       init(x); zero(t1_x); zero(t2_x); zero(t2_t1_x);
10      t1_x[j]=1; t2_x[i]=1;
11      t2_t1_f(n,x,t2_x,t1_x,t2_t1_x,y,t2_y,t1_y,t2_t1_y);
12      H[j][i]=H[i][j]=t2_t1_y;
13    }
14  }
15  print(H,"t2t1.out");
16  return 0;
17 }
```

Symmetry of the Hessian is exploited by computing only its upper (or lower) triangular submatrix. Again, the first-order directional derivatives of the input vector x are set to range independently over the Cartesian basis vector in $\mathbb{R}^n$ in line 10. The Hessian itself is written to a file for later comparison with the result obtained from a second-order adjoint code.

5.7.4 Second-Order Adjoint Code

While $O(n^2)$ runs of the second-order tangent-linear model are required to accumulate all entries of the Hessian, the same task can be performed by the second-order adjoint model at a computational cost that depends only linearly on n.

```
1 #include "t2_a1_f.c"
2 const int n=10;
3
4 int main() {
5   double x[n], y, H[n][n];
6   init(x);
7   double t2_x[n], a1_x[n], t2_a1_x[n], t2_y, a1_y, t2_a1_y;
8   for (int j=0;j<n;j++) {
9     t2_a1_y=0; zero(t2_a1_x); zero(a1_x);
10    a1_y=1;
11    zero(t2_x); t2_x[j]=1;
12    t2_a1_f(1,n,x,t2_x,a1_x,t2_a1_x,y,t2_y,a1_y,t2_a1_y);
```

```
13      for (int i=0;i<n;i++) H[j][i]=t2_a1_x[i];
14    }
15    print(H,"t2a1.out");
16    return 0;
17  }
```

According to Theorem 3.15, the first directional derivative of x as an input to the adjoint routine a1_f needs to range over the Cartesian basis vectors in $\mathbb{R}^n$ (line 11) to obtain the Hessian column by column. The first-order adjoint a1_y of the original output y is set to one for this purpose in line 10. The initialization of x in line 6 outside of the loop is feasible despite the fact that x is overwritten inside of `t2_a1_f.c`. The data flow reversal mechanism of `dcc` ensures that the values of variables are equal to their input values at the end of the reverse section of the adjoint code.

Hessian-vector products $\nabla^2 F \cdot v$ can be computed by a single run of the second-order adjoint code as shown in the following listing:

```
1     ...
2     t2_a1_y=0; zero(t2_a1_x); zero(a1_x);
3     init(t2_x);
4     a1_y=1;
5     t2_a1_f(1,n,x,t2_x,a1_x,t2_a1_x,y,t2_y,a1_y,t2_a1_y);
6     ...
7   }
```

The values of v are simply passed into t2_a1_f through t2_x by calling the subroutine init in line 3 of the above listing. The Hessian-vector product is returned through t2_a1_x.

To summarize, there are three issues to be taken care of by the user of `dcc`.

1. The input code needs to satisfy the syntactical and semantic constraints imposed by `dcc`'s front-end. See Chapter B for details. In particular, all subprograms must be provided in a single file.

2. The sizes of the stacks generated for the data and control flow reversal need to be adapted to the memory requirement of the adjoint code. Clashes between names generated by `dcc` (for required data and control stacks and for the associated counter variables) and names of variables present in the input program must be avoided.

3. Checkpoints need to be stored and restored correctly in interprocedural adjoint code. If a reapplication of `dcc` to the adjoint code is planned, then the checkpointing code also needs to satisfy the syntactic and semantic constraints imposed by `dcc`'s front-end.

The second point requires knowledge of upper bounds for the number of executions of basic blocks and for the number of overwrites performed on integer and floating-point variables, respectively. A possible solution is the inspection of the respective stack counters during a profiling run of the adjoint code. Setting the corresponding parameters exactly to these values allows us to evaluate correct adjoints for the same inputs that were used for the profiling runs. Different inputs may lead to different memory requirements in the presence of nontrivial flow of control. Failure to allocate sufficient memory may result in incorrect adjoints. In reality the original source code may have to be restructured to yield a feasible memory requirement in fully joint call tree reversal mode.

5.8 Projects

The range of potential exercises that involve `dcc` is very large. Many of the exercises in the previous chapters can (in fact, should) be solved with the help of `dcc`. Readers are encouraged to use the compiler with their favorite solvers, for example, for systems of nonlinear equations or for nonlinear optimization. Many small to medium size problems can be implemented in the subset of C++ that is accepted by `dcc`. Combinations of overloading and source transformation tools for AD are typically employed to handle more complicated simulation code.

Ongoing developments by various groups aim to provide derivative code compilers that cover an extended set of C/C++ language features. Complete language coverage appears to be unlikely for the foreseeable future. Refer to the AD community's web portal `www.autodiff.org` for up-to-date information on available AD tools.

Appendix A

Derivative Code by Overloading

We present parts of the dco source code implementing the scalar tangent-linear (A.1), adjoint (A.2), second-order tangent-linear (A.3), and second-order adjoint (A.4) modes of AD. Listings are restricted to a few selected arithmetic operators and intrinsic functions. Extensions are reasonably straightforward. Refer to Sections 2.1.2 (for tangent-linear mode), 2.2.2 (for adjoint mode), 3.2.2 (for second-order tangent-linear mode), and 3.3.2 (for second-order adjoint mode) for explanation of the code.

A.1 Tangent-Linear Code

Listing A.1. dco_t1s_type.hpp

```
#ifndef DCO_T1S_INCLUDED_
#define DCO_T1S_INCLUDED_

  class dco_t1s_type {
    public :
      double v; double t;
      dco_t1s_type(const double&);
      dco_t1s_type();
      dco_t1s_type& operator=(const dco_t1s_type&);
  };
  dco_t1s_type operator*(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type operator+(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type operator-(const dco_t1s_type&,
                         const dco_t1s_type&);
  dco_t1s_type sin(const dco_t1s_type&);
  dco_t1s_type cos(const dco_t1s_type&);
  dco_t1s_type exp(const dco_t1s_type&);
#endif
```

Listing A.2. `dco_t1s_type.cpp`

```
#include <cmath>
using namespace std;
#include "dco_t1s_type.hpp"

  dco_t1s_type::dco_t1s_type(const double& x): v(x), t(0) { };
  dco_t1s_type::dco_t1s_type(): v(0), t(0) { };

  dco_t1s_type& dco_t1s_type::operator=(const dco_t1s_type& x) {
    if (this==&x) return *this;
    v=x.v; t=x.t;
    return *this;
  }

  dco_t1s_type operator*(const dco_t1s_type& x1, const
      dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v*x2.v;
    tmp.t=x1.t*x2.v+x1.v*x2.t;
    return tmp;
  }

  dco_t1s_type operator+(const dco_t1s_type& x1, const
      dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v+x2.v;
    tmp.t=x1.t+x2.t;
    return tmp;
  }

  dco_t1s_type operator-(const dco_t1s_type& x1, const
      dco_t1s_type& x2) {
    dco_t1s_type tmp;
    tmp.v=x1.v-x2.v;
    tmp.t=x1.t-x2.t;
    return tmp;
  }

  dco_t1s_type sin(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=sin(x.v);
    tmp.t=cos(x.v)*x.t;
    return tmp;
  }

  dco_t1s_type cos(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=cos(x.v);
    tmp.t=-sin(x.v)*x.t;
```

```
    return tmp;
  }

  dco_t1s_type exp(const dco_t1s_type& x) {
    dco_t1s_type tmp;
    tmp.v=exp(x.v);
    tmp.t=tmp.v*x.t;
    return tmp;
  }
```

A.2 Adjoint Code

Listing A.3. `dco_a1s_type.hpp`

```
#ifndef DCO_A1S_INCLUDED_
  #define DCO_A1S_INCLUDED_
  #define DCO_A1S_TAPE_SIZE 1000000

  #define DCO_A1S_UNDEF -1
  #define DCO_A1S_CONST 0
  #define DCO_A1S_ASG 1
  #define DCO_A1S_ADD 2
  #define DCO_A1S_SUB 3
  #define DCO_A1S_MUL 4
  #define DCO_A1S_SIN 5
  #define DCO_A1S_COS 6
  #define DCO_A1S_EXP 7

  class dco_a1s_tape_entry {
    public:
      int oc;
      int arg1;
      int arg2;
      double v;
      double a;
      dco_a1s_tape_entry() : oc(DCO_A1S_UNDEF), arg1(
          DCO_A1S_UNDEF), arg2(DCO_A1S_UNDEF), v(0), a(0) {};
  };

  class dco_a1s_type {
    public:
      int va;
      double v;
      dco_a1s_type() : va(DCO_A1S_UNDEF), v(0) {};
      dco_a1s_type(const double&);
      dco_a1s_type& operator=(const dco_a1s_type&);
  };

  dco_a1s_type operator*(const dco_a1s_type&, const dco_a1s_type
      &);
```

```
  dco_a1s_type operator+(const dco_a1s_type&, const dco_a1s_type
      &);
  dco_a1s_type operator-(const dco_a1s_type&, const dco_a1s_type
      &);
  dco_a1s_type sin(const dco_a1s_type&);
  dco_a1s_type exp(const dco_a1s_type&);
  void dco_a1s_print_tape();
  void dco_a1s_interpret_tape();
  void dco_a1s_reset_tape();

#endif
```

Listing A.4. `dco_a1s_type.cpp`

```
#include <cmath>
#include <iostream>
using namespace std;
#include "dco_a1s_type.hpp"

static int dco_a1s_vac=0;
dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];

dco_a1s_type::dco_a1s_type(const double& x): v(x) {
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_CONST;
  dco_a1s_tape[dco_a1s_vac].v=x;
  va=dco_a1s_vac++;
};

dco_a1s_type& dco_a1s_type::operator=(const dco_a1s_type& x) {
  if (this==&x) return *this;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ASG;
  dco_a1s_tape[dco_a1s_vac].v=v=x.v;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  va=dco_a1s_vac++;
  return *this;
}

dco_a1s_type operator*(const dco_a1s_type& x1, const dco_a1s_type
    & x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_MUL;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type operator+(const dco_a1s_type& x1, const dco_a1s_type
    & x2) {
```

```
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_ADD;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v+x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type operator-(const dco_a1s_type& x1, const dco_a1s_type
   & x2) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SUB;
  dco_a1s_tape[dco_a1s_vac].arg1=x1.va;
  dco_a1s_tape[dco_a1s_vac].arg2=x2.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=x1.v-x2.v;
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type sin(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_SIN;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type cos(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_COS;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=cos(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}

dco_a1s_type exp(const dco_a1s_type& x) {
  dco_a1s_type tmp;
  dco_a1s_tape[dco_a1s_vac].oc=DCO_A1S_EXP;
  dco_a1s_tape[dco_a1s_vac].arg1=x.va;
  dco_a1s_tape[dco_a1s_vac].v=tmp.v=exp(x.v);
  tmp.va=dco_a1s_vac++;
  return tmp;
}

void dco_a1s_print_tape () {
  cout << "tape:" << endl;
  for (int i=0;i<dco_a1s_vac;i++)
```

```
    cout << i << ": [ " << dco_a1s_tape[i].oc << ", "
         << dco_a1s_tape[i].arg1 << ", "
         << dco_a1s_tape[i].arg2 << ", "
         << dco_a1s_tape[i].v << ", "
         << dco_a1s_tape[i].a << " ]" << endl;
}

void dco_a1s_reset_tape () {
  for (int i=0;i<dco_a1s_vac;i++)
    dco_a1s_tape[i].a=0;
  dco_a1s_vac=0;
}

void dco_a1s_interpret_tape () {
  for (int i=dco_a1s_vac;i>=0;i--) {
    switch (dco_a1s_tape[i].oc) {
      case DCO_A1S_ASG : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_ADD : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SUB : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a-=dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_MUL : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[
            dco_a1s_tape[i].arg2].v*dco_a1s_tape[i].a;
        dco_a1s_tape[dco_a1s_tape[i].arg2].a+=dco_a1s_tape[
            dco_a1s_tape[i].arg1].v*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_SIN : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=cos(dco_a1s_tape[
            dco_a1s_tape[i].arg1].v)*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_COS : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a-=sin(dco_a1s_tape[
            dco_a1s_tape[i].arg1].v)*dco_a1s_tape[i].a;
        break;
      }
      case DCO_A1S_EXP : {
        dco_a1s_tape[dco_a1s_tape[i].arg1].a+=dco_a1s_tape[i].v*
            dco_a1s_tape[i].a;
```

```
        break;
      }
    }
  }
}
```

A.3 Second-Order Tangent-Linear Code

Listing A.5. dco_t2s_t1s.hpp

```
#ifndef DCO_T2S_T1S_INCLUDED_
#define DCO_T2S_T1S_INCLUDED_

  #include "dco_t1s_type.hpp"

  class dco_t2s_t1s_type {
    public :
      dco_t1s_type v;
      dco_t1s_type t;
      dco_t2s_t1s_type(const double&);
      dco_t2s_t1s_type();
      dco_t2s_t1s_type& operator=(const dco_t2s_t1s_type&);
  };
  dco_t2s_t1s_type operator*(const dco_t2s_t1s_type&, const
      dco_t2s_t1s_type&);
  dco_t2s_t1s_type operator+(const dco_t2s_t1s_type&, const
      dco_t2s_t1s_type&);
  dco_t2s_t1s_type operator-(const dco_t2s_t1s_type&, const
      dco_t2s_t1s_type&);
  dco_t2s_t1s_type sin(const dco_t2s_t1s_type&);
  dco_t2s_t1s_type cos(const dco_t2s_t1s_type&);
  dco_t2s_t1s_type exp(const dco_t2s_t1s_type&);

#endif
```

Listing A.6. dco_t2s_t1s.cpp

```
#include "dco_t2s_t1s_type.hpp"

  dco_t2s_t1s_type::dco_t2s_t1s_type(const double& x): v(x), t(0)
      { };
  dco_t2s_t1s_type::dco_t2s_t1s_type(): v(0), t(0) { };

  dco_t2s_t1s_type& dco_t2s_t1s_type::operator=(const
      dco_t2s_t1s_type& x) {
    if (this==&x) return *this;
    v=x.v; t=x.t;
    return *this;
  }
```

```
dco_t2s_t1s_type operator*(const dco_t2s_t1s_type& x1, const
    dco_t2s_t1s_type& x2) {
  dco_t2s_t1s_type tmp;
  tmp.v=x1.v*x2.v;
  tmp.t=x1.t*x2.v+x1.v*x2.t;
  return tmp;
}

dco_t2s_t1s_type operator+(const dco_t2s_t1s_type& x1, const
    dco_t2s_t1s_type& x2) {
  dco_t2s_t1s_type tmp;
  tmp.v=x1.v+x2.v;
  tmp.t=x1.t+x2.t;
  return tmp;
}

dco_t2s_t1s_type operator-(const dco_t2s_t1s_type& x1, const
    dco_t2s_t1s_type& x2) {
  dco_t2s_t1s_type tmp;
  tmp.v=x1.v-x2.v;
  tmp.t=x1.t-x2.t;
  return tmp;
}

dco_t2s_t1s_type sin(const dco_t2s_t1s_type& x) {
  dco_t2s_t1s_type tmp;
  tmp.v=sin(x.v);
  tmp.t=cos(x.v)*x.t;
  return tmp;
}

dco_t2s_t1s_type cos(const dco_t2s_t1s_type& x) {
  dco_t2s_t1s_type tmp;
  tmp.v=cos(x.v);
  tmp.t=0-sin(x.v)*x.t;
  return tmp;
}

dco_t2s_t1s_type exp(const dco_t2s_t1s_type& x) {
  dco_t2s_t1s_type tmp;
  tmp.v=exp(x.v);
  tmp.t=tmp.v*x.t;
  return tmp;
}
```

A.4 Second-Order Adjoint Code

Listing A.7. `dco_a2s_t1s.hpp`

```
#ifndef DCO_T2S_A1S_INCLUDED_
  #define DCO_T2S_A1S_INCLUDED_

  #include "dco_t1s_type.hpp"

  #define DCO_T2S_A1S_TAPE_SIZE 1000000

  #define DCO_T2S_A1S_UNDEF -1
  #define DCO_T2S_A1S_CONST 0
  #define DCO_T2S_A1S_ASG 1
  #define DCO_T2S_A1S_ADD 2
  #define DCO_T2S_A1S_SUB 3
  #define DCO_T2S_A1S_MUL 4
  #define DCO_T2S_A1S_SIN 5
  #define DCO_T2S_A1S_COS 6
  #define DCO_T2S_A1S_EXP 7

  class dco_t2s_a1s_tape_entry {
    public:
      int oc;
      int arg1;
      int arg2;
      dco_t1s_type v;
      dco_t1s_type a;
      dco_t2s_a1s_tape_entry() : oc(0), arg1(DCO_T2S_A1S_UNDEF),
          arg2(DCO_T2S_A1S_UNDEF), v(0), a(0) {};
  };

  class dco_t2s_a1s_type {
    public:
      int va;
      dco_t1s_type v;
      dco_t2s_a1s_type() : va(DCO_T2S_A1S_UNDEF), v(0) {};
      dco_t2s_a1s_type(const double&);
      dco_t2s_a1s_type& operator=(const dco_t2s_a1s_type&);
  };

  dco_t2s_a1s_type operator*(const dco_t2s_a1s_type&, const
      dco_t2s_a1s_type&);
  dco_t2s_a1s_type operator+(const dco_t2s_a1s_type&, const
      dco_t2s_a1s_type&);
  dco_t2s_a1s_type operator-(const dco_t2s_a1s_type&, const
      dco_t2s_a1s_type&);
  dco_t2s_a1s_type sin(const dco_t2s_a1s_type&);
  dco_t2s_a1s_type exp(const dco_t2s_a1s_type&);
  void dco_t2s_a1s_print_tape();
```

```
  void dco_t2s_a1s_interpret_tape();
  void dco_t2s_a1s_reset_tape();

#endif
```

Listing A.8. `dco_a2s_t1s.cpp`

```
#include <cmath>
#include <iostream>
using namespace std;
#include "dco_t2s_a1s_type.hpp"

static int dco_t2s_a1s_vac=0;
dco_t2s_a1s_tape_entry dco_t2s_a1s_tape[DCO_T2S_A1S_TAPE_SIZE];

dco_t2s_a1s_type::dco_t2s_a1s_type(const double& x): v(x) {
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_CONST;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=x;
  va=dco_t2s_a1s_vac++;
};

dco_t2s_a1s_type& dco_t2s_a1s_type::operator=(const
    dco_t2s_a1s_type& x) {
  if (this==&x) return *this;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_ASG;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=v=x.v;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x.va;
  va=dco_t2s_a1s_vac++;
  return *this;
}

dco_t2s_a1s_type operator*(const dco_t2s_a1s_type& x1, const
    dco_t2s_a1s_type& x2) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_MUL;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x1.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg2=x2.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=x1.v*x2.v;
  tmp.va=dco_t2s_a1s_vac++;
  return tmp;
}

dco_t2s_a1s_type operator+(const dco_t2s_a1s_type& x1, const
    dco_t2s_a1s_type& x2) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_ADD;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x1.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg2=x2.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=x1.v+x2.v;
  tmp.va=dco_t2s_a1s_vac++;
```

```
  return tmp;
}

dco_t2s_a1s_type operator-(const dco_t2s_a1s_type& x1, const
    dco_t2s_a1s_type& x2) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_SUB;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x1.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg2=x2.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=x1.v-x2.v;
  tmp.va=dco_t2s_a1s_vac++;
  return tmp;
}

dco_t2s_a1s_type sin(const dco_t2s_a1s_type& x) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_SIN;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=sin(x.v);
  tmp.va=dco_t2s_a1s_vac++;
  return tmp;
}

dco_t2s_a1s_type cos(const dco_t2s_a1s_type& x) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_COS;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=cos(x.v);
  tmp.va=dco_t2s_a1s_vac++;
  return tmp;
}

dco_t2s_a1s_type exp(const dco_t2s_a1s_type& x) {
  dco_t2s_a1s_type tmp;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].oc=DCO_T2S_A1S_EXP;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].arg1=x.va;
  dco_t2s_a1s_tape[dco_t2s_a1s_vac].v=tmp.v=exp(x.v);
  tmp.va=dco_t2s_a1s_vac++;
  return tmp;
}

void dco_t2s_a1s_print_tape () {
  cout << "tape:" << endl;
  for (int i=0;i<dco_t2s_a1s_vac;i++)
    cout << i << ": [ " << dco_t2s_a1s_tape[i].oc << ", "
         << dco_t2s_a1s_tape[i].arg1 << ", "
         << dco_t2s_a1s_tape[i].arg2 << ", ("
         << dco_t2s_a1s_tape[i].v.v << ", "
         << dco_t2s_a1s_tape[i].v.t << "), ("
         << dco_t2s_a1s_tape[i].a.v << ", "
```

```
          << dco_t2s_a1s_tape[i].a.t
          << ") ]" << endl;
}

void dco_t2s_a1s_reset_tape () {
  for (int i=0;i<dco_t2s_a1s_vac;i++)
    dco_t2s_a1s_tape[i].a=0;
  dco_t2s_a1s_vac=0;
}

void dco_t2s_a1s_interpret_tape () {
  for (int i=dco_t2s_a1s_vac;i>=0;i--) {
    switch (dco_t2s_a1s_tape[i].oc) {
      case DCO_T2S_A1S_ASG : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+
            dco_t2s_a1s_tape[i].a;
        break;
      }
      case DCO_T2S_A1S_ADD : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+
            dco_t2s_a1s_tape[i].a;
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a+
            dco_t2s_a1s_tape[i].a;
        break;
      }
      case DCO_T2S_A1S_SUB : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+
            dco_t2s_a1s_tape[i].a;
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a-
            dco_t2s_a1s_tape[i].a;
        break;
      }
      case DCO_T2S_A1S_MUL : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].v*
            dco_t2s_a1s_tape[i].a;
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg2].a+
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].v*
            dco_t2s_a1s_tape[i].a;
        break;
      }
```

```
      case DCO_T2S_A1S_SIN : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+cos(
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].v)*
            dco_t2s_a1s_tape[i].a;
        break;
      }
      case DCO_T2S_A1S_COS : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a-sin(
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].v)*
            dco_t2s_a1s_tape[i].a;
        break;
      }
      case DCO_T2S_A1S_EXP : {
        dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a=
            dco_t2s_a1s_tape[dco_t2s_a1s_tape[i].arg1].a+
            dco_t2s_a1s_tape[i].v*dco_t2s_a1s_tape[i].a;
        break;
      }
    }
  }
}
```

Appendix B

Syntax of dcc Input

This appendix contains a summary of the syntax accepted by version 0.9 of `dcc`. The same information can be obtained by running `flex` and `bison` in their diagnostic modes on the respective input files `scanner.l` and `parser.y`. Refer to Chapter 4 for further explanation.

B.1 bison Grammar

```
code: sequence_of_global_declarations sequence_of_subroutines

sequence_of_subroutines: subroutine
                       | sequence_of_subroutines subroutine

subroutine: VOID SYMBOL '(' list_of_arguments ')' '{'
    sequence_of_local_declarations sequence_of_statements '}'

list_of_arguments: argument
                 | list_of_arguments ',' argument

sequence_of_asterixes: '*'
                     | sequence_of_asterixes '*'

argument: INT '&' SYMBOL
        | INT SYMBOL
        | FLOAT '&' SYMBOL
        | FLOAT sequence_of_asterixes SYMBOL
        | INT sequence_of_asterixes SYMBOL

sequence_of_global_declarations: /* empty */
                               | sequence_of_global_declarations
                                   global_declaration

sequence_of_local_declarations: /* empty */
```

```
                                | sequence_of_local_declarations
                                  local_declaration

local_declaration: FLOAT SYMBOL '=' CONSTANT ';'
                 | INT SYMBOL '=' CONSTANT ';'

global_declaration: INT SYMBOL '=' CONSTANT ';'
                  | INT SYMBOL '[' CONSTANT ']' ';'
                  | FLOAT SYMBOL '=' CONSTANT ';'
                  | FLOAT SYMBOL '[' CONSTANT ']' ';'

sequence_of_statements: statement
                      | sequence_of_statements statement

statement: assignment
         | if_statement
         | while_statement
         | subroutine_call_statement ';'

if_statement: IF '(' condition ')' '{' sequence_of_statements '}'
    else_branch

else_branch: /* empty */
           | ELSE '{' sequence_of_statements '}'

while_statement: WHILE '(' condition ')' '{'
   sequence_of_statements '}'

condition: memref_or_constant '<' memref_or_constant
         | memref_or_constant '>' memref_or_constant
         | memref_or_constant '=' '=' memref_or_constant
         | memref_or_constant '!' '=' memref_or_constant
         | memref_or_constant '>' '=' memref_or_constant
         | memref_or_constant '<' '=' memref_or_constant

subroutine_call_statement: SYMBOL '(' list_of_args ')'

assignment: memref '=' expression ';'

expression: '(' expression ')'
          | expression '*' expression
          | expression '/' expression
          | expression '+' expression
          | expression '-' expression
          | SIN '(' expression ')'
          | COS '(' expression ')'
          | EXP '(' expression ')'
          | SQRT '(' expression ')'
          | TAN '(' expression ')'
          | ATAN '(' expression ')'
```

```
          | LOG '(' expression ')'
          | POW '(' expression ',' SYMBOL ')'
          | memref
          | CONSTANT

list_of_args: memref_or_constant
            | memref_or_constant ',' list_of_args

memref_or_constant: memref
                  | CONSTANT

array_index: SYMBOL
           | CONSTANT

memref: SYMBOL
      | array_reference

array_reference: SYMBOL array_access

array_access: '[' array_index ']'

array_access: '[' array_index ']' array_access
```

B.2 flex Grammar

Whitespaces are ignored. Single-line comments are allowed starting with //. Some integer and floating-point constants are supported. Variable names start with lowercase and/or uppercase letters followed by further letters, underscores, or digits.

```
int      0|[1-9][0-9]*
float        {int}"."[0-9]*
const        {int}|{float}
symbol       ([A–Z]|[a–z])(([A–Z]|[a–z])|_|{int})*
```

Supported key words are the following: **double**, **int**, **void**, **if**, **else**, **while**, sin, cos, exp, sqrt, atan, tan, pow, and log.

Appendix C

(Hints on) Solutions

C.1 Chapter 1

C.1.1 Exercise 1.4.1

Write a C++ program that converts single precision floating-point variables into their bit representation (see Section 1.3). Investigate the effects of cancellation and rounding on the finite difference approximation of first and second derivatives of a set of functions of your choice.

The following function prints the binary representation of single as well as double and higher precision floating-point variables on *little endian architectures*:

```
template<class T>
void to_bin(T v) {
  union {
      T value;
      unsigned char bytes[sizeof(T)];
  };
  memset(&bytes, 0, sizeof(T));
  value = v;
  // assumes little endian architecture
  for (size_t i = sizeof(T);i>0;i--) {
    unsigned char pot=128;
    for (int j = 7;j>=0;j--,pot/=2)
      if (bytes[i-1]&pot)
        cout << "1";
      else
        cout << "0";
    cout << " ";
  }
  cout << endl;
}
```

It can be used to investigate numerical effects due to rounding and cancellation in finite difference approximations of first and higher derivatives as in Section 1.3.

C.1.2 Exercise 1.4.2

Apply Algorithm 1.1 *to approximate a solution* $\mathbf{y} = \mathbf{y}(\mathbf{x}_0, \mathbf{x}_1)$ *of the discrete SFI problem introduced in Example* 1.2*.*

1. *Approximate the Jacobian of the residual* $\mathbf{r} = F(\mathbf{y})$ *by finite differences. Write exact derivative code based on* (1.5) *for comparison.*

2. *Use finite differences to approximate the product of the Jacobian with a vector within a matrix-free implementation of the Newton algorithm based on Algorithm* 1.4*.*

3. *Repeat the above for further problems from the MINPACK-2 test problem collection* [5]*, for example, for the* Flow in a Channel *and* Elastic Plastic Torsion *problems.*

Listing C.1 shows an implementation of Newton's algorithm for solving the system of nonlinear equations that implements the SFI problem over the unit square for equidistant finite difference discretization with step size s^{-1}. Refer to Section 1.1.1 for details. The discrete two-dimensional domain is flattened by storing the rows of the matrix $Y = (y_{i,j})$ of the nonlinear system

$$-4 \cdot y_{i,j} + y_{i+1,j} + y_{i-1,j} + y_{i,j+1} + y_{i,j-1} = h^2 \cdot \lambda \cdot e^{y_{i,j}}$$

for $i, j = 1, \ldots, s-1$ consecutively in a vector $\mathbf{y} \in \mathbb{R}^{(s-1)\cdot(s-1)}$ as shown in the following code listing.

```
void f(int s, double * y, double * r) {
  double h=0;
  double left=0; double right=0; double up=0; double down=0.;
  double value_ij=0; double dyy=0; double drr=0;
  int i=0; int j=0; int smo=0; int idx_ij=0; int k=0;

  i=1; j=1;
  h=1.0/s;

  while (i<s) {
    j=1;
    while (j<s) {
      idx_ij=(i-1)*(s-1)+j-1;
      value_ij=y[idx_ij];
      up=0; down=0; left=0; right=0;
      smo=s-1;
      if (i!=1) { k=idx_ij-(s-1); down=y[k]; }
      if (i!=smo) { k=idx_ij+s-1; up=y[k]; }
      if (j!=1) { k=idx_ij-1; left=y[k]; }
      if (j!=smo) { k=idx_ij+1; right=y[k]; }
```

```
        dyy=(right-2*value_ij+left);
        drr=(up-2*value_ij+down);
        r[idx_ij]=dyy+drr+h*h*exp(value_ij);
        j=j+1;
      }
      i=i+1;
    }
}
```

Both Listing C.1 and Listing C.2 assume that f returns the negative residual that is required by Newton's algorithm for a constant parameter $\lambda = 1$. The given implementation is accepted as input by `dcc`. An implementation of Newton's algorithm is shown in Listing C.1.

Listing C.1. *Newton's algorithm.*

```
1  int main() {
2    const int s=50;
3    const int n=(s-1)*(s-1);
4    double y[n], delta_y[n], r[n], *J[n];
5    for (int i=0;i<n;i++) J[i]=new double[n];
6    for (int i=0;i<n;i++) y[i]=0;
7    do {
8      df(s,y,r,J);
9      Factorize(n,J);
10     double z[n];
11     FSubstitute(n,J,r,z);
12     BSubstitute(n,J,z,delta_y);
13     for (int i=0;i<n;i++) y[i]=y[i]+delta_y[i];
14     f(s,y,r);
15   } while (norm(n,r)>1e-9);
16   plot_solution(s,y);
17   for (int i=0;i<n;i++) delete [] J[i];
18   return 0;
19 }
```

At the beginning of each Newton iteration the Jacobian J of the residual r at the current point y is approximated using finite differences by the function df as shown in Listing C.2. A direct linear solver (for example, Cholesky) is used to decompose J into a lower and an upper triangular factor. Both factors overwrite the memory allocated for J. The Newton step delta_y is computed by forward and backward substitution in lines 11 and 12, and it is used to update the current point y. A reevaluation of the residual r at the new point is performed in line 14 followed by checking the convergence criterion that is defined as the Euclidean norm of r reaching 10^{-9}. A converged solution is written by the function plot_solution into a file whose format is suitable for visualization with `gnuplot`. Figure C.1 shows a corresponding plot.

Listing C.2. *Jacobian by forward finite differences.*

```
void df(int s, double* y, double *r, double **J) {
  const double h=1e-8;
  int n=(s-1)*(s-1);
  double *y_ph=new double[n];
```

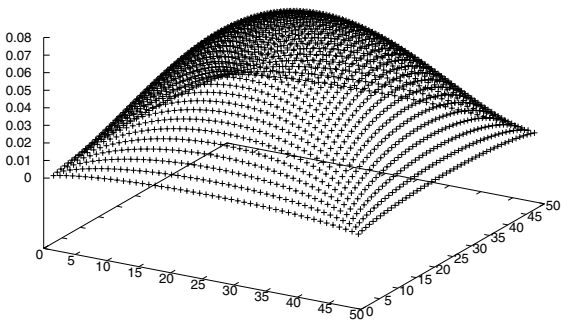


Figure C.1. *Solution of the SFI problem.*

```
  double *r_ph=new double[n];
  f(s,y,r);
  for (int i=0;i<n;i++) y_ph[i]=y[i];
  for (int i=0;i<n;i++) {
    y_ph[i]+=h;
    f(s,y_ph,r_ph);
    y_ph[i]-=h;
    for (int k=0;k<n;k++) J[i][k]=(r[k]-r_ph[k])/h;
  }
  delete [] r_ph;
  delete [] y_ph;
}
```

For the given problem specification, finite differences turn out to be rather robust with respect to changes in the size of the perturbation. The fact that f returns the negative residual is taken into account by computing the negative finite difference quotient.

The manual derivation of derivative code for the SFI problem is reasonably straightforward. In particular, there is a good check for correctness in the form of the finite difference code. This part of the exercise is meant to illustrate the disadvantages of hand-written derivative code in terms of development and debugging effort.

A matrix-free implementation of the Conjugate Gradient algorithm (Algorithm 1.4) for solving the Newton system is shown in Listing C.3. It returns the residual r and an approximation of the Newton step delta_y with accuracy eps at the current point y for given s and n.

Listing C.3. *Matrix-free CG solver for Newton system.*

```
1 void cg(double eps, int s, int n, double* y, double* r, double*
      delta_y) {
2   double* t=new double[n];
3   double* p=new double[n];
```

```
4   double* Jv=new double[n];
5   df(s,y,r,delta_y,Jv);
6   for (int i=0;i<n;i++) p[i]=t[i]=r[i]-Jv[i];
7   double normt=norm(n,t);
8   while (normt>eps) {
9     df(s,y,r,p,Jv);
10    double tTt=xTy(n,t,t);
11    double alpha=tTt/xTy(n,p,Jv);
12    axpy(n,alpha,p,delta_y,delta_y);
13    axpy(n,-alpha,Jv,t,t);
14    double beta=xTy(n,t,t)/tTt;
15    axpy(n,beta,p,t,p);
16    normt=norm(n,t);
17  }
18  delete [] Jv;
19  delete [] p;
20  delete [] t;
21 }
```

Standard inner product and axpy operations are implemented by the functions xTy and axpy, respectively. The Euclidean norm of a vector is returned by the function norm. Ideally, optimized native implementations of the Basic Linear Algebra Subprograms (BLAS) [12] should be used.

Required projections of the Jacobian in directions delta_y and p are approximated by the function df called in lines 5 and 9. A corresponding implementation of df is shown in Listing C.4. There, the current point y is perturbed in direction v in line 7. Refer to [44] for a discussion of scaling that may be required for numerically less stable problems. Again, f is assumed to return the negative residual yielding the finite difference quotient in line 9.

Listing C.4. *Projection of Jacobian by forward finite differences.*

```
1 void df(int s, double* y, double *r, double *v, double *Jv) {
2   const double h=1e-8;
3   int n=(s-1)*(s-1);
4   double *y_ph=new double[n];
5   double *r_ph=new double[n];
6   f(s,y,r);
7   for (int i=0;i<n;i++) y_ph[i]=y[i]+v[i]*h;
8   f(s,y_ph,r_ph);
9   for (int i=0;i<n;i++) Jv[i]=(r[i]-r_ph[i])/h;
10  delete [] r_ph;
11  delete [] y_ph;
12 }
```

The Newton algorithm initializes each call of cg with the Newton step computed in the previous iteration. Initially, delta_y and y are set to zero in line 6 of Listing C.5. The main loop approximates the Newton step in line 8 followed by updating the current point in line 9. Convergence is again defined as the norm of the residual reaching some upper bound. Successful termination can be observed for both the accuracies of the CG solution and of the solution of the Newton iteration itself set to 10^{-6}. Interested readers are

encouraged to investigate the behavior of the inexact Newton algorithm for increased target accuracies.

Listing C.5. *Matrix-free Newton-CG algorithm.*

```
1 int main() {
2   const int s=50;
3   const int n=(s-1)*(s-1);
4   double y[n], delta_y[n], r[n];
5   for (int i=0;i<n;i++) J[i]=new double[n];
6   for (int i=0;i<n;i++) y[i]=delta_y[i]=0;
7   do {
8     cg(1e-6,s,n,y,r,delta_y);
9     for (int i=0;i<n;i++) y[i]=y[i]+delta_y[i];
10    f(s,y,r);
11  } while (norm(n,r)>1e-6);
12  plot_solution(s,y);
13  return 0;
14 }
```

For $s = 50$ the solution of the SFI problem takes more than 11 seconds on our reference architecture when using the standard Newton algorithm with a direct linear solver. This run time grows rapidly, reaching more than 12 minutes for $s = 100$. Convergence is defined as the norm of the residual reaching 10^{-6}. The code is compiled at the highest optimization level. Less than one second is required by the matrix-free Newton-CG algorithm for $s = 100$. Its run time increases to 2.3 seconds for $s = 200$ and to 8.5 seconds for $s = 300$. The standard Newton algorithm fails to allocate enough memory for $s \geq 200$. Exploitation of sparsity is crucial in this case. A compressed Jacobian can be computed at significantly lower cost as described in Section 2.1.3. Moreover, sparse direct linear solvers are likely to decrease the memory requirement substantially.

The software infrastructure developed in this section should be applied to further MINPACK-2 test problems. Some functions may require additional parameters that can either be fixed (as done in the SFI problem for parameter `l`) or passed through slightly modified interfaces for `f`, `df`, and `cg`. (Matrix-free) Preconditioning is likely to become an issue when considering less well-conditioned problems.

C.1.3 Exercise 1.4.3

Apply the steepest descent and Newton algorithms to an extended version of the Rosenbrock function [54]*, which is defined as*

$$y = f(\mathbf{x}) \equiv \sum_{i=0}^{n-2} (1 - x_i)^2 + 10 \cdot (x_{i+1} - x_i^2)^2 \tag{C.1}$$

for $n = 10, 100, 1000$ *and for varying starting values of your choice. The function has a global minimum at* $x_i = 1$ *for* $i = 0, \ldots, n-1$*, where* $f(\mathbf{x}) = 0$*. Approximate the required derivatives by finite differences. Observe the behavior (development of function values and* L_2*-norm of gradient; run time) of the algorithms for varying values of the perturbation size. Use* (1.5) *to derive (hand-written) exact derivatives for comparison.*

The extended Rosenbrock function can be implemented as follows:

```
void f(int n, double *x, double &y) {
  double t;
  y=0;
  for (int i=0;i<n-1;i++) {
    t=x[i+1]-x[i]*x[i];
    y=y+1-2*x[i]+x[i]*x[i]+10*t*t;
  }
}
```

The implementation of the Steepest Descent algorithm shown in Listing C.6 uses iterative bisection as a local line search technique to determine the step size such that strict decrease in the objective function value is obtained.

Listing C.6. *Steepest descent algorithm.*

```
1  int main() {
2    const int n=10;
3    double x[n], y, g[n], alpha;
4    const double eps=1e-5;
5    for (int i=0;i<n;i++) x[i]=0;
6    df(n,x,y,g);
7    do {
8      alpha=1;
9      double x_t[n], y_t=y;
10     while (y_t>=y) {
11       for (int i=0;i<n;i++) x_t[i]=x[i]-alpha*g[i];
12       f(n,x_t,y_t);
13       alpha/=2;
14     }
15     for (int i=0;i<n;i++) x[i]=x_t[i];
16     df(n,x,y,g);
17   } while (norm(n,g)>eps);
18   cout << "y=" << y << endl;
19   for (int i=0;i<n;i++)
20     cout << "x[" << i << "]=" << x[i] << endl;
21   return 0;
22 }
```

The gradient is provided by calls to df in lines 6 and 16 followed by the local line search in lines 7–14 and, in case of termination of the line search, the update of the current point in line 15. Convergence is defined as the Euclidean norm of the gradient reaching or falling below 10^{-7}.

Starting from the origin for $n = 10$ a total of 1580 major and 11584 minor (local line search) iterations are performed to approximate the minimum $f(\mathbf{x}) = 0$ at $x_i = 1$ for $i = 0,\ldots,n-1$. The gradient is approximated by finite differences as shown in Listing C.7.

Listing C.7. *Gradient by forward finite differences.*

```
void df(int n, double *x, double &y, double *g) {
  const double h=1e-8;
  double *x_ph=new double[n], y_ph;
  f(n,x,y);
```

```
  for (int i=0;i<n;i++) x_ph[i]=x[i];
  for (int i=0;i<n;i++) {
    x_ph[i]+=h;
    f(n,x_ph,y_ph);
    x_ph[i]-=h;
    g[i]=(y_ph-y)/h;
  }
  delete [] x_ph;
}
```

The function df also returns the current objective function value y required by the local line search.

Steepest Descent typically requires a large number of iterations in order to reach a satisfactory level of accuracy. The corresponding run time may become infeasible very quickly for large values of n. For $n = 100$ the 1609 major and 11877 minor iterations still take less than one second on our reference architecture. While even less major (1604) and minor (11837) iterations are required for $n = 1000$, the run time increases to nearly twelve seconds due to the higher cost of the gradient approximation.

An implementation of Newton's algorithm for the minimization of the extended Rosenbrock function is shown in Listing C.8.

Listing C.8. *Newton's algorithm.*

```
int main() {
  const int n=10;
  double x[n], delta_x[n], g[n];
  double **H=new double*[n];
  for (int i=0;i<n;i++) H[i]=new double[n];

  double y;
  const double eps=1e-9;
  for (int i=0;i<n;i++) x[i]=0;
  int niters=0;
  h_g_f_cfd(n,x,y,g,H);
  if (norm(n,g)>eps)
    do {
      Factorize(n,H);
      for (int i=0;i<n;i++) g[i]=-g[i];
      double z[n];
      FSubstitute(n,H,g,z);
      BSubstitute(n,H,z,delta_x);
      for (int i=0;i<n;i++) x[i]=x[i]+delta_x[i];
      h_g_f_cfd(n,x,y,g,H);
    } while (norm(n,g)>eps);
  cout << "y=" << y << endl;
  for (int i=0;i<n;i++)
    cout << "x[" << i << "]=" << x[i] << endl;

  for (int i=0;i<n;i++) delete [] H[i];
  delete [] H;
  return 0;
}
```

At the beginning of each Newton iteration the gradient g and the Hessian H of the objective y at the current point x are approximated using finite differences by the function h_g_f_cfd as shown in Section 1.3. Cholesky decomposition is used to factorize H into a lower and an upper triangular factor. Both factors overwrite the memory allocated for H. The Newton step delta_x is computed by forward and backward substitution, and it is used to update the current point x. Convergence is defined as the Euclidean norm of g reaching 10^{-9}. A converged solution is written to the screen.

While the given configuration converges, the Hessian approximation becomes indefinite very quickly resulting in a failure to compute the Cholesky factorization. Try n=11 to observe this behavior.

C.1.4 Exercise 1.4.4

Use manual differentiation and finite differences with your favorite solver for

1. *systems of nonlinear equations to find a numerical solution of the SFI problem introduced in Section* 1.4.2*; repeat for further MINPACK-*2 *test problems;*

2. *nonlinear programming to minimize the Rosenbrock function; repeat for the other two test problems from Section* 1.4.3.

Refer to Section C.2.4 for case studies that illustrate the use of derivative code with the NAG Library. Adaptation to finite difference or manually derived code is straightforward.

C.2 Chapter 2

C.2.1 Exercise 2.4.1

1. *Write tangent-linear code for Listing C.9.*

Listing C.9. *Disputable implementation of a function.*

```
void g(int n, double* x, double& y) {
  y=1.0;
  for (int i=0;i<n;i++)
    y*=x[i]*x[i];
}

void f(int n, double* x, double &y) {
  for (int i=0;i<n;i++) x[i]=sqrt(x[i]/x[(i+1)%n]);
  g(n,x,y);
  y=cos(y);
}
```

Use the tangent-linear code to compute the Jacobian of the dependent outputs x and y with respect to the independent input x. Use central finite differences for verification.

The given code implements a function $F : \mathbb{R}^n \to \mathbb{R}^{n+1}$ as $(y, \mathbf{x}) = F(\mathbf{x})$. It features data flow dependencies of y on x[i] for i > 1 whose induced partial derivatives should vanish identically in infinite precision arithmetic. Preferably, the squaring of square roots should also be avoided; a different implementation of the underlying mathematical function should be found. Nevertheless, *algorithmic* differentiation differentiates the given algorithm. Interesting numerical effects can be observed.

An implementation of the corresponding tangent-linear model is the following:

```
void t1_g(int n, double* x, double* t1_x,
                 double& y, double& t1_y) {
  t1_y=0;
  y=1.0;
  for (int i=0;i<n;i++) {
    t1_y=2*x[i]*y*t1_x[i]+x[i]*x[i]*t1_y;
    y*=x[i]*x[i];
  }
}

void t1_f(int n, double* x, double* t1_x,
                 double &y, double& t1_y) {
  for (int i=0;i<n;i++) {
    double t1_v=t1_x[i]/x[(i+1)%n]-
        t1_x[(i+1)%n]*x[i]/(x[(i+1)%n]*x[(i+1)%n]);
    double v=x[i]/x[(i+1)%n];
    t1_x[i]=t1_v/(2*sqrt(v));
    x[i]=sqrt(v);
  }
  t1_g(n,x,t1_x,y,t1_y);
  t1_y=-sin(y)*t1_y;
  y=cos(y);
}

int main() {
  const int n=3;
  double x[n], y;
  double t1_x[n], t1_y;
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) {
      x[j]=2+cos(j);
      t1_x[j]=0;
    }
    t1_x[i]=1;
    t1_f(n,x,t1_x,y,t1_y);
    cout << t1_y << endl;
    for (int j=0;j<n;j++)
      cout << t1_x[j] << endl;
    cout << endl;
  }
  return 0;
}
```

For $n = 3$, the Jacobian $\nabla F \in \mathbb{R}^{4\times 3}$ at $\mathbf{x} = (3, 2.5403, 1.58385)^T$ becomes

$$\nabla F(\mathbf{x}) = \begin{pmatrix} -0.171085 & -0.202045 & 1.65131e-16 \\ 0.18112 & -0.213896 & 0 \\ 0 & 0.24927 & -0.399798 \\ -0.100604 & 0.11881 & 0.381113 \end{pmatrix}.$$

The first row contains the partial derivatives of y with respect to $\mathbf{x}$. Closer inspection of the computation reveals that its third entry should vanish identically in infinite precision arithmetic. We leave the bit-level exploration of the numerical effects caused by the data-flow dependence of y on x[2] to the reader; see also Section C.1.1.

Qualitatively, the results obtained from the tangent-linear code can be verified by the following central finite difference approximation:

```
int main() {
  const int n=3;
  const double h=1e-8;
  double xph[n], yph;
  double xmh[n], ymh;
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) xmh[j]=xph[j]=2+cos(j);
    xph[i]+=h;
    f(n,xph,yph);
    xmh[i]-=h;
    f(n,xmh,ymh);
    cout << (yph-ymh)/(2*h) << endl;
    for (int j=0;j<n;j++)
      cout << (xph[j]-xmh[j])/(2*h) << endl;
    cout << endl;
  }
  return 0;
}
```

The third entry in the first row of the Jacobian matrix is approximated as $2.77556e-08$. Truncation amplifies the previously observed numerical effects even further.

2. *Write adjoint code for*

```
void g(int n, double* x, double& y) {
  double l;
  int i=0;
  y=0;
  while (i<n) {
    l=x[i];
    y+=x[i]*l;
    i=i+1;
  }
}
```

and use it for the computation of the gradient of the dependent output y with respect to the independent input x. Apply backward finite differences for verification.

The given subroutine g implements a function $f : \mathbb{R}^n \to \mathbb{R}$ as $y = f(\mathbf{x})$. An implementation of the corresponding adjoint model is the following:

```
stack<double> fds;

void a1_g(int n, double* x, double *a1_x,
                 double& y, double& a1_y) {
  double l, a1_l=0;
  int i=0;
  // augmented forward section
  y=0;
  while (i<n) {
    fds.push(l);
    l=x[i];
    y+=x[i]*l;
    i=i+1;
  }

  // reverse section
  while (i>0) {
    i=i-1;
    a1_x[i]+=l*a1_y;
    a1_l+=x[i]*a1_y;
    l=fds.top(); fds.pop();
    a1_x[i]+=a1_l; a1_l=0;
  }
  a1_y=0;
}
```

The linear dependence of the output on intermediate values of y makes storage of the latter on the floating-point data stack obsolete. Conservatively, the overwritten values of l need to be saved due to the nonlinear dependence of y on l.

The main routine computes the desired gradient:

```
int main() {
  const int n=3;
  double x[n], a1_x[n], y, a1_y=1;
  for (int i=0;i<n;i++) {
    x[i]=1./(1.+i);
    a1_x[i]=0;
  }
  a1_g(n,x,a1_x,y,a1_y);
  for (int i=0;i<n;i++)
    cout << a1_x[i] << endl;
  return 0;
}
```

For $n = 3$, the gradient $\nabla f \in \mathbb{R}^3$ at point $\mathbf{x} = (1, 0.5, 333333)^T$ becomes $\nabla f(\mathbf{x}) = (2, 1, 0.666667)^T$ as successfully verified by the following backward finite difference approximation:

```
int main() {
  const int n=3;
  const double h=1e-8;
  double x[n], y, xmh[n], ymh;
  for (int i=0;i<n;i++) xmh[i]=x[i]=1./(1.+i);
  g(n,x,y);
  for (int i=0;i<n;i++) {
    xmh[i]-=h;
    g(n,xmh,ymh);
    xmh[i]+=h;
    cout << (y-ymh)/h << endl;
  }
  return 0;
}
```

3. *Write adjoint code (split mode) for the example code in Listing* C.9. *Use the adjoint code to accumulate the gradient of the dependent output y with respect to the independent input x. Ensure that the correct function values are returned in addition to the gradient.*

   The given code is regarded as a function $f : \mathbb{R}^n \to \mathbb{R}$, $y = f(\mathbf{x})$. An implementation of the corresponding adjoint model in split call tree reversal mode is the following:

```
stack<double> fds;

void a1_g(int a1_mode, int n,
          double* x, double* a1_x, double& y, double& a1_y)
              {
  if (a1_mode==1) { // augmented forward section
    y=1.0;
    for (int i=0;i<n;i++) {
      fds.push(y);
      y*=x[i]*x[i];
    }
  }
  else { // reverse section
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=2*x[i]*y*a1_y;
      a1_y=x[i]*x[i]*a1_y;
    }
    a1_y=0;
  }
}

void a1_f(int n, double* x, double* a1_x,
                 double &y, double& a1_y) {
  // augmented forward section
  for (int i=0;i<n;i++) {
    fds.push(x[i]);
```

```
    x[i]=sqrt(x[i]/x[(i+1)%n]);
  }
  a1_g(1,n,x,a1_x,y,a1_y);
  fds.push(y);
  y=cos(y);
  double res_cp=y;
  // reverse section
  y=fds.top(); fds.pop();
  a1_y=-sin(y)*a1_y;
  a1_g(2,n,x,a1_x,y,a1_y);
  for (int i=n-1;i>=0;i--) {
    x[i]=fds.top(); fds.pop();
    double v=x[i]/x[(i+1)%n];
    double a1_v=a1_x[i]/(2*sqrt(v)); a1_x[i]=0;
    a1_x[i]+=a1_v/x[(i+1)%n];
    a1_x[(i+1)%n]-=a1_v*x[i]/(x[(i+1)%n]*x[(i+1)%n]);
  }
  y=res_cp;
}
```

For $n = 3$, the gradient $\nabla f \in \mathbb{R}^3$ at $\mathbf{x} = (3, 2.5403, 1.58385)^T$ becomes $\nabla f(\mathbf{x}) = (-0.171085, -0.202045, 0)^T$. Result checkpointing in a1_f ensures the return of the correct function value $f(\mathbf{x}) = -0.928296$. The floating-point data stack grows to a maximum of seven.

A central finite difference approximation with $h = 10^{-8}$ returns

$$\nabla f(\mathbf{x}) = (-0.171085, -0.202045, 2.77556e-08)^T$$

due to the data-flow dependence of y on x[2].

4. *Write adjoint code (joint mode) for the example code in Listing* C.9. *Use it to accumulate the gradient of the dependent output y with respect to the independent input x. Correct function values need not be returned.*

   An implementation of the corresponding adjoint model in joint call tree reversal mode is the following:

```
stack<double> fds;
stack<double> arg_cp_g;

void a1_g(int a1_mode, int n,
          double* x, double* a1_x,
          double& y, double& a1_y) {
  if (a1_mode==1) { // joint augmented forward and ...
    y=1.0;
    for (int i=0;i<n;i++) {
      fds.push(y);
      y*=x[i]*x[i];
    }
    // ... reverse section
```

```
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=2*x[i]*y*a1_y;
      a1_y=x[i]*x[i]*a1_y;
    }
    a1_y=0;
  }
  else if (a1_mode==3) { // store inputs
    for (int i=0;i<n;i++) arg_cp_g.push(x[i]);
  }
  else if (a1_mode==4) { // restore inputs
    for (int i=n-1;i>=0;i--) {
      x[i]=arg_cp_g.top(); arg_cp_g.pop();
    }
  }
}

void a1_f(int n, double* x, double* a1_x,
                 double &y, double& a1_y) {
  // augmented forward section
  for (int i=0;i<n;i++) {
    fds.push(x[i]);
    x[i]=sqrt(x[i]/x[(i+1)%n]);
  }
  a1_g(3,n,x,a1_x,y,a1_y);
  g(n,x,y);
  fds.push(y);
  y=cos(y);
  // reverse section
  y=fds.top(); fds.pop();
  a1_y=-sin(y)*a1_y;
  a1_g(4,n,x,a1_x,y,a1_y);
  a1_g(1,n,x,a1_x,y,a1_y);
  for (int i=n-1;i>=0;i--) {
    x[i]=fds.top(); fds.pop();
    double v=x[i]/x[(i+1)%n];
    double a1_v=a1_x[i]/(2*sqrt(v)); a1_x[i]=0;
    a1_x[i]+=a1_v/x[(i+1)%n];
    a1_x[(i+1)%n]-=a1_v*x[i]/(x[(i+1)%n]*x[(i+1)%n]);
  }
}
```

For $n = 3$, the same gradient $\nabla f \in \mathbb{R}^3$ as in split mode is computed at the point $\mathbf{x} = (3, 2.5403, 1.58385)^T$. The incorrect function value $y = 1$ is returned due to missing result checkpointing in a1_f. The maximum size of the required floating-point data stack is reduced to six at the expense of an additional evaluation of g. With the size of an argument checkpoint not exceeding the size of the local floating-point data stack of a1_g, the maximum overall memory requirement for the data flow reversal remains equal to six **double** variables.

5. *Use the adjoint code developed under* 3 *and* 4 *to compute the gradient of the dependent output* $x[0]$ *with respect to the independent input* x*. Optimize the adjoint code by eliminating obsolete (dead) statements.*

   If the given code is regarded as a function $f : \mathbb{R}^n \to \mathbb{R}$, $x_0 = f(\mathbf{x})$, then many statements in the original adjoint code become obsolete. The only active assignment is

   ```
   x[0]= sqrt(x[0]/x[1%n]);
   ```

   yielding the gradient $\nabla f(\mathbf{x}) = (0.18112, -0.213896, 0)^T$ at $\mathbf{x} = (3, 2.5403, 1.58385)^T$. It is equal to the second row of the Jacobian computed under 1.

C.2.2 Exercise 2.4.2

Consider an implementation of the discrete residual $\mathbf{r} = F(\mathbf{y})$ *for the SFI problem introduced in Example* 1.2*.*

1. *Implement the tangent-linear model* $\mathbf{r}^{(1)} = \nabla F(\mathbf{y}) \cdot \mathbf{y}^{(1)}$ *by writing a tangent-linear code by hand and use it to accumulate* $\nabla F(\mathbf{y})$ *with machine accuracy. Verify the numerical results with finite differences.*

   We consider a flattened version of the residual of the SFI problem similar to the solution of Exercise 1.4.2.

   ```
   void f(int s, double * y, double l, double * r) {
     double left,right,up,down,dyy,drr;
     int idx_ij=0;

     for (int i=1;i<s;i++)
       for (int j=1;j<s;j++) {
         idx_ij=(i-1)*(s-1)+j-1;
         up=0; down=0; left=0; right=0;
         if (i!=1) down=y[idx_ij-(s-1)];
         if (i!=s-1) up=y[idx_ij+s-1];
         if (j!=1) left=y[idx_ij-1];
         if (j!=s-1) right=y[idx_ij+1];
         dyy=right-2*y[idx_ij]+left;
         drr=up-2*y[idx_ij]+down;
         r[idx_ij]=-dyy-drr-l*exp(y[idx_ij])/(s*s);
       }
   }
   ```

   A tangent-linear version is the following:

   ```
   void t1_f(int s, double * y, double * t1_y, double l,
                    double * r, double * t1_r) {
     double left,right,up,down,dyy,drr;
     double t1_left,t1_right,t1_up,t1_down,t1_dyy,t1_drr;
     int idx_ij=0;
   ```

```
  for (int i=1;i<s;i++)
    for (int j=1;j<s;j++) {
      idx_ij=(i-1)*(s-1)+j-1;
      t1_up=0; t1_down=0; t1_left=0; t1_right=0;
      up=0; down=0; left=0; right=0;
      if (i!=1) {
        t1_down=t1_y[idx_ij-(s-1)];
        down=y[idx_ij-(s-1)];
      }
      if (i!=s-1) {
        t1_up=t1_y[idx_ij+s-1];
        up=y[idx_ij+s-1];
      }
      if (j!=1) {
        t1_left=t1_y[idx_ij-1];
        left=y[idx_ij-1];
      }
      if (j!=s-1) {
        t1_right=t1_y[idx_ij+1];
        right=y[idx_ij+1];
      }
      t1_dyy=t1_right-2*t1_y[idx_ij]+t1_left;
      dyy=right-2*y[idx_ij]+left;
      t1_drr=t1_up-2*t1_y[idx_ij]+t1_down;
      drr=up-2*y[idx_ij]+down;
      t1_r[idx_ij]=-t1_dyy-t1_drr
                   -t1_y[idx_ij]*l*exp(y[idx_ij])/(s*s);
      r[idx_ij]=-dyy-drr-l*exp(y[idx_ij])/(s*s);
    }
}
```

The Jacobian of the residual with respect to y can be computed as follows:

```
int main() {
  const int s=5;
  const int n=(s-1)*(s-1);
  const double l=1;
  double y[n], r[n], t1_y[n], t1_r[n];
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++)
      y[j]=t1_y[j]=r[j]=t1_r[j]=0;
    t1_y[i]=1;
    t1_f(s,y,t1_y,l,r,t1_r);
    for (int j=0;j<n;j++) cout << t1_r[j] << " ";
    cout << endl;
  }
  return 0;
}
```

Sparsity is not taken into account; it should be exploited. Correctness of the tangent-linear code is easily verified by finite differences.

2. *Implement the adjoint model* $\mathbf{y}_{(1)} = \mathbf{y}_{(1)} + \nabla F(\mathbf{y})^T \cdot \mathbf{r}_{(1)}$ *by writing an adjoint code by hand and use it to accumulate* $\nabla F(\mathbf{y})$ *with machine accuracy. Compare the numerical results with those obtained by the tangent-linear approach.*

   An adjoint version of the same implementation as considered under 1 is the following:

```
void a1_f(int s, double * y, double * a1_y, double l,
                 double * r, double * a1_r) {
  double left,right,up,down,dyy,drr;
  double a1_left=0,a1_right=0,a1_up=0;
  double a1_down=0,a1_dyy=0,a1_drr=0;
  int idx_ij=0;

  // augmented forward section
  for (int i=1;i<s;i++)
    for (int j=1;j<s;j++) {
      idx_ij=(i-1)*(s-1)+j-1;
      up=0; down=0; left=0; right=0;
      if (i!=1) down=y[idx_ij-(s-1)];
      if (i!=s-1) up=y[idx_ij+s-1];
      if (j!=1) left=y[idx_ij-1];
      if (j!=s-1) right=y[idx_ij+1];
      dyy=right-2*y[idx_ij]+left;
      drr=up-2*y[idx_ij]+down;
      r[idx_ij]=-dyy-drr-l*exp(y[idx_ij])/(s*s);
    }
  // reverse section
  for (int i=s-1;i>0;i--)
    for (int j=s-1;j>0;j--) {
      idx_ij=(i-1)*(s-1)+j-1;

      a1_dyy-=a1_r[idx_ij];
      a1_drr-=a1_r[idx_ij];
      a1_y[idx_ij]-=l*exp(y[idx_ij])/(s*s)*a1_r[idx_ij];
      a1_r[idx_ij]=0;

      a1_up+=a1_drr;
      a1_y[idx_ij]-=2*a1_drr;
      a1_down+=a1_drr;
      a1_drr=0;

      a1_right+=a1_dyy;
      a1_y[idx_ij]-=2*a1_dyy;
      a1_left+=a1_dyy;
      a1_dyy=0;

      if (j!=s-1) { a1_y[idx_ij+1]+=a1_right; a1_right=0; }
      if (j!=1) { a1_y[idx_ij-1]+=a1_left; a1_left=0; }
      if (i!=s-1) { a1_y[idx_ij+s-1]+=a1_up; a1_up=0; }
      if (i!=1) { a1_y[idx_ij-(s-1)]+=a1_down; a1_down=0; }
```

```
      a1_up=0; a1_down=0; a1_left=0; a1_right=0;
    }
}
```

The Jacobian of the residual with respect to y can be computed as follows:

```
int main() {
  const int s=5;
  const int n=(s-1)*(s-1);
  const double l=1;
  double y[n], r[n], a1_y[n], a1_r[n];
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++)
      y[j]=a1_y[j]=r[j]=a1_r[j]=0;
    a1_r[i]=1;
    a1_f(s,y,a1_y,l,r,a1_r);
    for (int j=0;j<n;j++) cout << a1_y[j] << " ";
    cout << endl;
  }
  return 0;
}
```

Again, sparsity is not taken into account. The numerical results are equal to those computed by the tangent-linear code.

3. *Use* `dco` *to implement the tangent-linear and adjoint models.*

A tangent-linear version of the given implementation of the SFI problem that uses `dco` is the following:

```
#include "dco_t1s_type.hpp"

void f(int s, dco_t1s_type* y, double l, dco_t1s_type* r) {
  dco_t1s_type left,right,up,down,dyy,drr;
  ...
}

int main() {
  const int s=5;
  const int n=(s-1)*(s-1);
  const double l=1;
  dco_t1s_type y[n], r[n];
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) y[j]=r[j]=0;
    y[i].t=1;
    f(s,y,l,r);
    for (int j=0;j<n;j++) cout << r[j].t << " ";
    cout << endl;
  }
  return 0;
}
```

The types of all active variables in f are switched to dco_t1s_type. The main routine seeds the tangent-linear components y[i].t of all inputs with the Cartesian basis vectors in $\mathbb{R}^n$. The Jacobian is harvested from the tangent-linear components of the outputs r[j].t. Numerical correctness is verified by comparison with the previously computed values. Sparsity is not exploited.

Similarly, an adjoint version that uses dco is the following:

```
#include "dco_a1s_type.hpp"

extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];

void f(int s, dco_a1s_type * y, double l,
                dco_a1s_type * r) {
  dco_a1s_type left,right,up,down,dyy,drr;
  ...
}

int main() {
  const int s=5;
  const int n=(s-1)*(s-1);
  const double l=1;
  dco_a1s_type y[n], r[n];
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) y[j]=r[j]=0;
    f(s,y,l,r);
    dco_a1s_tape[r[i].va].a=1;
    dco_a1s_interpret_tape();
    for (int j=0;j<n;j++)
      cout << dco_a1s_tape[y[j].va].a << " ";
    cout << endl;
    dco_a1s_reset_tape();
  }
  return 0;
}
```

A tape is generated by the overloaded version of f. Seeding of the adjoint components of all tape entries that correspond to outputs of f with the Cartesian basis vectors in $\mathbb{R}^n$ is followed by an interpretation of the tape. The rows of the Jacobian are extracted from the adjoint components of all tape entries that correspond to inputs of f. The tape is reset to its initial state after the computation of a row of the Jacobian.

4. *Use the Newton algorithm as well as a corresponding matrix-free implementation based on Conjugate Gradients to solve the SFI problem. Compare the run times.*

The various implementations of the tangent-linear model replace the respective code for the evaluation / approximation of the Jacobian and of its product with a vector in the solution of Exercise 1.4.2 discussed in Section C.1.2.

C.2.3 Exercise 2.4.3

Consider the same implementation of the extended Rosenbrock function $y = f(\mathbf{x})$ *as in Section* 1.4.3.

1. *Implement the tangent-linear model* $y^{(1)} = \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$ *by writing a tangent-linear code by hand and use it to accumulate* $\nabla f(\mathbf{x})$ *with machine accuracy. Compare the numerical results with those obtained by the finite difference approximation in Section* 1.4.3.

   An implementation of the tangent-linear extended Rosenbrock function is the following:

```
void t1_f(int n, double *x, double *t1_x,
                 double &y, double& t1_y) {
  double t, t1_t;
  t1_y=0;
  y=0;
  for (int i=0;i<n-1;i++) {
    t1_t=t1_x[i+1]-2*x[i]*t1_x[i];
    t=x[i+1]-x[i]*x[i];
    t1_y=t1_y-2*t1_x[i]+2*x[i]*t1_x[i]+10*2*t*t1_t;
    y=y+1-2*x[i]+x[i]*x[i]+10*t*t;
  }
}
```

   Verification of the correctness at any given input by finite differences is straightforward.

2. *Implement the adjoint model* $\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \nabla f(\mathbf{x})^T \cdot y_{(1)}$ *by writing an adjoint code by hand and use it to accumulate* $\nabla f(\mathbf{x})$ *with machine accuracy. Compare the numerical results with those obtained by the tangent-linear approach.*

   An implementation of the adjoint extended Rosenbrock function is the following:

```
stack<double> fds;

void a1_f(int n, double *x, double *a1_x,
                 double &y, double &a1_y) {
  double t,a1_t=0;
  // augmented forward section
  y=0;
  for (int i=0;i<n-1;i++) {
    fds.push(t);
    t=x[i+1]-x[i]*x[i];
    y=y+1-2*x[i]+x[i]*x[i]+10*t*t;
  }
  // reverse section
  for (int i=n-2;i>=0;i--) {
    a1_x[i]-=2*a1_y;
    a1_x[i]+=2*x[i]*a1_y;
```

```
      a1_t+=10*2*t*a1_y;
      t=fds.top(); fds.pop();
      a1_x[i+1]+=a1_t;
      a1_x[i]-=2*x[i]*a1_t;
      a1_t=0;
    }
    a1_y=0;
}
```

The returned gradient is equal to the gradient computed by seeding the tangent-linear code with the identity in $\mathbb{R}^n$.

3. *Use* `dco` *to implement the tangent-linear and adjoint models.*

Again, the types of all active floating-point variables in f are switched to dco_t1s_type to obtain a tangent-linear code based on `dco`.

```
#include "dco_t1s_type.hpp"

void f(int n, dco_t1s_type *x, dco_t1s_type &y) {
  dco_t1s_type t;
  ...
}

int main() {
  const int n=3;
  dco_t1s_type x[n], y;
  for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) x[j]=2+cos(j);
    x[i].t=1;
    f(n,x,y);
    cout << y.t << endl;
  }
  return 0;
}
```

The main routine seeds the tangent-linear components x[i].t of all inputs with the Cartesian basis vectors in $\mathbb{R}^n$. The entries of the gradient are extracted from the tangent-linear component of the output y.t.

Similarly, an adjoint version that use `dco` is the following:

```
#include "dco_a1s_type.hpp"

extern dco_a1s_tape_entry dco_a1s_tape[DCO_A1S_TAPE_SIZE];

void f(int n, dco_a1s_type *x, dco_a1s_type &y) {
  dco_a1s_type t;
  ...
}
```

```
int main() {
  const int n=3;
  dco_a1s_type x[n], y;
  for (int j=0;j<n;j++) x[j]=2+cos(j);
  f(n,x,y);
  dco_a1s_tape[y.va].a=1;
  dco_a1s_interpret_tape();
  for (int j=0;j<n;j++)
    cout << dco_a1s_tape[x[j].va].a << endl;
  return 0;
}
```

A tape is generated by the overloaded version of f. Seeding of the adjoint component of the tape entry that corresponds to the output y of f with one is followed by the interpretation of the tape. The gradient is extracted from the adjoint components of all tape entries that correspond to inputs of f.

4. *Use the Steepest Descent algorithm with both first derivative models to minimize the extended Rosenbrock function. Compare the run times.*

   The various implementations of the tangent-linear and adjoint models replace the respective code for the evaluation / approximation of the gradient in the solution of Exercise 1.4.3 discussed in Section C.1.3.

C.2.4 Exercise 2.4.4

We use the NAG Library as a case study for a wide range of numerical libraries. Examples from the online documentation of the library are used for easier cross-reference. Tangent-linear and adjoint code is generated by `dcc`.

1. *Use the tangent-linear model with your favorite solver for systems of nonlinear equations to find a numerical solution of the SFI problem; repeat for further MINPACK-2 test problems.*

   To demonstrate the use of the nonlinear equations solver from the NAG Library (c05ubc), an example is used that computes the values $x_0,\ldots,x_8$ of the tridiagonal system $F(\mathbf{x}) = 0$, where the residual $F : \mathbb{R}^9 \to \mathbb{R}^9$, $\mathbf{y} = F(\mathbf{x})$, is implemented as

$$\begin{aligned} y_0 &= (3-2\cdot x_0)\cdot x_0 - 2\cdot x_1 + 1,\\ y_i &= -x_{i-1} + (3-2\cdot x_i)\cdot x_i - 2\cdot x_{i+1} + 1,\\ y_8 &= -x_7 + (3-2\cdot x_8)\cdot x_8 + 1 \end{aligned}$$

   for $i = 1,\ldots,7$. A corresponding C++ implementation that is accepted by `dcc` is the following:

```
void f(int n, double *x, double *y) {
  int k=0;
  int nm1=0;
  int km1=0;
```

```
  int kp1=0;
  nm1=n-1;
  while (k<n) {
    y[k]=(3.0-x[k]*2.0)*x[k]+1.0;
    if (k>0) { km1=k-1; y[k]=y[k]-x[km1]; }
    if (k<nm1) { kp1=k+1; y[k]=y[k]-x[kp1]*2.0; }
    k=k+1;
  }
}
```

Running `dcc` in tangent-linear mode results in a routine

```
void t1_f(int n, double* x, double* t1_x,
                 double* y, double* t1_y);
```

that can be used to accumulate the Jacobian matrix $\nabla F(\mathbf{x})$ alongside with the value of the residual as part of the following driver that needs to be passed to the NAG library routine nag_zero_nonlin_eqns_deriv_1.

```
static void NAG_CALL f(Integer n, double x[], double fvec[],
                       double fjac[], Integer tdfjac,
                       Integer *userflag, Nag_User *comm) {
#define FJAC(I, J) fjac[((I))*tdfjac+(J)]
  Integer j, k;
  double* t1_x=new double[n];
  double* t1_fvec=new double[n];

  if (*userflag!=2)
      f(n,x,fvec);
  else {
    memset(t1_x,0,n*sizeof(double));
    for (int i=0;i<n;i++) {
      t1_x[i]=1;
      t1_f(n,x,t1_x,fvec,t1_fvec);
      t1_x[i]=0;
      for (int j=0;j<n;j++) FJAC(j,i)=t1_fvec[j];
    }
  }
  delete [] t1_fvec, t1_x;
}
```

The parameter *userflag is used to select between pure function evaluation and the computation of the Jacobian. Simply replacing the hand-written version of f that is originally provided by NAG followed by building the example as outlined in the documentation yields the desired output.

```
nag_zero_nonlin_eqns_deriv_1 (c05ubc)
Example Program Results

Final approximate solution
```

−0.5707	−0.6816	−0.7017
−0.7042	−0.7014	−0.6919
−0.6658	−0.5960	−0.4164

The application of the same steps to the SFI problem as well as to other MINPACK-2 test problems is straightforward.

2. *Use the adjoint model with your favorite solver for nonlinear programming to minimize the extended Rosenbrock function; repeat for the other two test problems from Section* 1.4.3.

The function $F : \mathbb{R}^2 \to \mathbb{R}$

$$y = F(\mathbf{x}) = e^{x_0} \cdot (4 \cdot x_0^2 + 2 \cdot x_1^2 + 4 \cdot x_0 \cdot x_1 + 2 \cdot x_1 + 1)$$

implemented as

```
void f(int n, double* x, double& y) {
  y=exp(x[0])*(4*x[0]*x[0]+2*x[1]*x[1]
                          +4*x[0]*x[1]+2*x[1]+1);
}
```

is minimized starting from $(-1, 1)$ and using the adjoint routine

```
void a1_f(int a1_mode, int n,
          double* x, double* a1_x,
          double& y, double& a1_y)
```

generated by `dcc`. The driver

```
static void NAG_CALL objfun(Integer n, double x[],
                            double *objf, double g[],
                            Nag_Comm *comm) {
  double a1_y=1;
  memset(g,0,n*sizeof(double));
  a1_f(1,n,x,g,*objf,a1_y);
}
```

to be passed to the NAG library routine expects a1_f to return the correct function value in addition to the gradient at point x. The former needs to be stored as a result checkpoint after the augmented forward section in order to be restored after the reverse section of the adjoint code. Hence, the files declare_checkpoints.inc, f_store_results.inc, and f_restore_results.inc need to contain the corresponding declarations, store and restore code, respectively. For example,

- declare_checkpoints.inc: **double** rescp=0;
- f_store_results.inc: rescp=y;
- f_restore_results.inc: y=rescp;

Table C.1. *Run time of minimization of the extended Rosenbrock function using* `e04dgc` *of the NAG C Library.*

n	TLM	ADM
100	0.2	0.01
500	18	0.1
1000	220	0.5

Consequently, the minimizer $(0.5,-1)$ is found yielding an optimum of zero.

It is straightforward to adapt the example for the extended Rosenbrock function. Starting from $x_i = 2$ for $i = 0,\ldots,n-1$, the minimizer $x_i^* = 1$, $i = 0,\ldots,n-1$ is computed. The computational effort is dominated by the run time of the gradient accumulation. Table C.1 summarizes the results. The adjoint outperforms the tangent-linear code with increasing n as expected.

C.2.5 Exercise 2.4.5

1. *Consider the following modification of the example code from Section* 2.4.1*:*

```
void h(double& x) {
  x*=x;
}

void g(int n, double* x, double& y) {
  y=0;
  for (int i=0;i<n;i++) {
    h(x[i]); y*=x[i];
  }
}

void f(int n, double* x, double &y) {
  for (int i=0;i<n;i++) x[i]=sqrt(x[i]/x[(i+1)%n]);
  g(n,x,y);
  y=cos(y);
}
```

Write adjoint code that correspond to the four call tree reversal schemes

- $R_1 = \{(f,g,0),(g,h,0)\}$
- $R_2 = \{(f,g,1),(g,h,0)\}$
- $R_3 = \{(f,g,0),(g,h,1)\}$
- $R_4 = \{(f,g,1),(g,h,1)\}$,

respectively. Apply the reversal mode of (g,h) *to all* n *calls of* h *inside of* g*.*

Globally split call tree reversal ($R_1 = \{(\mathrm{f},\mathrm{g},0),(\mathrm{g},\mathrm{h},0)\}$) yields the following adjoint code:

```
stack<double> fds;

void a1_h(int a1_mode, double& x, double& a1_x) {
  if (a1_mode==1) { // split augmented forward ...
    fds.push(x);
    x*=x;
  }
  else { // ... and reverse sections
    x=fds.top(); fds.pop();
    a1_x*=2*x;

  }
}

void a1_g(int a1_mode, int n, double* x, double* a1_x,
          double& y, double& a1_y) {
  if (a1_mode==1) { // split augmented forward
    y=1.0;
    for (int i=0;i<n;i++) {
      a1_h(1,x[i],a1_x[i]);
      fds.push(y);
      y*=x[i];
    }
  }
  else { // ... and reverse sections
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=y*a1_y;
      a1_y=x[i]*a1_y;
      a1_h(2,x[i],a1_x[i]);
    }
    a1_y=0;
  }
}

void a1_f(int n, double* x, double* a1_x,
                 double &y, double& a1_y) {
  // joint augmented forward ...
  for (int i=0;i<n;i++) {
    fds.push(x[i]);
    x[i]=sqrt(x[i]/x[(i+1)%n]);
  }
  a1_g(1,n,x,a1_x,y,a1_y);
  fds.push(y);
  y=cos(y);
  double res_cp=y; // store result
  // ... and reverse sections
  y=fds.top(); fds.pop();
  a1_y=-sin(y)*a1_y;
  a1_g(2,n,x,a1_x,y,a1_y);
```

```
  for (int i=n-1;i>=0;i--) {
    x[i]=fds.top(); fds.pop();
    double v=x[i]/x[(i+1)%n];
    double a1_v=a1_x[i]/(2*sqrt(v)); a1_x[i]=0;
    a1_x[i]+=a1_v/x[(i+1)%n];
    a1_x[(i+1)%n]-=a1_v*x[i]/(x[(i+1)%n]*x[(i+1)%n]);
  }
  y=res_cp; // restore result
}
```

Result checkpointing ensures the return of the correct function value.

2. Joint-over-split call tree reversal ($R_2 = \{(f,g,1),(g,h,0)\}$) yields the following adjoint code:

```
stack<double> fds;
stack<double> arg_cp_g;

void a1_h(int a1_mode, double& x, double& a1_x) {
  if (a1_mode==1) { // split augmented forward ...
    fds.push(x);
    x*=x;
  }
  else { // ... and reverse sections
    x=fds.top(); fds.pop();
    a1_x*=2*x;
  }
}

void a1_g(int a1_mode, int n, double* x, double* a1_x,
          double& y, double& a1_y) {
  if (a1_mode==1) { // joint augmented forward ...
    y=1.0;
    for (int i=0;i<n;i++) {
      a1_h(1,x[i],a1_x[i]);
      fds.push(y);
      y*=x[i];
    }
    // ... and reverse sections
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=y*a1_y;
      a1_y=x[i]*a1_y;
      a1_h(2,x[i],a1_x[i]);
    }
    a1_y=0;
  }
  else if (a1_mode==3) { // store inputs
    for (int i=0;i<n;i++) arg_cp_g.push(x[i]);
  }
  else if (a1_mode==4) { // restore inputs
```

```
    for (int i=n-1;i>=0;i--) {
      x[i]=arg_cp_g.top(); arg_cp_g.pop();
    }
  }
}

void a1_f(int n, double* x, double* a1_x,
                 double &y, double& a1_y) {
  // joint augmented forward ...
  for (int i=0;i<n;i++) {
    fds.push(x[i]);
    x[i]=sqrt(x[i]/x[(i+1)%n]);
  }
  a1_g(3,n,x,a1_x,y,a1_y);
  g(n,x,y);
  fds.push(y);
  y=cos(y);
  double res_cp=y; // store result
  // ... and reverse sections
  y=fds.top(); fds.pop();
  a1_y=-sin(y)*a1_y;
  a1_g(4,n,x,a1_x,y,a1_y);
  a1_g(1,n,x,a1_x,y,a1_y);
  for (int i=n-1;i>=0;i--) {
    x[i]=fds.top(); fds.pop();
    double v=x[i]/x[(i+1)%n];
    double a1_v=a1_x[i]/(2*sqrt(v)); a1_x[i]=0;
    a1_x[i]+=a1_v/x[(i+1)%n];
    a1_x[(i+1)%n]-=a1_v*x[i]/(x[(i+1)%n]*x[(i+1)%n]);
  }
  y=res_cp; // restore result
}
```

3. Split-over-joint call tree reversal ($R_3 = \{(f,g,0),(g,h,1)\}$) yields the following adjoint code.

```
stack<double> fds;
stack<double> arg_cp_h;

void a1_h(int a1_mode, double& x, double& a1_x) {
  if (a1_mode==1) { // joint augmented forward ...
    fds.push(x);
    x*=x;
    // ... and reverse sections
    x=fds.top(); fds.pop();
    a1_x*=2*x;
  }
  else if (a1_mode==3) { // store inputs
    arg_cp_h.push(x);
  }
```

```
  else if (a1_mode==4) { // restore inputs
    x=arg_cp_h.top(); arg_cp_h.pop();
  }
}

void a1_g(int a1_mode, int n, double* x, double* a1_x,
          double& y, double& a1_y) {
  if (a1_mode==1) { // split augmented forward ...
    y=1.0;
    for (int i=0;i<n;i++) {
      a1_h(3,x[i],a1_x[i]);
      h(x[i]);
      fds.push(y);
      y*=x[i];
    }
  }
  else { // ... and reverse sections
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=y*a1_y;
      a1_y=x[i]*a1_y;
      a1_h(4,x[i],a1_x[i]);
      a1_h(1,x[i],a1_x[i]);
    }
    a1_y=0;
  }
}
```

The implementation of a1_f is the same is in R_1.

4. Globally joint call tree reversal ($R_4 = \{(f,g,1),(g,h,1)\}$) yields the following adjoint code:

```
stack<double> fds;
stack<double> arg_cp_h;
stack<double> arg_cp_g;

void a1_h(int a1_mode, double& x, double& a1_x) {
  if (a1_mode==1) { // joint augmented forward ...
    fds.push(x);
    x*=x;
    // ... and reverse sections
    x=fds.top(); fds.pop();
    a1_x*=2*x;
  }
  else if (a1_mode==3) { // store inputs
    arg_cp_h.push(x);
  }
```

```
  else if (a1_mode==4) { // restore inputs
    x=arg_cp_h.top(); arg_cp_h.pop();
  }
}

void a1_g(int a1_mode, int n, double* x, double* a1_x,
          double& y, double& a1_y) {
  if (a1_mode==1) { // joint augmented forward ...
    y=1.0;
    for (int i=0;i<n;i++) {
      a1_h(3,x[i],a1_x[i]);
      h(x[i]);
      fds.push(y);
      y*=x[i];
    }
    // ... and reverse sections
    for (int i=n-1;i>=0;i--) {
      y=fds.top(); fds.pop();
      a1_x[i]+=y*a1_y;
      a1_y=x[i]*a1_y;
      a1_h(4,x[i],a1_x[i]);
      a1_h(1,x[i],a1_x[i]);
    }
    a1_y=0;
  }
  else if (a1_mode==3) { // store inputs
    for (int i=0;i<n;i++) arg_cp_g.push(x[i]);
  }
  else if (a1_mode==4) { // restore inputs
    for (int i=n-1;i>=0;i--) {
      x[i]=arg_cp_g.top(); arg_cp_g.pop();
    }
  }
}
```

The implementation of a1_f is the same is in R_2.

For $n = 3$, the gradient $\nabla f \in \mathbb{R}^3$ at $\mathbf{x} = (3, 2.5403, 1.58385)^T$ becomes $\nabla f(\mathbf{x}) = (-0.171085, -0.202045, 1.11022e-16)^T$.

5. *Consider the annotated call tree in Figure* C.2.

   (a) *Derive all call tree reversal schemes. Compute their respective operation counts and memory requirements.*

   (b) *Compare the results of the greedy* Smallest- *and* Largest-Recording-First *heuristics for an available memory of size* 140*,* 150*, and* 160*.*

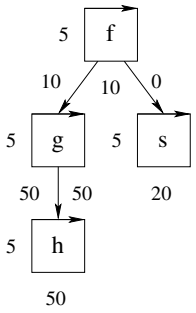


Figure C.2. *Annotated call tree for Exercise* 2.4.5.

- $R_0 = ((\mathrm{f},\mathrm{g},1),(\mathrm{f},\mathrm{s},1),(\mathrm{g},\mathrm{h},1))$: $\mathrm{MEM}(R_0) = 115$, $\mathrm{OPS}(R_0) = 410$

```
|_ a1_f(RECORD)
|       |_ a1_g(STORE_INPUTS)
|       |_ g
|       |   |_ h
|       |_ a1_s(STORE_INPUTS)
|       |_ s
|_ a1_f(ADJOIN)
        |_ a1_s(RESTORE_INPUTS)
        |_ a1_s(RECORD)
        |_ a1_s(ADJOIN)
        |_ a1_g(RESTORE_INPUTS)
        |_ a1_g(RECORD)
        |       |_ a1_h(STORE_INPUTS)
        |       |_ h
        |_ a1_g(ADJOIN)
                |_ a1_h(RESTORE_INPUTS)
                |_ a1_h(RECORD)
                |_ a1_h(ADJOIN)
```

- $R_1 = ((\mathrm{f},\mathrm{g},1),(\mathrm{f},\mathrm{s},1),(\mathrm{g},\mathrm{h},0))$: $\mathrm{MEM}(R_1) = 160$, $\mathrm{OPS}(R_1) = 360$

```
|_ a1_f(RECORD)
|       |_ a1_g(STORE_INPUTS)
|       |_ g
|       |   |_ h
|       |_ a1_s(STORE_INPUTS)
|       |_ s
|_ a1_f(ADJOIN)
        |_ a1_s(RESTORE_INPUTS)
        |_ a1_s(RECORD)
        |_ a1_s(ADJOIN)
        |_ a1_g(RESTORE_INPUTS)
        |_ a1_g(RECORD)
        |       |_ a1_h(RECORD)
        |_ a1_g(ADJOIN)
                |_ a1_h(ADJOIN)
```

- $R_2 = ((f,g,1),(f,s,0),(g,h,1))$: MEM(R_2) = 115, OPS(R_2) = 390;

```
|_ a1_f(RECORD)
|       |_ a1_g(STORE_INPUTS)
|       |_ g
|       |  |_ h
|       |_ a1_s(RECORD)
|_ a1_f(ADJOIN)
        |_ a1_s(ADJOIN)
        |_ a1_g(RESTORE_INPUTS)
        |_ a1_g(RECORD)
        |       |_ a1_h(STORE_INPUTS)
        |       |_ h
        |_ a1_g(ADJOIN)
                |_ a1_h(RESTORE_INPUTS)
                |_ a1_h(RECORD)
                |_ a1_h(ADJOIN)
```

- $R_3 = ((f,g,1),(f,s,0),(g,h,0))$: MEM(R_3) = 160, OPS(R_3) = 340;

```
|_ a1_f(RECORD)
|       |_ a1_g(STORE_INPUTS)
|       |_ g
|       |  |_ h
|       |_ a1_s(RECORD)
|_ a1_f(ADJOIN)
        |_ a1_s(ADJOIN)
        |_ a1_g(RESTORE_INPUTS)
        |_ a1_g(RECORD)
        |       |_ a1_h(RECORD)
        |_ a1_g(ADJOIN)
                |_ a1_h(ADJOIN)
```

- $R_4 = ((f,g,0),(f,s,1),(g,h,1))$: MEM(R_4) = 145, OPS(R_4) = 260;

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|       |       |_ a1_h(STORE_INPUTS)
|       |       |_ h
|       |_ a1_s(STORE_INPUTS)
|       |_ s
|_ a1_f(ADJOIN)
        |_ a1_s(RESTORE_INPUTS)
        |_ a1_s(RECORD)
        |_ a1_s(ADJOIN)
        |_ a1_g(ADJOIN)
                |_ a1_h(RESTORE_INPUTS)
                |_ a1_h(RECORD)
                |_ a1_h(ADJOIN)
```

- $R_5 = ((f,g,0),(f,s,1),(g,h,0)) : \mathrm{MEM}(R_5) = 180, \mathrm{OPS}(R_5) = 210;$

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|       |       |_ a1_h(RECORD)
|       |_ a1_s(STORE_INPUTS)
|       |_ s
|_ a1_f(ADJOIN)
        |_ a1_s(RESTORE_INPUTS)
        |_ a1_s(RECORD)
        |_ a1_s(ADJOIN)
        |_ a1_g(ADJOIN)
                |_ a1_h(ADJOIN)
```

- $R_6 = ((f,g,0),(f,s,0),(g,h,1)) : \mathrm{MEM}(R_6) = 145, \mathrm{OPS}(R_6) = 240;$

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|       |       |_ a1_h(STORE_INPUTS)
|       |       |_ h
|       |_ a1_s(RECORD)
|_ a1_f(ADJOIN)
        |_ a1_s(ADJOIN)
        |_ a1_g(ADJOIN)
                |_ a1_h(RESTORE_INPUTS)
                |_ a1_h(RECORD)
                |_ a1_h(ADJOIN)
```

- $R_7 = ((f,g,0),(f,s,0),(g,h,0)) : \mathrm{MEM}(R_7) = 190, \mathrm{OPS}(R_7) = 190;$

```
|_ a1_f(RECORD)
|       |_ a1_g(RECORD)
|       |       |_ a1_h(RECORD)
|       |_ a1_s(RECORD)
|_ a1_f(ADJOIN)
        |_ a1_s(ADJOIN)
        |_ a1_g(ADJOIN)
                |_ a1_h(ADJOIN)
```

For an available memory of size 140, both the greedy Smallest- and Largest-Recording-First heuristics yield the optimal reversal scheme R_2 performing $\mathrm{OPS}(R_2) = 390$ operations. If a memory of size 150 is at our disposal, then the Largest-Recording-First heuristic selects the optimal reversal scheme R_6 ($\mathrm{OPS}(R_6) = 240$), whereas the Smallest-Recording-First heuristic fails to improve R_2. The Largest-Recording-First heuristic also outperforms its competitor for an available memory of size 160 by selecting reversal scheme R_6 as opposed to R_3 ($\mathrm{OPS}(R_3) = 340$).

C.3 Chapter 3

C.3.1 Exercise 3.5.1

Consider the code in Listing C.9.

1. *Write second-order tangent-linear code based on the tangent-linear code that was developed in Section* 2.4.1*; use it to accumulate the Hessian of the dependent output y with respect to the independent input x.*

   An implementation of the second-order tangent-linear model can be obtained by applying the Tangent-Linear Code Generation Rules from Section 2.1.1 to the tangent-linear code developed in Section C.2.1. Listings of second derivative code become rather lengthy; hence they are omitted.

   A smart way of solving this exercise is by applying `dcc` to the following variant of Listing C.9:

   Listing C.10. *Variant of listing* C.9 *that is accepted by* `dcc`.

```
void g(int n, double* x, double& y) {
  int i=0;
  y=1.0;
  while (i<n) {
    y=y*x[i]*x[i];
    i=i+1;
  }
}

void f(int n, double* x, double &y) {
  int i=0;
  int ip1=0;
  int nm1=0;
  while (i<n) {
    nm1=n-1;
    if (i<nm1) {
      ip1=i+1;
      x[i]=sqrt(x[i]/x[ip1]);
    } else {
      x[i]=sqrt(x[i]/x[0]);
    }
    i=i+1;
  }
  g(n,x,y);
  y=cos(y);
}
```

   A second-order tangent-linear code is generated by applying `dcc` twice in order to generate `t1_f.c` from a file `f.c` that contains the sources of both `g` and `f` followed by the generation of `t2_t1_f.c` from `t1_f.c`. See Chapter 5 for details.

2. *Write second-order adjoint code based on the adjoint code that was developed in Section* 2.4.1 *(forward-over-reverse mode in both split and joint modes); use it to accumulate the same Hessian as in* 1.

   Implementations of the second-order adjoint model can be obtained by applying the Tangent-Linear Code Generation Rules from Section 2.1.1 to the adjoint code developed in Section C.2.1. Alternatively, or in order to verify the solutions, the adjoint code can be reimplemented in a syntax that is accepted by `dcc`. Application of `dcc` in tangent-linear mode yields the desired second-order adjoint code.

3. *Write second-order adjoint code based on the tangent-linear code that was developed in Section* 2.4.1 *(reverse-over-forward mode in both split and joint modes); use it to accumulate the same Hessian as in* 1.

   The Adjoint Code Generation Rules from Section 2.2.1 need to be applied to the code developed in Section C.2.1. Alternatively, the tangent-linear code can be reimplemented in a syntax that is accepted by `dcc`. Application of `dcc` in adjoint mode yields the desired second-order adjoint code in joint call tree reversal mode. Special care must be taken when defining the argument checkpoint of `g`. It should contain both `x` and its tangent-linear counterpart `t1_x`. Split call tree reversal can be derived by simple local modifications. Again, listings are omitted due to their excessive length. The correctness of a given solution can always be verified for a given input by comparing the numerical results with those obtained by second derivative code that was generated by `dcc`.

C.3.2 Exercise 3.5.2

Consider the given implementation of the extended Rosenbrock function f from Section 1.4.3.

1. *Write a second-order tangent-linear code and use it to accumulate $\nabla^2 f$ with machine accuracy. Compare the numerical results with those obtained by finite difference approximation.*

   Application of the Tangent-Linear Code Generation Rules from Section 2.1.1 to an implementation of the tangent-linear extended Rosenbrock function (see `t1_f` in Section C.2.3) yields the second-order tangent-linear code. Alternatively, `dcc` can be applied to the following variant of an implementation of the extended Rosenbrock function:

```
void f(int n, double *x, double &y) {
  int i=0;
  int nm1=0;
  int ip1=0;
  double t1=0;
  double t2=0;
  y=0;
  nm1=n-1;
  while (i<nm1) {
    t1=1-x[i];
```

```
      ip1=i+1;
      t2=(x[ip1]-x[i]*x[i]);
      y=y+t1*t1+10*t2*t2;
      i=i+1;
    }
}
```

For a given second-order tangent-linear code, the accumulation of the Hessian is straightforward; see Section 3.2.1.

2. *Write a second-order adjoint code in forward-over-reverse mode and use it to accumulate $\nabla^2 f$ with machine accuracy. Compare the numerical results with those obtained with the second-order tangent-linear approach.*

   Application of the Tangent-Linear Code Generation Rules from Section 2.1.1 to an implementation of the adjoint extended Rosenbrock function (see a1_f in Section C.2.3) yields the desired second-order adjoint code. Again, `dcc` can help to make this process more efficient and less error-prone.

3. *Use* `dco` *to accumulate $\nabla^2 f$ in second-order tangent-linear and adjoint modes with machine accuracy. Compare the numerical results with those obtained from the hand-written derivative code.*

   Refer to Sections 3.2.2 and 3.3.2 for instructions. The examples can be transferred identically to the extended Rosenbrock function. Simple type changes yield

```
void f(int n, dco_t2s_t1s_type *x, dco_t2s_t1s_type &y) {
  dco_t2s_t1s_type t;
  ...
}
```

   in second-order tangent-linear and

```
void f(int n, dco_t2s_a1s_type *x, dco_t2s_a1s_type &y) {
  dco_t2s_a1s_type t;
  ...
}
```

   in second-order adjoint modes. The driver routines, as well as the build process, is the same as in the examples in Sections 3.2.2 and 3.3.2.

4. *Use the Newton algorithm and a corresponding matrix-free implementation based on Conjugate Gradients for the solution of the Newton system to minimize the extended Rosenbrock function for different start values of your own choice. Compare the run times for the various approaches to computing the required derivatives as well as the run times of the optimization algorithms for increasing values of n.*

   The various implementations of the second-order adjoint model replace the respective code for the evaluation / approximation of the gradient and the Hessian in the solution of Exercise 1.4.3 discussed in Section C.1.3. A matrix-free implementation of the

Newton-CG algorithm based on the first-order version discussed in Section C.1.2 is shown in Listing C.11.

Listing C.11. *Newton-CG algorithm.*

```
void cg(double eps, double* x, double& y, double* v,
        double* g) {
  double r[n], p[n], Hv[n];
  for (int i=0;i<n;i++) g[i]=Hv[i]=0;
  double t2_y=0, a1_y=1, t2_a1_y=0;
  t2_a1_f(1, n, x, v, g, Hv, y, t2_y, a1_y, t2_a1_y);
  for (int i=0;i<n;i++) p[i]=r[i]=-g[i]-Hv[i];
  double normr=norm(r);
  while (normr>eps) {
    for (int i=0;i<n;i++) g[i]=Hv[i]=0;
    t2_y=0, a1_y=1, t2_a1_y=0;
    t2_a1_f(1, n, x, p, g, Hv, y, t2_y, a1_y, t2_a1_y);
    double rTr=xTy(r,r);
    double alpha=rTr/xTy(p,Hv);
    axpy(alpha,p,v,v);
    axpy(-alpha,Hv,r,r);
    double beta=xTy(r,r)/rTr;
    axpy(beta,p,r,p);
    normr=norm(r);
  }
}

int main(int argc, char* argv[]) {
  const double eps=1e-12;
  double x[n], v[n], g[n], y;

  for (int i=0;i<n;i++) { x[i]=1; v[i]=0; }
  do {
    cg(eps,x,y,v,g);
    for (int i=0;i<n;i++) x[i]+=v[i];
  } while (norm(g)>eps);
  cout << "Solution: " << y << endl;
  return 0;
}
```

It uses the second-order adjoint version t2_a1_f of the extended Rosenbrock function to compute the required objective values, gradients, and Hessian vector products. Qualitatively, the observed run times are similar to Table 1.3.

C.3.3 Exercise 3.5.3

1. *Write third-order tangent-linear and adjoint versions for the code in Section* 3.5.1. *Run numerical tests to verify correctness.*

The application of the Tangent-Linear Code Generation Rules to the previously developed second-order tangent-linear and adjoint code is straightforward. Listings become rather lengthy and are hence omitted. Finite differences can be applied to the second derivative code to qualitatively verify the numerical correctness of third derivative code at selected points.

2. *Given* $\mathbf{y} = F(\mathbf{x})$*, derive the following higher derivative code and provide drivers for its use in the accumulation of the corresponding derivative tensors:*

   (a) *third-order adjoint code in reverse-over-reverse-over-reverse mode;*

   (b) *fourth-order adjoint code in forward-over-forward-over-forward-over-reverse mode;*

   (c) *fourth-order adjoint code in reverse-over-forward-over-reverse-over-forward mode.*

   Discuss the complexity of computing various projections of the third and fourth derivative tensors.

   - Third-order adjoint code in reverse-over-reverse-over-reverse mode:
     Application of reverse mode AD to an implementation of $\mathbf{y} = F(\mathbf{x})$ yields the first-order adjoint code

$$\begin{aligned}\mathbf{y} &= F(\mathbf{x})\\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x})\rangle\\ \mathbf{y}_{(1)} &= 0.\end{aligned}$$

     Application of reverse mode AD with required floating-point data stack s to the first-order adjoint code yields the second-order adjoint code with augmented forward section

$$\begin{aligned}\mathbf{y} &= F(\mathbf{x})\\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x})\rangle\\ s[0] &= \mathbf{y}_{(1)}\\ \mathbf{y}_{(1)} &= 0\end{aligned}$$

     and reverse section

$$\begin{aligned}\mathbf{y}_{(1)} &= s[0]\\ \mathbf{y}_{(1,2)} &= 0\\ \mathbf{y}_{(1,2)} &= \mathbf{y}_{(1,2)} + \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x})\rangle\\ \mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x})\rangle\\ \mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x})\rangle\\ \mathbf{y}_{(2)} &= 0.\end{aligned}$$

     This second-order adjoint code computes

$$\begin{aligned}\mathbf{y} &= F(\mathbf{x})\\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x})\rangle\end{aligned}$$

$$\begin{aligned}\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(1,2)} &= \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(2)} &= 0\end{aligned}$$

and leaves $\mathbf{x}$, $\mathbf{y}_{(1)}$, and $\mathbf{x}_{(1,2)}$ unchanged. The value of $\mathbf{y}_{(1)}$ that is set equal to zero by the first-order adjoint code is recovered by the second-order adjoint code. It is pushed to s prior to the last assignment in the augmented forward section followed by restoring the original value at the beginning of the reverse section. Application of reverse mode AD to the second-order adjoint code yields the third-order adjoint code with augmented forward section

$$\mathbf{y} = F(\mathbf{x}) \tag{C.2}$$
$$\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \tag{C.3}$$
$$s[0] = \mathbf{y}_{(1)} \tag{C.4}$$
$$r[0] = \mathbf{y}_{(1)} \tag{C.5}$$
$$\mathbf{y}_{(1)} = 0 \tag{C.6}$$
$$\mathbf{y}_{(1)} = s[0] \tag{C.7}$$
$$\mathbf{y}_{(1,2)} = 0 \tag{C.8}$$
$$\mathbf{y}_{(1,2)} = \mathbf{y}_{(1,2)} + \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle \tag{C.9}$$
$$\mathbf{x}_{(2)} = \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.10}$$
$$\mathbf{x}_{(2)} = \mathbf{x}_{(2)} + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle \tag{C.11}$$
$$r[1] = \mathbf{y}_{(2)} \tag{C.12}$$
$$\mathbf{y}_{(2)} = 0 \tag{C.13}$$

and reverse section

$$\mathbf{y}_{(2)} = r[1] \tag{C.14}$$
$$\mathbf{y}_{(2,3)} = 0 \tag{C.15}$$
$$\mathbf{y}_{(2,3)} = \mathbf{y}_{(2,3)} + \langle \mathbf{x}_{(2,3)}, \nabla F(\mathbf{x}) \rangle \tag{C.16}$$
$$\mathbf{x}_{(3)} = \mathbf{x}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.17}$$
$$\mathbf{x}_{(1,2,3)} = \mathbf{x}_{(1,2,3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.18}$$
$$\mathbf{y}_{(1,3)} = \mathbf{y}_{(1,3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.19}$$
$$\mathbf{x}_{(3)} = \mathbf{x}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}) \rangle \tag{C.20}$$
$$\mathbf{x}_{(1,2,3)} = \mathbf{x}_{(1,2,3)} + \langle \mathbf{y}_{(1,2,3)}, \nabla F(\mathbf{x}) \rangle \tag{C.21}$$
$$\mathbf{x}_{(3)} = \mathbf{x}_{(3)} + \langle \mathbf{y}_{(1,2,3)}, \mathbf{x}_{(1,2)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.22}$$
$$\mathbf{y}_{(1,2,3)} = 0 \tag{C.23}$$
$$s[0]_{(3)} = s[0]_{(3)} + \mathbf{y}_{(1,3)} \tag{C.24}$$
$$\mathbf{y}_{(1,3)} = 0 \tag{C.25}$$
$$\mathbf{y}_{(1)} = r[0] \tag{C.26}$$
$$\mathbf{y}_{(1,3)} = 0 \tag{C.27}$$
$$\mathbf{y}_{(1,3)} = \mathbf{y}_{(1,3)} + s[0]_{(3)} \tag{C.28}$$

$$s[0]_{(3)} = 0 \tag{C.29}$$
$$\mathbf{y}_{(1,3)} = \mathbf{y}_{(1,3)} + \langle \mathbf{x}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \tag{C.30}$$
$$\mathbf{x}_{(3)} = \mathbf{x}_{(3)} + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \tag{C.31}$$
$$\mathbf{x}_{(3)} = \mathbf{x}_{(3)} + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \tag{C.32}$$
$$\mathbf{y}_{(3)} = 0. \tag{C.33}$$

This third-order adjoint code computes

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,2)} &= \langle \mathbf{x}_{(1,2)}, \nabla F(\mathbf{x}) \rangle (= \langle \nabla F(\mathbf{x}), \mathbf{x}_{(1,2)} \rangle) \\
\mathbf{x}_{(2)} &= \mathbf{x}_{(2)} + \langle \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(2)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(2,3)} &= \langle \mathbf{x}_{(2,3)}, \nabla F(\mathbf{x}) \rangle (= \langle \nabla F(\mathbf{x}), \mathbf{x}_{(2,3)} \rangle) \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}_{(1,2,3)}, \mathbf{x}_{(1,2)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(1,2,3)} &= \mathbf{x}_{(1,2,3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1,2,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,3)} &= \mathbf{y}_{(1,3)} + \langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \\
&(= \mathbf{y}_{(1,3)} + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}_{(1,3)} \rangle) \\
\mathbf{y}_{(1,2,3)} &= 0 \\
\mathbf{y}_{(3)} &= 0
\end{aligned}$$

and leaves $\mathbf{x}$, $\mathbf{y}_{(1)}$, $\mathbf{y}_{(2)}$, $\mathbf{x}_{(1,2)}$, $\mathbf{x}_{(2,3)}$, $\mathbf{x}_{(1,3)}$ unchanged. Equation (C.4) and (C.7) ensure that $\mathbf{y}_{(1)}$ has the same value in (C.3), (C.10), (C.18), and (C.20). Because of (C.5) and (C.26) this value is also valid in (C.31). Similarly, the value of $\mathbf{y}_{(2)}$ is the same in (C.11) and (C.17). The incrementation of $\mathbf{y}_{(1,2)}$ in (C.9) is made obsolete by (C.8). An analogous statement holds for $\mathbf{y}_{(2,3)}$ in (C.16) because of (C.15). The right-hand side value of $s[0]_{(3)}$ is initially zero implying that (C.24), (C.25), and (C.27) – (C.29) leave $\mathbf{y}_{(1,3)}$ unchanged. Hence, its value is incremented by (C.30).

The whole third derivative tensor is accumulated as

$$\begin{aligned}
\mathbf{x}_{(3)} = \mathbf{x}_{(3)} &+ \langle \mathbf{x}_{(2,3)}, \mathbf{y}_{(2)}, \nabla^2 F(\mathbf{x}) \rangle \\
&+ \langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}) \rangle \\
&+ \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&+ \langle \mathbf{y}_{(1,2,3)}, \mathbf{x}_{(1,2)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle
\end{aligned}$$

at a computational cost of $O(n^3) \cdot Cost(F)$ by setting $\mathbf{x}_{(3)} = \mathbf{x}_{(1,3)} = \mathbf{y}_{(2)} = \mathbf{y}_{(1,2,3)} = \mathbf{y}_{(3)} = 0$ on input and letting $\mathbf{x}_{(2,3)}$, $\mathbf{x}_{(1,2)}$, and $\mathbf{y}_{(1)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$, $\mathbb{R}^n$, and $\mathbb{R}^m$, respectively. Projections of $\nabla^3 F(\mathbf{x})$ can be obtained at a lower computational cost, for example,

- $\langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}$ at the cost of $O(1) \cdot Cost(F)$;
- $\langle \mathbf{x}_{(2,3)}, \mathbf{x}_{(1,2)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}^m$ at the cost of $O(m) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^m$);
- $\langle \mathbf{x}_{(1,2)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}^{m \times n}$ at the cost of $O(m \cdot n) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ and $\mathbf{x}_{(2,3)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively);
- $\langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}) \rangle \in \mathbb{R}^{n \times n}$ at the cost of $O(n^2) \cdot Cost(F)$ ($\mathbf{x}_{(2,3)}$ and $\mathbf{x}_{(1,2)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$).

Moreover, the third-order adjoint code returns arbitrary projections of the second and first derivative tensors in addition to the original function value. Potential sparsity should be exploited to reduce the cost of computing certain required projections.

- Fourth-order adjoint code in forward-over-forward-over-forward-over-reverse mode:

  Application of reverse mode AD to an implementation of $\mathbf{y} = F(\mathbf{x})$ yields the first-order adjoint code

$$\begin{aligned} \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(1)} &= 0. \end{aligned}$$

  Application of forward mode AD to the first-order adjoint code yields the second-order adjoint code

$$\begin{aligned} \mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)}^{(2)} &= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\ \mathbf{y}_{(1)}^{(2)} &= 0;\ \mathbf{y}_{(1)} = 0. \end{aligned}$$

  Application of forward mode AD to the second-order adjoint code yields the third-order adjoint code

$$\begin{aligned} \mathbf{y}^{(2,3)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle \\ \mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\ \mathbf{y}^{(3)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(3)} \rangle \\ \mathbf{y} &= F(\mathbf{x}) \\ \mathbf{x}_{(1)}^{(2,3)} &= \mathbf{x}_{(1)}^{(2,3)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}^{(2,3)}, \nabla F(\mathbf{x}) \rangle \\ &\quad + \langle \mathbf{y}1(2)_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle \\ &\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle \end{aligned}$$

$$
\begin{aligned}
\mathbf{x}_{(1)}^{(2)} &= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)}^{(3)} &= \mathbf{x}_{(1)}^{(3)} + \langle \mathbf{y}_{(1)}^{(3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1)}^{(2,3)} &= 0;\ \mathbf{y}_{(1)}^{(2)} = 0;\ \mathbf{y}_{(1)}^{(3)} = 0;\ \mathbf{y}_{(1)} = 0.
\end{aligned}
$$

Application of forward mode AD to the third-order adjoint code yields the fourth-order adjoint code

$$
\begin{aligned}
\mathbf{y}^{(2,3,4)} &= \langle \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)}, \mathbf{x}^{(3)} \rangle \\
&\quad + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3,4)} \rangle + \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2,3,4)} \rangle \\
\mathbf{y}^{(2,3)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle \\
\mathbf{y}^{(2,4)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(3,4)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(3,4)} \rangle \\
\mathbf{y}^{(3)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(3)} \rangle \\
\mathbf{y}^{(4)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}_{(1)}^{(2,3,4)} &= \mathbf{x}_{(1)}^{(2,3,4)} + \langle \mathbf{y}_{(1)}^{(2,4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3,4)} \rangle + \langle \mathbf{y}_{(1)}^{(2,3,4)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle + \langle \mathbf{y}_{(1)}^{(2,4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(4)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,4)}, \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3,4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3,4)} \rangle \\
\mathbf{x}_{(1)}^{(2,3)} &= \mathbf{x}_{(1)}^{(2,3)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}^{(2,3)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle \\
\mathbf{x}_{(1)}^{(2,4)} &= \mathbf{x}_{(1)}^{(2,4)} + \langle \mathbf{y}_{(1)}^{(2,4)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{x}_{(1)}^{(2)} &= \mathbf{x}_{(1)}^{(2)} + \langle \mathbf{y}_{(1)}^{(2)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)}^{(3,4)} &= \mathbf{x}_{(1)}^{(3,4)} + \langle \mathbf{y}_{(1)}^{(3,4)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}^{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1,4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3,4)} \rangle \\
\mathbf{x}_{(1)}^{(3)} &= \mathbf{x}_{(1)}^{(3)} + \langle \mathbf{y}_{(1)}^{(3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle \\
\mathbf{x}_{(1)}^{(4)} &= \mathbf{x}_{(1)}^{(4)} + \langle \mathbf{y}_{(1)}^{(4)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1)}^{(2,3,4)} &= 0;\ \mathbf{y}_{(1)}^{(2,3)} = 0;\ \mathbf{y}_{(1)}^{(2,4)} = 0;\ \mathbf{y}_{(1)}^{(2)} = 0 \\
\mathbf{y}_{(1)}^{(3,4)} &= 0;\ \mathbf{y}_{(1)}^{(3)} = 0;\ \mathbf{y}_{(1)}^{(4)} = 0;\ \mathbf{y}_{(1)} = 0.
\end{aligned}
$$

Arbitrary projections of the fourth derivative tensor can be computed as

$$
\begin{aligned}
\mathbf{x}_{(1)}^{(2,3,4)} &= \mathbf{x}_{(1)}^{(2,3,4)} + \langle \mathbf{y}_{(1)}^{(2,4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(3,4)} \rangle + \langle \mathbf{y}_{(1)}^{(2,3,4)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle + \langle \mathbf{y}_{(1)}^{(2,4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \mathbf{y}_{(1)}^{(2)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(4)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,4)}, \mathbf{x}^{(3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3,4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}^{(4)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,3)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,3,4)} \rangle
\end{aligned}
$$

by setting $\mathbf{x}$, $\mathbf{y}_{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, and $\mathbf{x}^{(4)}$ appropriately while ensuring that the other terms vanish identically as the result of initializing the remaining inputs to zero. The whole fourth derivative tensor can be accumulated by letting $\mathbf{y}_{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, and $\mathbf{x}^{(4)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$, $\mathbb{R}^n$, $\mathbb{R}^n$, and $\mathbb{R}^n$, respectively, Projections of $\nabla^4 F(\mathbf{x})$ can be obtained at a lower computational cost, for example,

- $\langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}$ at the cost of $O(1) \cdot Cost(F)$;
- $\langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}^m$ at the cost of $O(m) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^m$);
- $\langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \in R^n$ at the cost of $O(n) \cdot Cost(F)$ ($\mathbf{x}^{(3)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^n$);
- $\langle \nabla^4 F(\mathbf{x}), \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}^{m \times n}$ at the cost of $O(m \cdot n) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ and $\mathbf{x}^{(2)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively).

Moreover, the fourth-order adjoint code returns arbitrary projections of the third, second, and first derivative tensors in addition to the original function value.

Potential sparsity should be exploited to reduce the cost of computing certain required projections.

- Fourth-order adjoint code in forward-over-reverse-over-forward-over-reverse mode:
  Application of reverse mode AD to an implementation of $\mathbf{y} = F(\mathbf{x})$ yields the first-order adjoint code

$$\begin{aligned}
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1)} &= 0.
\end{aligned}$$

  Application of forward mode AD to the first-order adjoint code yields the second-order adjoint code

$$\begin{aligned}
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}^{(2)}_{(1)} &= \mathbf{x}^{(2)}_{(1)} + \langle \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(2)}_{(1)} &= 0 \\
\mathbf{y}_{(1)} &= 0.
\end{aligned}$$

  Application of reverse mode AD with required floating-point data stack s to the second-order adjoint code yields the third-order adjoint code with augmented forward section

$$\begin{aligned}
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}^{(2)}_{(1)} &= \mathbf{x}^{(2)}_{(1)} + \langle \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
s[0] &= \mathbf{y}^{(2)}_{(1)};\ \mathbf{y}^{(2)}_{(1)} = 0 \\
s[1] &= \mathbf{y}_{(1)};\ \mathbf{y}_{(1)} = 0
\end{aligned}$$

  and reverse section

$$\begin{aligned}
\mathbf{y}_{(1)} &= s[1];\ \mathbf{y}_{(1,3)} = 0 \\
\mathbf{y}^{(2)}_{(1)} &= s[1];\ \mathbf{y}^{(2)}_{(1,3)} = 0 \\
\mathbf{y}_{(1,3)} &= \mathbf{y}_{(1,3)} + \langle \mathbf{x}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{y}^{(2)}_{(1,3)} &= \mathbf{y}^{(2)}_{(1,3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle
\end{aligned}$$

$$
\begin{aligned}
\mathbf{y}_{(1,3)} &= \mathbf{y}_{(1,3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}^{(2)}_{(3)} &= \mathbf{x}^{(2)}_{(3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(3)} &= 0 \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}^{(2)}_{(3)} &= \mathbf{x}^{(2)}_{(3)} + \langle \mathbf{y}^{(2)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(2)}_{(3)} &= 0.
\end{aligned}
$$

The resulting third-order adjoint code computes

$$
\begin{aligned}
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}^{(2)}_{(1)} &= \mathbf{x}^{(2)}_{(1)} + \langle \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(1,3)} &= \langle \mathbf{x}_{(1,3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(2)}_{(1,3)} &= \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}^{(2)}_{(3)} &= \mathbf{x}^{(2)}_{(3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(2)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}_{(3)} &= 0;\ \mathbf{y}^{(2)}_{(3)} = 0.
\end{aligned}
$$

Application of forward mode AD to this third-order adjoint code yields the fourth-order adjoint code

$$
\begin{aligned}
\mathbf{y}^{(2,4)} &= \langle \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
\mathbf{y}^{(2)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(4)} &= \langle \nabla F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
\mathbf{y} &= F(\mathbf{x}) \\
\mathbf{x}^{(2,4)}_{(1)} &= \mathbf{x}^{(2,4)}_{(1)} + \langle \mathbf{y}^{(2,4)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}^{(4)}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
\mathbf{x}^{(2)}_{(1)} &= \mathbf{x}^{(2)}_{(1)} + \langle \mathbf{y}^{(2)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}^{(4)}_{(1)} &= \mathbf{x}^{(4)}_{(1)} + \langle \mathbf{y}^{(4)}_{(1)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + \langle \mathbf{y}_{(1)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(4)}_{(1,3)} &= \langle \mathbf{x}^{(4)}_{(1,3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
\mathbf{y}_{(1,3)} &= \langle \mathbf{x}_{(1,3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{x}^{(4)}_{(3)} &= \mathbf{x}^{(4)}_{(3)} + \langle \mathbf{x}^{(4)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(1,3)}, \mathbf{y}^{(4)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2,4)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(4)}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
&\quad + \langle \mathbf{y}^{(4)}_{(3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}^{(2,4)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
\mathbf{x}_{(3)} &= \mathbf{x}_{(3)} + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \\
\mathbf{y}^{(2,4)}_{(1,3)} &= \langle \mathbf{x}^{(2,4)}_{(1,3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
\mathbf{y}^{(2)}_{(1,3)} &= \langle \mathbf{x}^{(2)}_{(1,3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{x}^{(2,4)}_{(3)} &= \mathbf{x}^{(2,4)}_{(3)} + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(4)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle + \langle \mathbf{y}^{(2,4)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
\mathbf{x}^{(2)}_{(3)} &= \mathbf{x}^{(2)}_{(3)} + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{y}^{(2)}_{(3)}, \nabla F(\mathbf{x}) \rangle \\
\mathbf{y}^{(4)}_{(3)} &= 0;\ \mathbf{y}_{(3)} = 0;\ \mathbf{y}^{(2,4)}_{(3)} = 0;\ \mathbf{y}^{(2)}_{(3)} = 0.
\end{aligned}
$$

Arbitrary projections of the fourth derivative tensor can be computed as

$$
\begin{aligned}
\mathbf{x}^{(4)}_{(3)} &= \mathbf{x}^{(4)}_{(3)} + \langle \mathbf{x}^{(4)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}_{(1,3)}, \mathbf{y}^{(4)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle \\
&\quad + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2,4)}_{(1)}, \nabla^2 F(\mathbf{x}) \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(2)}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&\quad + \langle \mathbf{x}^{(2,4)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}^{(4)}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&+ \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle + \langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle \\
&+ \langle \mathbf{y}^{(4)}_{(3)}, \nabla F(\mathbf{x}) \rangle + \langle \mathbf{y}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(4)} \rangle \\
&+ \langle \mathbf{y}^{(2,4)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle + \langle \mathbf{y}^{(2)}_{(3)}, \nabla^3 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \\
&+ \langle \mathbf{y}^{(2)}_{(3)}, \nabla^2 F(\mathbf{x}), \mathbf{x}^{(2,4)} \rangle
\end{aligned}
$$

by setting $\mathbf{x}$, $\mathbf{x}^{(2)}_{(1,3)}$, $\mathbf{y}_{(1)}$, $\mathbf{x}^{(2)}$, and $\mathbf{x}^{(4)}$ appropriately while ensuring that the other terms vanish identically as the result of initializing the remaining inputs to zero. The whole fourth derivative tensor can be accumulated by letting $\mathbf{x}^{(2)}_{(1,3)}$, $\mathbf{y}_{(1)}$, $\mathbf{x}^{(2)}$, and $\mathbf{x}^{(4)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^n$, $\mathbb{R}^m$, $\mathbb{R}^n$, and $\mathbb{R}^n$, respectively. Projections of $\nabla^4 F(\mathbf{x})$ can be obtained at a lower computational cost, for example,

- $\langle \mathbf{x}^{(2)}_{(1,3)}, \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}$ at the cost of $O(1) \cdot Cost(F)$;
- $\langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}^m$ at the cost of $O(m) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^m$);
- $\langle \mathbf{y}_{(1)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)}, \mathbf{x}^{(4)} \rangle \in \mathbb{R}^n$ at the cost of $O(n) \cdot Cost(F)$ ($\mathbf{x}^{(2)}_{(1,3)}$ ranges over the Cartesian basis vectors in $\mathbb{R}^n$);
- $\langle \mathbf{x}^{(2)}_{(1,3)}, \nabla^4 F(\mathbf{x}), \mathbf{x}^{(2)} \rangle \in \mathbb{R}^{m \times n}$ at the cost of $O(m \cdot n) \cdot Cost(F)$ ($\mathbf{y}_{(1)}$ and $\mathbf{x}^{(4)}$ range independently over the Cartesian basis vectors in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively).

Moreover, the fourth-order adjoint code returns arbitrary projections of the third, second, and first derivative tensors in addition to the original function value. Potential sparsity should be exploited to reduce the cost of computing certain required projections.

C.4 Chapter 4

C.4.1 Exercise 4.7.1

Derive DFAs for recognizing the languages that are defined by the following regular expressions:

1. `0|1+(0|1)*`.
2. `0+|1(0|1)+`.

Implement scanners for these languages with `flex` *and* `gcc`*. Compare the NFAs and DFAs derived by yourself with the ones that are generated by* `flex`*.*

Refer to Figures C.3, C.4, and C.5 for the NFAs and DFAs. Transitions into the dedicated error states are omitted.

The corresponding `flex` input files are analogous to the one discussed in Section 4.3.4. Running `flex` with the `-T` option produces diagnostic output that contains the automata shown in Figures C.3, C.4, and C.5.

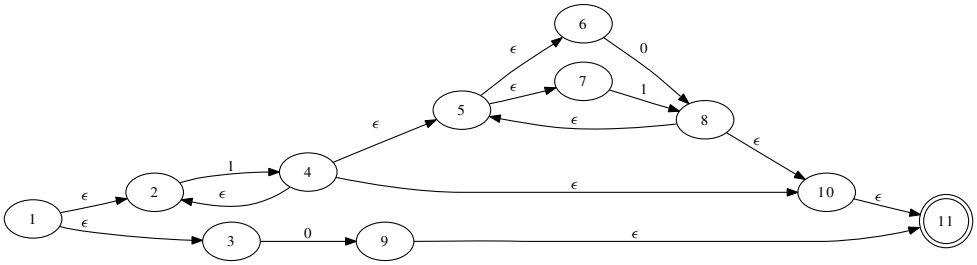


Figure C.3. *NFA for* `0|1+(0|1)*`.

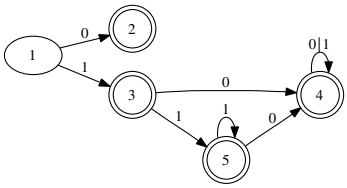


Figure C.4. *DFA for* `0|1+(0|1)*`.

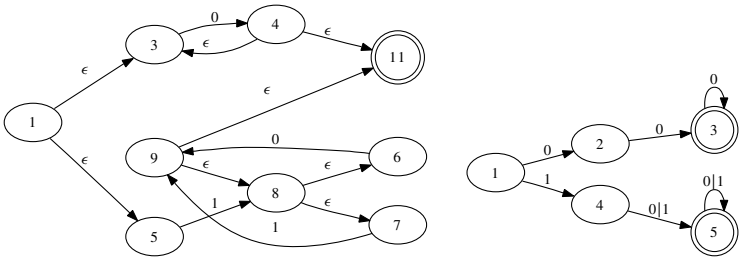


Figure C.5. *NFA (left) and DFA (right) for* `0+|1(0|1)+`.

C.4.2 Exercise 4.7.2

1. *Use the parser for SL*^{2} *to parse the assignment "*$y = \sin(x) + x * 2;$*" as shown in Table* 4.3*. Draw the parse tree.*

   Refer to Table C.2 for illustration. The parse tree is derived by applying the reductions in the ACTION column in reverse order.

2. *Extend SL*^{2} *and its parser to include the ternary* fused-multiply-add *operation, defined as* $y = \texttt{fma}(a,b,c) \equiv a * b + c$*. Derive the characteristic automaton.*

   Both the `flex` and the `bison` input files are listed below.

Table C.2. *Parsing "$V = F(V)LVNC;$"*

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(V)LVNC;$	S
2	0	1	$V =$	$F(V)LVNC;$	S
3	0,1	4	$V = F$	$(V)LVNC;$	S
4	0,1,4	9	$V = F($	$V)LVNC;$	S
5	0,1,4,9	11	$V = F(V$	$)LVNC;$	S
6	0,1,4,9,11	7		$)LVNC;$	R($P7$)
7	0,1,4,9	11	$V = F(e$	$)LVNC;$	S
8	0,1,4,9,11	15	$V = F(e)$	$LVNC;$	S
9	0,1,4,9,11,15	18		$LVNC;$	R($P6$)
10	0,1	4	$V = e$	$LVNC;$	S
11	0,1,4	10	$V = eL$	$VNC;$	S
12	0,1,4,10	12	$V = eLV$	$NC;$	S
13	0,1,4,10,12	7		$NC;$	R($P7$)
14	0,1,4,10	12	$V = eLe$	$NC;$	S
15	0,1,4,10,12	16	$V = eLeN$	$C;$	S
16	0,1,4,10,12,16	13	$V = eLeNC$	;	S
17	0,1,4,10,12,16,13	8		;	R($P8$)
18	0,1,4,10,12,16	13	$V = eLeNe$	;	S
19	0,1,4,10,12,16,13	17		;	R($P5$)
20	0,1,4,10	12	$V = eLe$	;	S
21	0,1,4,10,12	16		;	R($P4$)
22	0,1	4	$V = e$	;	S
23	0,1,4	10	$V = e;$		S
24	0,1,4,10	14			R($P3$)
25		0	a		S
26	0	3			R($P1$)
27		0	s		S
28	0	2	$s\$end$		S
29	0,2	5			R($P0$)
30		0	$\$accept$		ACCEPT

Listing C.12. `flex` *input file.*

```
%{
#include "parser.tab.h"
%}

whitespace      [ \t\n]+
variable        [a-z]
constant        [0-9]

%%

{whitespace}    { }
"sin"           { return F; }
"fma"           { return T; }
"+"             { return L; }
"*"             { return N; }
```

```
{variable}  { return V; }
{constant}  { return C; }
.           { return yytext[0]; }

%%

void lexinit(FILE *source) { yyin=source; }
```

Listing C.13. `bison` *input file.*

```
%token V C F T L N

%left L
%left N

%%

s : a
  | a s
  ;
a : V '=' e ';' ;
e : e L e
  | e N e
  | F '(' e ')'
  | T '(' e ',' e ',' e ')'
  | V
  | C
  ;

%%

#include<stdio.h>

int yyerror(char *msg) {
  printf("ERROR: %s \n",msg);
  return -1;
}

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

Run `bison -v parser.y` to generate the characteristic automaton.

3. *Use* `flex` *and* `bison` *to implement a parser for SL programs that prints a syntactically equivalent copy of the input code.*

A syntax-directed unparser for SL is a straightforward extension of the SL^2 unparser. The `flex` input file is extended with scanner rules for the additional key words.

```
...
%%
...
"if"              { return IF; }
"while"           { return WHILE; }
  ...
%%
...
```

Corresponding new tokens are defined in the `bison` input file in addition to proper actions associated with the loop and branch statements.

```
...
%token V C F IF WHILE O R
...
%%

s : a | a s | b | b s | l | l s ;
b : IF '('
      { printf("if ("); }
    c ')'
      '{' { printf(") {\n"); }
    s '}'
      { printf("}\n"); } ;
l : WHILE '('
      { printf("while ("); }
    c ')' '{'
      { printf(") { \n"); }
    s '}'
      { printf("}\n"); } ;
c : V R V { printf("%s%s%s",$1,$2,$3); } ;
...
%%
...
```

C.4.3 Exercise 4.7.3

1. *Use* `flex` *and* `bison` *to implement a single-pass tangent-linear code compiler for* SL^2 *programs. Extend it to SL.*

   The corresponding `flex` and `bison` input files for SL^2 are listed below. The extension to SL is straightforward as control-flow statements are simply unparsed. Refer to Section 4.5.3 for conceptual details on this syntax-directed tangent-linear code compiler.

Listing C.14. *Definition of parse tree node.*

```
#define BUFFER_SIZE 100000

typedef struct {
  int j;
  char* c;
} astNodeType;

#define YYSTYPE astNodeType
```

Listing C.15. `flex` *input file.*

```
%{
#include "ast.h"
#include "parser.tab.h"

#include<stdlib.h> // malloc
#include<string.h> // strcpy

void to_parser() {
  yylval.c=(char*)malloc(BUFFER_SIZE*sizeof(char));
  strcpy(yylval.c,yytext);
}
%}

whitespace      [ \t\n]+
variable    [a-z]
constant    [0-9]

%%

{whitespace}    { }
"sin"       { to_parser(); return F; }
"+"             { to_parser(); return L; }
"*"             { to_parser(); return N; }
{variable}      { to_parser(); return V; }
{constant}      { to_parser(); return C; }
.       { return yytext[0]; }

%%

void lexinit(FILE *source) { yyin=source; }
```

Listing C.16. `bison` *input file.*

```
%{

#include <stdio.h>
#include <stdlib.h>
#include "ast.h"
```

```
extern int yylex();
extern void lexinit(FILE*);

static int sacvc; // SAC variable counter

void get_memory(YYSTYPE* v) {
  v->c=malloc(BUFFER_SIZE*sizeof(char));
}
void free_memory(YYSTYPE* v) {
  if (v->c) free(v->c);
}

%}

%token V C F L N

%left L
%left N

%%

sl_program : s
  {
    printf("%s",$1.c);
    free_memory(&$1);
  }
  ;
s : a
  | a s
  {
    get_memory(&$$);
    sprintf($$.c,"%s%s",$1.c,$2.c);
    free_memory(&$1); free_memory(&$2);
  }
  ;
a : V '='
  {
    sacvc=0;
  }
  e ';'
  {
    get_memory(&$$);
    sprintf($$.c,"%s%s_=v%d_; %s=v%d; \n",
                 $4.c,$1.c,$4.j,$1.c,$4.j);
    free_memory(&$1); free_memory(&$4);
  }
  ;
e : e L e
  {
```

```
      $$.j=sacvc++;
      get_memory(&$$);
      sprintf($$.c,"%s%sv%d_=v%d_%sv%d_; v%d=v%d%sv%d; \n",
                   $1.c,$3.c,$$.j,$1.j,$2.c,$3.j,
                   $$.j,$1.j,$2.c,$3.j);
      free_memory(&$1);
    }
    |e N e
    {
      if (!strcmp($2.c,"*")) {
        $$.j=sacvc++;
        get_memory(&$$);
        sprintf($$.c,
          "%s%sv%d_=v%d_*v%d+v%d*v%d_; v%d=v%d%sv%d; \n",
           $1.c,$3.c,$$.j,$1.j,$3.j,$1.j,$3.j,
           $$.j,$1.j,$2.c,$3.j);
        free_memory(&$1);
      }
    }
    | F '(' e ')'
    {
      if (!strcmp($2.c,"sin")) {
        $$.j=sacvc++;
        get_memory(&$$);
        sprintf($$.c,"%sv%d_=cos(v%d)*v%d_; v%d=sin(v%d);\n",
                     $3.c,$$.j,$3.j,$3.j,$$.j,$3.j);
        free_memory(&$3);
      }
    }
    | V
    {
      $$.j=sacvc++;
      get_memory(&$$);
      sprintf($$.c,"v%d_=%s_; v%d=%s;\n",$$.j,$1.c,$$.j,$1.c);
      free_memory(&$1);
    }
    | C
    {
      $$.j=sacvc++;
      get_memory(&$$);
      sprintf($$.c,"v%d_=0; v%d=%s;\n",$$.j,$$.j,$1.c);
      free_memory(&$1);
    }
    ;

%%

int yyerror(char *msg) {
  printf("ERROR: %s \n",msg);
  return -1;
```

```
}

int main(int argc,char** argv) {
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

2. *Use* `flex` *and* `bison` *to implement a single-pass adjoint code compiler for SL*^{2} *programs. Extend it to SL.*

The corresponding `flex` and `bison` input files for SL are listed below. A solution of SL^2 is implied. Refer to Section 4.5.4 for conceptually details of the syntax-directed adjoint code compiler.

Listing C.17. *Definition of parse tree node.*

```
#define maxBB 100

typedef struct {
  int j;
  char* af;
  char* ar[maxBB];
} astNodeType;

#define YYSTYPE astNodeType
```

Listing C.18. `flex` *input file.*

```
%{
#include <string.h>
#include "ast.h"
#include "parser.tab.h"
%}

whitespace      [ \t\n]+
symbol      [a-z]
const       [0-9]

%%

{whitespace} { }
"if" { return IF; }
"while" { return WHILE; }
"sin" {
  yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
  strcpy(yylval.af,yytext);
  int i;
```

```
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return F;
  }
  "<" {
    yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
    strcpy(yylval.af,yytext);
    int i;
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return R;
  }
  "+" {
    yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
    strcpy(yylval.af,yytext);
    int i;
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return L;
  }
  "*" {
    yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
    strcpy(yylval.af,yytext);
    int i;
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return N;
  }
  {symbol} {
    yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
    strcpy(yylval.af,yytext);
    int i;
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return V;
  }
  {const} {
    yylval.af=(char*)malloc((strlen(yytext)+1)*sizeof(char));
    strcpy(yylval.af,yytext);
    int i;
    for (i=0;i<maxBB;i++) yylval.ar[i]=0;
    yylval.j=0;
    return C;
  }
  . { return yytext[0]; }

  %%

  void lexinit(FILE *source) { yyin=source; }
```

Listing C.19. `bison` *input file.*

```
%{

#include <stdio.h>
#include <string.h>
#include "ast.h"

extern int yylex();
extern void lexinit(FILE*);

static int c=1,cmax=1,bs=1000;
static int newBB=0;
static int idxBB=0;

%}

%token V C F L N R IF WHILE

%left L
%left N

%%

sl_program : ss
  {
    for (c=1;c<cmax;c++) printf("double v%d, v%d_;\n",c,c);
    printf("%s",$1.af);
    free($1.af);
    int i;
    printf("int i_;\n");
    printf("while (pop_c(i_)) {\n");
    for (i=0;i<=idxBB;i++) {
      if (i==0)
        printf("if");
      else
        printf("else if");
      if ($1.ar[i])
        printf(" (i_==%d) {\n%s}\n",i,$1.ar[i]);
      else
        printf(" (i_==%d) {\n}\n",i);
      free($1.ar[i]);
    }
    printf("}\n");
  }
  ;
ss : s { $$=$1; }
  | s ss
  {
```

```
      $$.af=(char*)malloc(bs*sizeof(char));
      sprintf($$.af,"%s%s",$1.af,$2.af);
      free($2.af); free($1.af);

      int i;
      for (i=0;i<=idxBB;i++) {
        if ($2.ar[i]&&$1.ar[i]) {
          $$.ar[i]=(char*)malloc(bs*sizeof(char));
          sprintf($$.ar[i],"%s%s",$2.ar[i],$1.ar[i]);
          free($2.ar[i]); free($1.ar[i]);
        }
        else if ($2.ar[i]) {
          $$.ar[i]=(char*)malloc(bs*sizeof(char));
          sprintf($$.ar[i],"%s",$2.ar[i]);
          free($2.ar[i]);
        }
        else if ($1.ar[i]) {
          $$.ar[i]=(char*)malloc(bs*sizeof(char));
          sprintf($$.ar[i],"%s",$1.ar[i]);
          free($1.ar[i]);
        }
      }
    }
    ;
s : a { $$=$1; }
  | b { $$=$1; }
  | l { $$=$1; }
  ;
b : IF '(' c ')' '{'
    {
      newBB=1;
    }
    ss '}'
    {
      $$.af=(char*)malloc(bs*sizeof(char));
      sprintf($$.af,"if (%s) {\n%s}\n",$3.af,$7.af);
      free($3.af); free($7.af);
      int i;
      for (i=0;i<=idxBB;i++) {
        if ($7.ar[i]) {
          $$.ar[i]=(char*)malloc(bs*sizeof(char));
          sprintf($$.ar[i],"%s",$7.ar[i]);
          free($7.ar[i]);
        }
      }
      newBB=1;
    }
    ;
l : WHILE '(' c ')' '{'
    {
```

```
      newBB=1;
    }
    ss '}'
    {
      $$.af=(char*)malloc(bs*sizeof(char));
      sprintf($$.af,"while (%s) {\n%s}\n",$3.af,$7.af);
      free($3.af); free($7.af);
      int i;
      for (i=0;i<=idxBB;i++) {
        if ($7.ar[i]) {
          $$.ar[i]=(char*)malloc(bs*sizeof(char));
          sprintf($$.ar[i],"%s",$7.ar[i]);
          free($7.ar[i]);
        }
      }
      newBB=1;
    }
    ;
c : V R V
    {
      $$.af=(char*)malloc(bs*sizeof(char));
      if (!strcmp($2.af,"<"))
        sprintf($$.af,"%s%s%s",$1.af,$2.af,$3.af);
      free($1.af); free($3.af);
    }
a : V '='
    {
      if (newBB) idxBB++;
    }
    e ';'
    {
      if (newBB||!idxBB) {
        $$.af=(char*)malloc(bs*sizeof(char));
        sprintf($$.af,"push_c(%d);\n%spush_d(%s); %s=v%d;\n",
                      idxBB,$4.af,$1.af,$1.af,$4.j);
      }
      else {
        $$.af=(char*)malloc(bs*sizeof(char));
        sprintf($$.af,"%spush_d(%s); %s=v%d;\n",
                      $4.af,$1.af,$1.af,$4.j);
      }
      $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
      sprintf($$.ar[idxBB],"pop_d(%s); v%d_=%s_; %s_=0;\n%s",
                       $1.af,$4.j,$1.af,$1.af,$4.ar[idxBB]);
      free($4.ar[idxBB]);
      newBB=0;
      free($1.af); free($4.af);
      c=1;
    }
    ;
```

```
e : e N e
  {
    $$.j=c++; if (c>cmax) cmax=c;
    $$.af=(char*)malloc(bs*sizeof(char));
    sprintf($$.af,"%s%spush_d(v%d); v%d=v%d * v%d;\n",
                  $1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
    free($1.af); free($3.af);
    $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
    sprintf($$.ar[idxBB],
      "pop_d(v%d); v%d_=v%d_*v%d; v%d_=v%d_*v%d;\n%s%s",
       $$.j, $1.j,$$.j,$3.j, $3.j,$$.j,$1.j,
       $3.ar[idxBB],$1.ar[idxBB]);
    free($1.ar[idxBB]); free($3.ar[idxBB]);
  }
  | e L e
  {
    $$.j=c++; if (c>cmax) cmax=c;
    $$.af=(char*)malloc(bs*sizeof(char));
    sprintf($$.af,"%s%spush_d(v%d); v%d=v%d+v%d;\n",
                  $1.af,$3.af,$$.j,$$.j,$1.j,$3.j);
    free($1.af); free($3.af);
    $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
    sprintf($$.ar[idxBB],
      "pop_d(v%d); v%d_=v%d_; v%d_=v%d_;\n%s%s",
       $$.j, $1.j,$$.j, $3.j,$$.j,
       $3.ar[idxBB],$1.ar[idxBB]);
    free($1.ar[idxBB]); free($3.ar[idxBB]);
  }
  | F '(' e ')'
  {
    $$.j=c++; if (c>cmax) cmax=c;
    $$.af=(char*)malloc(bs*sizeof(char));
    sprintf($$.af, "%spush_d(v%d); v%d=%s(v%d);\n",
                  $3.af,$$.j,$$.j,$1.af,$3.j);
    free($3.af);

    $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
    if (!strcmp($1.af,"sin"))
      sprintf($$.ar[idxBB],
        "pop_d(v%d);v%d_=cos(v%d)*v%d_;\n%s",
         $$.j, $3.j,$3.j,$$.j,$3.ar[idxBB]);
    free($3.ar[idxBB]);
  }
  | V
  {
    $$.j=c++; if (c>cmax) cmax=c;
    $$.af=(char*)malloc(bs*sizeof(char));
    sprintf($$.af,"push_d(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

    $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
```

```
        sprintf($$.ar[idxBB],"pop_d(v%d); %s_+=v%d_;\n",
                                $$.j,$1.af,$$.j);
        free($1.af);
      }
      | C
      {
        $$.j=c++; if (c>cmax) cmax=c;
        $$.af=(char*)malloc(bs*sizeof(char));
        sprintf($$.af,"push_d(v%d); v%d=%s;\n",$$.j,$$.j,$1.af);

        $$.ar[idxBB]=(char*)malloc(bs*sizeof(char));
        sprintf($$.ar[idxBB],"pop_d(v%d);\n",$$.j);
        free($1.af);
      }
      ;

%%

int yyerror(char *msg) {
  printf("ERROR: %s \n",msg);
  return -1;
}

int main(int argc,char** argv) {
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

C.4.4 Exercise 4.7.4

Use `flex` *and* `bison` *to implement a compiler that generates an intermediate representation for explicitly typed SL programs in the form of a parse tree and a symbol table. Implement an unparser.*

A fully functional solution is listed below. Refer to Section 4.6 for details.

Listing C.20. *parse_tree.hpp.*

```
#ifndef PARSE_TREE_INC
#define PARSE_TREE_INC

#include <string>
#include <list>
using namespace std;
#include "symbol_table.hpp"
```

```
const unsigned short UNDEFINED_PTV=0;
const unsigned short SEQUENCE_OF_STATEMENTS_PTV=1;
const unsigned short LOOP_PTV=2;
const unsigned short BRANCH_PTV=3;
const unsigned short ASSIGNMENT_PTV=4;
const unsigned short INTRINSIC_CALL_PTV=5;
const unsigned short ADDITION_PTV=6;
const unsigned short MULTIPLICATION_PTV=7;
const unsigned short SYMBOL_PTV=8;
const unsigned short CONSTANT_PTV=9;
const unsigned short LT_CONDITION_PTV=10;
const unsigned short PARENTHESES_PTV=11;


class parse_tree_vertex {
public:
  unsigned short type;
  list<parse_tree_vertex*> succ;
  parse_tree_vertex(unsigned short);
  virtual ~parse_tree_vertex();
  virtual const string& get_name() const;
  virtual int& symbol_type();
  virtual void unparse() const;
};

class parse_tree_vertex_named : public parse_tree_vertex {
public:
  string name;
  parse_tree_vertex_named(unsigned short, string);
  ~parse_tree_vertex_named();
  const string& get_name() const;
  void unparse() const;
};

class parse_tree_vertex_symbol : public parse_tree_vertex {
public:
  symbol* sym;
  parse_tree_vertex_symbol(unsigned short, string);
  ~parse_tree_vertex_symbol();
  void unparse() const;
  int& symbol_type();
};

#define YYSTYPE parse_tree_vertex*

#endif
```

Listing C.21. *parse_tree.cpp.*

```
#include <assert.h>
#include <iostream>
using namespace std;
#include "parse_tree.hpp"

extern symbol_table stab;

parse_tree_vertex::parse_tree_vertex(unsigned short t)
  : type(t) {}

parse_tree_vertex::~parse_tree_vertex() {
  list<parse_tree_vertex*>::iterator i;
  for (i=succ.begin();i!=succ.end();i++) {
    delete (*i);
  }
}

const string& parse_tree_vertex::get_name() const {
  assert(false);
}
int& parse_tree_vertex::symbol_type() {
  assert(false);
}

void parse_tree_vertex::unparse() const {
  switch (type) {
    case SEQUENCE_OF_STATEMENTS_PTV : {
     list<parse_tree_vertex*>::const_iterator i;
     for (i=succ.begin();i!=succ.end();i++)
        (*i)->unparse();
      break;
    }
    case LOOP_PTV : {
      list<parse_tree_vertex*>::const_iterator i;
      cout << "while (";
      (*(succ.begin()))->unparse();
      cout << ") {" << endl;
      (*(++(succ.begin())))->unparse();
      cout << "}" << endl;
      break;
    }
    case BRANCH_PTV : {
      list<parse_tree_vertex*>::const_iterator i;
      cout << "if (";
      (*(succ.begin()))->unparse();
      cout << ") {" << endl;
      (*(++(succ.begin())))->unparse();
      cout << "}" << endl;
```

```
      break;
    }
    case ADDITION_PTV : {
      list<parse_tree_vertex*>::const_iterator i=succ.begin();
      (*i++)->unparse();
      cout << "+";
      (*i)->unparse();
      break;
    }
    case MULTIPLICATION_PTV : {
      (*(succ.begin()))->unparse();
      cout << "*";
      (*(++(succ.begin())))->unparse();
      break;
    }
    case LT_CONDITION_PTV : {
      (*(succ.begin()))->unparse();
      cout << "<";
      (*(++(succ.begin())))->unparse();
      break;
    }
    case PARENTHESES_PTV : {
      cout << "(";
      (*(succ.begin()))->unparse();
      cout << ")";
      break;
    }
  }
}

parse_tree_vertex_named::parse_tree_vertex_named
  (unsigned short t, string n) :
  parse_tree_vertex(t), name(n) { }

parse_tree_vertex_named::~parse_tree_vertex_named() {}

const string& parse_tree_vertex_named::get_name() const {
  return name;
}

void parse_tree_vertex_named::unparse() const {
  switch (type) {
    case CONSTANT_PTV : {
      cout << name;
      break;
    }
    case INTRINSIC_CALL_PTV : {
      cout << name << "(";
      (*(succ.begin()))->unparse();
      cout << ")";
```

```
      break;
    }
  }
}

parse_tree_vertex_symbol::parse_tree_vertex_symbol(i
     unsigned short t, string n) : parse_tree_vertex(t) {
  sym=stab.insert(n);
}

parse_tree_vertex_symbol::~parse_tree_vertex_symbol() {}

int& parse_tree_vertex_symbol::symbol_type() {
  return sym->type;
}

void parse_tree_vertex_symbol::unparse() const {
  switch (type) {
    case ASSIGNMENT_PTV : {
      cout << sym->name << "=";
      (*(succ.begin()))->unparse();
      cout << ";" << endl;
      break;
    }
    case SYMBOL_PTV : {
      cout << sym->name;
      break;
    }
  }
}
```

Listing C.22. *symbol_table.hpp.*

```
#ifndef SYMBOL_TABLE
#define SYMBOL_TABLE

#include <list>
#include <string>
using namespace std;

const unsigned short UNDEFINED_ST=0;
const unsigned short INTEGER_ST=1;
const unsigned short FLOAT_ST=2;


/**
 symbol
 */
class symbol {
  public:
```

```
  string name;
  int type;
  symbol();
};

/**
 symbol table
 */
class symbol_table {
public:
/**
 symbol table is stored as simple list of symbols;
 */
  list<symbol*> tab;
  symbol_table();
  ~symbol_table();
/**
 insert a string into the symbol table; checks for duplication.
 */
  symbol* insert(string);
  void unparse() const;
};

#endif
```

Listing C.23. *symbol_table.cpp.*

```
#include<iostream>
using namespace std;
#include "symbol_table.hpp"

symbol::symbol() : type(UNDEFINED_ST) {}

symbol_table::symbol_table() {}

symbol_table::~symbol_table() {
  list<symbol*>::iterator it;
  if(!tab.empty())
    for (it=tab.begin();it!=tab.end();it++) delete *it;
}

symbol* symbol_table::insert(string n) {
  list<symbol*>::iterator tab_it;
  for (tab_it=tab.begin();tab_it!=tab.end();tab_it++)
    if ((*tab_it)->name==n) return *tab_it;
  symbol* sym=new symbol;
  sym->name=n;
  tab.push_back(sym);
  return sym;
}
```

```
void symbol_table::unparse() const {
  list<symbol*>::const_iterator tab_it;
  for (tab_it=tab.begin();tab_it!=tab.end();tab_it++)
    switch ((*tab_it)->type) {
      case INTEGER_ST: {
        cout << "int " << (*tab_it)->name << ";" << endl;
        break;
      }
      case FLOAT_ST: {
        cout << "float " << (*tab_it)->name << ";" << endl;
        break;
      }
    }
}
```

Listing C.24. *parser.y.*

```
%{

#include<assert.h>
#include<iostream>
#include "parse_tree.hpp"

extern int line_counter;
extern int yylex();
extern int yyerror(const char*);
extern void lexinit(FILE*);

extern parse_tree_vertex* pt_root;
%}

%token INT FLOAT IF WHILE F L N R V C

%left L
%left N

%%

sl : d s
  {
    pt_root=$2;
  }
  ;
d :
  | INT V ';' d
  {
    $2->symbol_type()=INTEGER_ST;
  }
  | FLOAT V ';' d
```

```
  {
    $2->symbol_type()=FLOAT_ST;
  }
  ;
s : a
  {
  }
  | a s
  {
    $$=new parse_tree_vertex(SEQUENCE_OF_STATEMENTS_PTV);
    $$->succ.push_back($1);
    $$->succ.push_back($2);
  }
  | b
  {
  }
  | b s
  {
    $$=new parse_tree_vertex(SEQUENCE_OF_STATEMENTS_PTV);
    $$->succ.push_back($1);
    $$->succ.push_back($2);
  }
  | l
  {
  }
  | l s
  {
    $$=new parse_tree_vertex(SEQUENCE_OF_STATEMENTS_PTV);
    $$->succ.push_back($1);
    $$->succ.push_back($2);
  }
  ;
b :IF '(' c ')' '{' s '}'
  {
    $$=new parse_tree_vertex(BRANCH_PTV);
    $$->succ.push_back($3);
    $$->succ.push_back($6);
  }
  ;
l : WHILE '(' c ')' '{' s '}'
  {
    $$=new parse_tree_vertex(LOOP_PTV);
    $$->succ.push_back($3);
    $$->succ.push_back($6);
  }
  ;
c : V R V
  {
    if ($2->get_name()=="<")
      $$=new parse_tree_vertex(LT_CONDITION_PTV);
```

```
    $$->succ.push_back($1);
    $$->succ.push_back($3);
    delete $2; ;
  }
a : V '=' e ';'
  {
    $$=$1; $$->type=ASSIGNMENT_PTV;
    $$->succ.push_back($3);
  }
  ;
e :  e N e
  {
    if ($2->get_name()=="*")
      $$=new parse_tree_vertex(MULTIPLICATION_PTV);
    $$->succ.push_back($1);
    $$->succ.push_back($3);
    delete $2;
  }
  | e L e
  {
    if ($2->get_name()=="+")
      $$=new parse_tree_vertex(ADDITION_PTV);
    $$->succ.push_back($1);
    $$->succ.push_back($3);
    delete $2;
  }
  | F '(' e ')'
  {
    $$=$1;
    $$->type=INTRINSIC_CALL_PTV;
    $$->succ.push_back($3);
  }
  | '(' e ')'
  {
    $$=new parse_tree_vertex(PARENTHESES_PTV);
    $$->succ.push_back($2);
  }
  | V
  {
    $$=$1;
  }
  | C
  {
    $$=$1;
  }
  ;

%%

int yyerror(const char *msg) {
```

```
    cout << "Error: " << msg <<" in line "<< line_counter
         <<" of the input file."<< endl;
    exit(-1);
}
```

Listing C.25. *scanner.l.*

```
%{
#include "parse_tree.hpp"
#include "parser.tab.h"

#include<iostream>
using namespace std;

int line_counter=1;
%}

whitespace      [ \t]+
linefeed        \n
constant    [0-9]
symbol      [a-z]

%%

{whitespace}    { }
{linefeed}  { line_counter++; }
"int"           { return INT; }
"float"         { return FLOAT; }
"if"            { return IF; }
"while"         { return WHILE; }
"sin"           {
  yylval=new parse_tree_vertex_named(UNDEFINED_PTV,yytext);
  return F;
}
"+"             {
  yylval=new parse_tree_vertex_named(UNDEFINED_PTV,yytext);
  return L;
}
"*"             {
  yylval=new parse_tree_vertex_named(UNDEFINED_PTV,yytext);
  return N;
}
"<"             {
  yylval=new parse_tree_vertex_named(UNDEFINED_PTV,yytext);
  return R;
}
{symbol}    {
  yylval=new parse_tree_vertex_symbol(SYMBOL_PTV,yytext);
  return V;
}
```

```
{constant}  {
  yylval=new parse_tree_vertex_named(CONSTANT_PTV,yytext);
  return C;
}
.       { return yytext[0]; }

%%

void lexinit(FILE *source) { yyin=source; }
```

Listing C.26. *compile.cpp.*

```
#include <stdio.h>
#include <cstdlib>
#include <iostream>
#include "parse_tree.hpp"
#include "symbol_table.hpp"

using namespace std;

extern void lexinit(FILE*);
extern void yyparse();

parse_tree_vertex* pt_root;
symbol_table stab;

int main(int argc, char* argv[]) {
  // open source file
  FILE *source_file = fopen(argv[1],"r");
  // parse
  lexinit(source_file);
  yyparse();
  // close source file
  fclose(source_file);

  // unparse
  cout << "int main() {" << endl;
  stab.unparse();
  pt_root->unparse();
  cout << "return 0;" << endl << "}" << endl;

  return 0;
}
```

Bibliography

[1] 754–1985 IEEE standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22: 9–25, 1987.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison-Wesley, Reading, MA, 2007.

[3] P. Amestoy, I. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng.*, 184:501–520, 2000.

[4] H. Anton. *Calculus, 6th edition*. Wiley, 1999.

[5] B. Averik, R. Carter, and J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1991.

[6] F. L. Bauer. Computational graphs and rounding error. *SIAM J. Numer. Anal.*, 11: 87–96, 1974.

[7] B. Bell and J. Burke. Algorithmic differentiation of implicit functions and optimal values. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, Lecture Notes in Comput. Sci. Engrg. 64, Springer, Berlin, 2008, pages 67–77, 2008.

[8] R. Bellmann. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[9] C. Bendtsen and O. Stauning. *FADBAD, A Flexible C++ Package for Automatic Differentiation*. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[10] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proc. Appl. Math. Ser. 89, SIAM, Philadelphia, 1996.

[11] C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, Lecture Notes in Comput. Sci. Engrg. 64, Springer, Berlin, 2008.

[12] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An

updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28:135–151, 2002.

[13] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Comput. Sci. Engrg. 50, Springer, Berlin, 2005.

[14] R. Byrd, J. Nocedal, and R. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer, Berlin, New York, 2006.

[15] N. Chomsky. Three models for the description of language. *IRE Trans. Inform. Theory*, 2:113–124, 1956.

[16] J. Cocke and J. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Technical report, Courant Institute of Mathematical Sciences of New York University, New York, 1970.

[17] T. Coleman and J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Prog.*, 28:243–270, 1984.

[18] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, Springer, New York, 2002.

[19] A. Griewank and G. F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proc. Appl. Math. Ser., SIAM, Philadelphia, 1991.

[20] P. Cousot. Abstract interpretation based formal methods and future challenges. In R. Wilhelm, editor, *Informatics, 10 Years Back – 10 Years Ahead*, number 2000, pages 138–156. Springer, Berlin, New York, 2001.

[21] A. Curtis, M. Powell, and J. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

[22] T Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[23] J. W. Demmel, S. L. Eisenstat, J. P. Gilbert, X. S. Li, and J. L. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20:720–755, 1999.

[24] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[25] N. Dunford and J. T. Schwartz. *Linear Operators, Part 1, General Theory*. Wiley, New York, 1988.

[26] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing/Cengage Learning, Florence, KY, 2002.

[27] E. Gansner, E. Koutsofios, and S. North. *Drawing Graphs with* dot. Technical report, AT&T Research, Middleton, NJ, 2009.

[28] M. Garey and D. Johnson. *Computers and Intractability*. Mathematical Sciences. Freeman, New York, 1979.

[29] F. Garvan. *The MAPLE Book*. Chapman and Hall/CRC, Boca Raton, FL, 2001.

[30] A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Rev.*, 47:629–705, 2005.

[31] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Soft.*, 24:437–474, 1998.

[32] P. Gill and W. Murray. Newton-type methods for unconstrained and linearly constrained optimization. *Math. Prog.*, 7:311–350, 1974.

[33] P. Gill and W. Murray. Conjugate-gradient methods for large-scale nonlinear optimization. Technical Report SOL 79-15, Department of Operations Research, Stanford University, Palo Alto, CA, 1979.

[34] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22:131–167, 1996.

[35] A. Griewank, J. Utke, and A. Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Math. Comp.*, 69:1117–1130, 2000.

[36] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation (Second Edition)*. SIAM, Philadelphia, 2008.

[37] L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Comput. Sci. Engrg. 50, Springer, Berlin, 2005, pages 135–146. Springer, 2005.

[38] L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Comput. Syst.*, 21:1401–1417, 2005.

[39] M. Hestens and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards*, 49:409–436, 1952.

[40] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (International Edition)*. Pearson Education International, Upper Saddle River, NJ, 2003.

[41] J. Huber, U. Naumann, O. Schenk, E. Varnik, and A. Wächter. Algorithmic differentiation and nonlinear optimization for an inverse medium problem. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, Computational Science Series. Chapman & Hall / CRC Press, Taylor and Francis Group, 2011. To appear.

[42] N. M. Josuttis. *The C++ Standard Library – A Tutorial and Reference*. Addison-Wesley, Reading, MA, 1999.

[43] T. Kasami. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Technical Report Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, 1965.

[44] C. T. Kelley. *Solving Nonlinear Equations with Newton's Method*. Fund. Alg. 1, SIAM, Philadelphia, 2003.

[45] D. E. Knuth. *The Art of Computer Programming (Third Edition)*. Addison-Wesley, Reading, MA, 2011.

[46] R. Mecklenburg. *Managing Projects with GNU Make (Third Edition)*. O'Reilly, Sebastopol, CA, 2005.

[47] J. Moré, B. Garbow, and K. Hillstrom. *User Guide for Minpack-1 Technical*. Technical Report ANL-80-74, Mathematical and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1980.

[48] U. Naumann. Call tree reversal is NP-complete. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, Lecture Notes in Comput. Sci. Engrg. 64, Springer, Berlin, 2008, pages 13–21. 2008.

[49] U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 7:402–410, 2009.

[50] J. Nocedal and S. J. Wright. *Numerical Optimization* (Second Edition). Springer-Verlag, New York, 2006.

[51] T. Parr. *The Definite ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, LLC, Lewisville, TX/Raleigh, NC, 2007.

[52] V. Pascual and L. Hascoët. TAPENADE for C. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, Lecture Notes in Comput. Sci. Engrg. 64, Springer, Berlin, 2008, pages 199–210, 2008.

[53] M. Powell. A hybrid method for nonlinear algebraic equations. In P. Rabinowitz, editor, *Numerical Methods for Nonlinear Algebraic Equations*. Gordon and Breach, 1970.

[54] H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *Comput. J.*, 3:175–184, 1960, 1961.

[55] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Comput. Syst.*, 20:475–487, 2004.

[56] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, 1980.

[57] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.

[58] H. A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, Cambridge, UK, 2003.

[59] A. Wächter and L. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Prog.*, 1:25–57, 2006.

[60] R. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7: 463–464, 1964.

[61] S. Wolfram. *The Mathematica Book*. Wolfram Research, Champaign, IL, 2004.

[62] D. Younger. Recognition and parsing of context-free languages in time n^3. *Inform. Control*, 10:189–208, 1967.

Index