# Preface

A fundamental need that occurs across the field of scientific computing is the calculation of partial derivatives. For example, to determine the direction of sharpest ascent of a differentiable function of $n$ variables is to determine the gradient, i.e., the vector of first partial derivatives. The sensitivity of this vector, in turn, is the matrix of second partial derivatives (i.e., Hessian matrix), assuming the original function is twice continuously differentiable. Optimization of multidimensional functions, nonlinear regression, and multidimensional zero-finding represent broad and significant numerical tasks that rely heavily on the repeated calculation of derivatives. The vast majority of modern methods for solving these numerical problems demand the repeated calculation of partial derivatives. Hence, across numerous application areas, determining partial derivatives accurately and efficiently (in space and time) is of paramount importance.

Computational finance is one such area relying on derivative calculations. Determining the "Greeks" corresponding to (models of) financial instruments comes down to the determination of first derivatives. While this is an easy exercise for simple instruments, it can be a complex and expensive task for "exotic" instruments. Derivatives are required to measure sensitivity to parameters (such as initial stock price, volatility, interest rates) and subsequently determine hedging portfolios. Second derivatives are sometimes used to further augment this hedging strategy. For more discussion on the use of automatic differentiation in finance, see Chapter 7.

Engineering applications involve the differentiation of complex codes. Automatic differentiation (AD) can be applied, straightforwardly, to get all necessary partial derivatives (usually first and possibly second derivatives) regardless of the complexity of the code. However, the space and time efficiency of AD can be dramatically improved, sometimes transforming a problem from intractable to highly feasible, if inherent problem structure is used to apply AD in a judicious manner. This is an important matter, and we devote significant parts of Chapters 2, 5, 4, 7, and 8 to these issues.

The ideas and concepts presented in this book are largely independent of computer languages. AD tools now exist in many computer languages. However, we develop all our examples in the popular MATLAB environment and illustrate with the general differentiator ADMAT[1] for use in MATLAB. A version of ADMAT is available for this book from www.siam.org/books/se27. The reader is encouraged to try test examples with ADMAT. Occasionally some MATLAB codes also contain snippets of C or Fortran for a variety of reasons. ADMAT can handle such codes—as illustrated with some examples in Chapter 6—though the C or Fortran snippets are themselves approximately differenti-

[1]The user manual for ADMAT 2.0 is given in [26]. Code is available in [95].

ated via finite differencing. Generally, we recommend that the codes be 100% MATLAB if possible.

Why AD? It is our belief that AD is generally the best available technology for obtaining derivatives of codes used in scientific computing. Let us consider the other possibilities:

1. Finite differencing is by far the most common method in use today. It is popular primarily because it is easy to understand and easy to implement, requiring no additional tools. However, it is a seriously flawed approach, and we advise against its use. First, it requires a differencing parameter that is difficult to choose well in any general sense. The accuracy of the derivative depends on this choice, and a good choice can be challenging [53]. AD, in contrast, requires no such parameter and is highly accurate. In addition, finite differencing can require considerably more computing time compared to AD. An example of this can be seen in gradient computation (see Chapter 1 for more details).

2. Hand coding of the derivative functions is also a popular approach. If done correctly, this option can be useful yet laborious. Moreover, it is incredibly easy to err in hand coding derivative functions and notoriously hard to find the error. In addition, hand-coded functions cannot match the speed of AD in many cases (see Chapter 1 for examples), whereas in principle AD cannot be outperformed by hand-coded derivative functions.

3. Symbolic differentiation is a technology that differentiates a program at symbol-by-symbol, independent of iterate values, and produces code for the derivatives [76]. However, this approach has two serious drawbacks. First, it can be very costly and therefore appears not to be competitive for "large-scale" problems. Second, and more subtly, the derivative code that is produced may actually represent a poor method for evaluating the derivative in terms of round-off error magnification.

In summary, AD has strong advantages over alternative ways to compute derivatives. Many of the "too much space" and "too much time" criticisms of several years ago have been overcome with recent technical advances.

## Automatic Differentiation Background

AD has a long history: Newton and Leibniz applied derivative computation in differential equations instead of symbolic formulae. In 1950, compilers were developed to compute derivatives by the chain rule. There are several early independent contributors to the field of automatic differentiation. Iri et al. [66, 67, 68], Speelpenning [92], and Rall [84, 85, 86, 87, 88, 89, 90] are three of the AD pioneers. In our view Andreas Griewank, Chris Bischof, and their colleagues can be credited for much of the recent interest in AD as well as the reemergence of AD as a very practical tool in scientific computing; see e.g., [4, 55, 57].

There are two basic modes of AD: the forward mode and the reverse mode. The forward mode is a straightforward implementation of the chain rule. The earliest compilers [6, 99, 100] were implemented in this mode. Later, the potential of the reverse mode was realized by several independent researchers [20, 75, 91, 92, 101]. Compared to the forward mode, the reverse mode records each intermediate operation of the differentiating function on a "tape," then rolls back from the end of the "tape" to the beginning to obtain the

derivative. When computing a gradient for a large problem, reverse mode requires much lower complexity than the forward mode, but it also requires significantly more memory. Reverse mode versus forward mode often represents a trade-off between complexity and space. In order to deal with the massive space requirement problem sometimes posed by reverse mode, considerable work has been done, roughly described under the banner of a computer science technique known as "checkpointing" [59]. Checkpointing is a general procedure that can be defined on a computational graph of a function to evaluate and differentiate an arbitrary differentiable function $z = f(x)$, where for convenience we restrict our discussion to a scalar-valued function of $n$-variables, i.e., $f : \mathbb{R}^n \to \mathbb{R}$. The general idea is to cut the computational graph at a finite set of places (checkpoints) in a forward evaluation of the function $f(x)$, saving relevant state information at each of those checkpoints, and then recompute the information between checkpoints in a backward sweep in which the derivatives are computed. The advantage is potentially considerable savings in required space (compared to a reverse-mode evaluation of the gradient, which requires saving the entire computational graph). There is some computing cost, but the total cost is just a constant factor times the cost to evaluate the function (i.e., the cost to perform the forward sweep). So this checkpointing procedure is the same order of work as straight reverse-mode AD but uses considerably less space.

There are many ways to proceed with this general checkpointing concept—some involve machine/environment specific aspects, some involve user interference, and all involve making choices about where to put the checkpoints and what state information to store [80]. Examples of checkpointing manifestations are included in [65]. Typically the user of the checkpointing idea either has to explicitly, in a "hands-on" way, decide on these questions for a particular application or allow a checkpoint/AD tool to decide. There is no "well-accepted" automatic way (that we know of) to choose optimal checkpoints, and choice of checkpoint location clearly matters. The authors in [43] have done some work on this matter, with reference to determining the general Jacobian matrix: one conclusion is that it is very hard to choose an ideal set of checkpoints in an automatic way, given the code to evaluate $f(x)$. Some AD software apparently does decide automatically on checkpoint location (and which state variable to save), though the actual rules that are used are, in our view, somewhat opaque.

There is also considerable interest in the AD community in explicitly reformulating "macro-steps" (such as implicit equation solution, or matrix equations) as an alternative to "checkpointing," as "elemental" operations and reusing by-products of the forward step to expedite the corresponding adjoint step. An example of such a "macro-step" is a single time step in a discrete time evolution, such as an optimal control problem. One of the themes in this book is the importance of applying AD in a structured way (for efficiency), which can be regarded as a natural generalization of such an approach. Pantoja's construction of the exact Newton step leads to recurrence equations [83] very similar to those introduced in Chapter 3. Another development by Gower and Mello [54] is to use the technique of edge-pushing, which involves the exploitation of internal, implicit structural sparsity to construct the complete derivative matrix. Edge-pushing also exploits the sparsity of the working set [98] to reinforce the connection with checkpointing.

Moreover, in many applications, the derivative matrix is sparse or structured. An efficient approach for computing the derivative matrix through AD is to explore the sparsity and combine AD with a graph coloring technique. Coleman et al. [31, 32] proposed a bi-coloring method to compute a sparse derivative matrix efficiently, based on the for-

ward and reverse modes in AD. Recently, some acyclic and star coloring methods were proposed [49, 50] to efficiently compute the second-order derivative matrix with AD. A good survey of the derivative matrix coloring methods can be found in [51]. Various sparsity pattern determination techniques and a package for derivative matrices were also developed in the AD framework [64, 97, 98]. Readers are encouraged to visit the website for the AD community at www.siam.org/books/se27. ADMAT is a toolbox designed to help readers implement the AD concepts and compute first and second derivatives and related structures efficiently, accurately, and automatically with the templates introduced in this book. This toolbox employs many sophisticated techniques, exploiting sparsity and structure, to help readers gain efficiency in the calculation of derivative structures (e.g., gradients, Jacobians, and Hessians).

## Synopsis of This Book

The focus of this book is on how to use AD to efficiently solve real problems (especially multidimensional zero-finding and optimization) in the MATLAB environment. A matrix-friendly point of view is developed, and with this view we illustrate how to exploit structure and reveal hidden sparsity to gain efficiency in the application of AD. MATLAB templates are provided to help the user take advantage of structure to increase AD efficiency. The focus of earlier AD reference books, e.g., [60, 79], is somewhat different with more attention on computer science issues and viewpoints such as checkpointing, sparsity exploitation, tensor computing, operator overloading, and compiler development.

This book is concerned with the determination of the first and second derivatives in the context of solving scientific computing problems in MATLAB. We emphasize optimization and solutions to nonlinear systems. All examples are illustrated with the use of ADMAT [26].

The remainder of this book is organized as follows. Chapter 1 introduces some basic notions underpinning automatic differentiation methodologies and illustrates straightforward examples with the use of ADMAT. The emphasis in this chapter is on simplicity and ease of use: efficiency matters will be deferred to subsequent chapters. In Chapter 2 we illustrate that Jacobian-matrix, and Hessian-matrix products can be determined directly by AD without first determining the Jacobian (Hessian) matrix. Not only is this useful and cost effective in its own right, but this small generalization to basic AD also opens the window to the efficient determination of sparse derivative matrices (Jacobians and Hessians). Sparse techniques are subsequently discussed in Chapter 2.

While many problems in scientific computing are sparse, i.e., exhibiting sparse Jacobian and/or Hessian matrices, there are equally many problems that are dense but with underlying structure. Chapter 4 develops this notion of structure and illustrates a surprising but powerful result: efficient sparse techniques for the AD determination of Jacobians (Hessians) can be effectively applied to dense but structured problems. This is a far-reaching observation since many applications exhibit structure, and the ability to use sparse AD techniques on dense problems yields significant speed benefits. Chapter 5 also discusses this structure theme. It begins with the observation that the important multidimensional Newton step can be computed with AD technology without first (entirely) computing the associated Jacobian (or Hessian) matrix. This is related to this notion of structure and, again, can yield significant computational benefits. Chapter 5 develops this idea in

some generality while addressing a basic question in scientific computing: how best to compute the Newton step.

In Chapter 3 we illustrate how to use ADMAT with the MATLAB optimization toolbox, with several examples. Occasionally, MATLAB users include FORTRAN and C codes with the calling MATLAB program: in Chapter 6 we show how ADMAT can "work around" such complications. Chapter 7 deals with structure problems in more detail. Specifically, inverse problems are considered with several examples, including a computational finance problem. Chapter 8 provides a template for using ADMAT on structured problems.

Appendix A discusses installation of ADMAT. In Appendix B, we cover some of the basic mechanisms behind AD.

## Acknowledgments

First, we thank Arun Verma for all his preliminary work and help in developing ADMAT 1.0. In addition, we very much appreciate his advice on our development of ADMAT 2.0.

We also thank our colleagues at the University of Waterloo, Cornell University, and Tongji University for their help and advice. We are particularly grateful to our many graduate students who worked with early versions. Six of our former students specifically helped with this book and tested all the enclosed examples. They are Xi Chen, Yuehuan Chen, Wanqi Li, Yichen Zhang, and Hanxing Zhang (all at the University of Waterloo) and Ling Lu at Tongji University. Special thanks to Wanqi Li for his significant help with the design and testing of the template for structured problems described in Chapter 9.

The research behind this book has been supported over the years by the U.S. Department of Energy and the Canadian Natural Sciences and Engineering Research Council. The first author is particularly grateful to Mike and Ophelia Lazaridis for their research support.