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.