Introduction

0.1 Getting Started

This book represents our approach and prejudices toward writing computer software for numerical computing and applications. Both of us have served as editors of the Collected Algorithms (or CALGO) from the ACM [2]. This experience and our personal application development and programming histories have convinced us that Fortran is currently the best computer language for numerical software. We present our views and practices primarily in the form of a tapestry of meaningful examples along with exercises for the reader to work through. We do this because writing or modifying an algorithm or application may involve several elements of the Fortran language, and, while these elements are individually defined in the standard sources, they tend to be organized from a language standpoint, and therefore they appear sprinkled throughout the text.

Constructing software often involves combining ideas with a variety of language constructs. Integrating these ideas, checking for conformance to the current standard, and recalling the syntax of the language elements all place significant research demands on the programmer. We emphasize our preferences for combining these constructs to accomplish the goals of performance and maintainability, as well as the righting of out-of-date practices.

Other aspects of software development are cost and the need to save both time and money. Toward this end, if an existing C function or application can provide some required functionality, then we should consider using it rather than implementing an equivalent Fortran code. We provide examples of integrating C functions into Fortran software even when these functions are available only as precompiled libraries. The same tactic may be used by C programmers when a useful component is available in Fortran by making an interlanguage call from C to Fortran.

In the past, doing this was highly dependent on the C and Fortran compilers being used and was, thus, a high maintenance strategy. The Fortran 2003 standard specifies a mechanism for performing this interoperability between consenting compilers, thus making it an effective option.

In this book we present a set of problems and their solutions that we believe will be of common interest to both Fortran programmers and those who must maintain or interface to existing Fortran software.
We have tried to anticipate many of the common questions that programmers face as they write codes that use the algorithms of numerical analysis and scientific computing. In our choice of illustrative examples we emphasize the new features of the language that help with the writing and maintenance of this type of software. We will dispel the common myth that Fortran is an out-of-date language that does not conform to modern programming requirements. Fortran is no longer your father’s or grandfather’s language; rather, it is a modern and vibrant language consistently being updated with new standards that respond to the needs of new computing hardware. In addition, a significant effort has been made with each new standard to ensure that existing standard-complying codes continue to compile and execute correctly.

The spelling of the “FORTRAN” language name used all CAPS prior to the introduction of the Fortran 90 standard [47] when it was agreed to use the name “Fortran.”

Most of the chapters in this book make reference to a complete program that illustrates the practical use of features of the newer Fortran standards. For space reasons, within the book we have generally avoided including complete listings. Essentially all the software has been made available for downloading from the SIAM web site, www.siam.org/books/ot134, and details are given in Chapter 20. We encourage you to study and modify these codes in order to reinforce your understanding of the material we are presenting. We also urge you to attempt the exercises provided within each chapter, many of which require compilation and execution of the software.

We begin chapters with a Synopsis section giving the main features covered so that you may quickly ascertain the content without further reading.

All the code we have provided within the book is copyrighted by SIAM, but there is no restriction on the use of the software for any purpose. The only requirement is that, if use is made of our codes, our book and SIAM are referenced in any derived work.

0.2 Using a Reference Book

Throughout the book we frequently refer to two texts that describe the Fortran 2003 language: Metcalf, Reid, and Cohen [65] and Adams et al. [4]. We strongly recommend that you have access to one of these while reading our book. Finally, there is the full standard, ISO/IEC [48], which is the authoritative reference but is no easy read for the humble coder! While a comprehensive discussion of the language elements can be found in these references, our goal is to illustrate the use of the Fortran 2003 language with helpful and nontrivial examples.

0.3 Become Familiar with the New

If you are acquainted with Fortran only up to the Fortran 77 standard [46], you will need to gain some understanding of the basic new features that were introduced with Fortran 90 [47]. These include the formatting of source code, use of comments,
and new system functions, along with a number of the newer constructs, including derived types, elemental functions, and the use of modules. We discuss these further in Chapter 1. As a guide for study, the top ten features added to Fortran since the 1977 standard are, in our prejudiced order,

1. The Fortran Module
2. The Derived Type and Classes
3. Allocatable Objects
4. Optional Arguments
5. IEEE Exception Handling
6. Interoperability with C
7. Support for Recursion
8. Array Operations
9. Machine Constants
10. Generic Typing of Routine Names

0.4 Moving to Interesting Topics

Every chapter in this book was included for a reason. What follows is a “road map” from topics to their discussion in the chapters that follow. Reasons for including these chapters come from our experiences as editors and our work in application development.

0.4.1 Leading to QUADPACK

One aim is to present Fortran software technology in the role of interfacing numerical codes with user routines. The demands of modern hardware and application requirements give compelling reasons for using extensible derived types—designated as class arguments—passed to user routines. A meaningful example is developed in Chapter 7 for numerical quadrature, a modernized version of QUADPACK, described in Piessens et al. [72].

This example illustrates an essential design element of Fortran numerical software that is missing from many popular codes, and getting comfortable with it requires study. Here are guidelines to topics that we believe will help lead to such an understanding.

Qpack: new source formats, logical operators, comments, and optional arguments; see Chapter 1.

Qpack: derived types, module organization; see Chapter 2.
Qpack: generic names; see Chapter 3; type extension, classes, information passing; see Chapter 5.

Qpack: recursion for multiple integrals; see Chapter 6.

Qpack: using parallelism; coarrays and OpenMP; see Chapters 14 and 15.

Qpack: the QUADPACK Fortran source; see Chapter 20.

Qpack: documenting the use of the revised QUADPACK; see Chapter 8.

0.4.2 Leading to SuperLU

The use of SuperLU [see 59, 58] for solving sparse matrix systems of linear equations may often be a critical part of a Fortran application code. However, many developers may shun the use of this function library because of the integration and maintenance problems perceived in integrating their application code with software only available in C. With older versions of the Fortran standard this would have involved a compiler-dependent solution, but the introduction of interlanguage calls in the newer standards alleviates many of these problems. Furthermore, we do not need to make any changes to the source code of the C application library to allow Fortran access to the underlying functions; indeed, we may only have a compiled version of the library available. To allow interlanguage calls we only need to study the calling sequences of the C functions; the objective then is to make things simple for the Fortran programmer while still retaining flexibility.

There are other preliminary algorithmic steps that are part of sparse matrix linear algebra: building the data structures, forming transposed matrices, computing matrix-vector or matrix-matrix products, adding sparse matrices, accumulating entries, and preprocessing a matrix to ensure that every row or column contains at least one nonzero entry. We show how to implement these by using defined operators that make interlanguage calls to existing C functions. Finally, we provide two practical applications, a data fitting problem and a linear boundary value problem for Airy’s equation, to illustrate how all these components act together.

SupLU: building data structures, accumulating entries, user-defined operations, addition of sparse matrices; see Chapter 4.

SupLU: interlanguage calls to C; see Chapter 10; sparse solves with user-defined operations; see Chapter 11.

SupLU: sparse matrix defined operations in sample applications; see Chapter 12.

0.4.3 IEEE Arithmetic Exceptions

In many applications there are situations where a computation may encounter an exceptional operation. This may be a divide by zero, an overflow, a significant underflow, or an operation with a NaN—an unspecified floating-point value. Modern software will typically include code to protect against program crashes or the wasteful discarding of results by trapping such conditions and taking appropriate action.
However, even a relatively small amount of testing for exceptional arithmetic operations will almost invariably lead to an inefficient use of the floating-point processor. If such events are rare, an alternative strategy may be to try a simple approach and use the Fortran standard IEEE intrinsic modules to detect any arithmetic problems. Only if the simple approach has an exception do we resort to a more careful but less efficient means of performing the calculations. If this method also fails, we may need to bow out gracefully after printing an explanatory error message.

**IEEE**: computing the $l_2$ norm of a vector, BLAS function \texttt{DNRM2} [56]; see Chapter 9.

**IEEE**: managing exceptions during an OpenMP parallel construct; see Chapter 15.

### 0.4.4 Quicksort and Recursion

Conventional thinking regards the use of recursion as inefficient, but this is certainly not the case with large scale data sorting. Using the Quicksort algorithm (see [85, Chapter 7]) we illustrate how the software development is best presented using recursion. The practical requirements of sorting are varied, for example, sorting data of different precisions or data types; sorting data into increasing or decreasing order; sorting a subset of the data or returning a permutation vector rather than sorted data. We illustrate how, by using specialized derived types, we may produce very general purpose software.

**QuickS**: recursion, type-bound procedures, abstract data types and abstract interfaces; see Chapter 6.

### 0.4.5 Generic Names of Subprograms

Numerical software is easier to use if the routine names are easily recalled. For example, since the introduction of Fortran 77, users have been able to use intrinsic functions like $\texttt{SIN(x)}$ to compute the expected result for $x$ of real or complex type, regardless of precision. Later standards extended this to return an array of values when the argument is an array. Such functions are termed *generic*, and modern Fortran also allows users to define their own generic subprograms. This is a welcome design feature and we illustrate its use by constructing an interface to some existing software for Airy functions [26].

**Gener**: generic naming, real and complex arguments, arguments that are arrays; see Chapter 3.

### 0.4.6 MPI, Coarrays, and OpenMP

MPI, or the Message Passing Interface, is used in applications where the size of the problems and/or the execution times requires the use of more than one process executing the same program simultaneously. These parallel running executables typically interact with one another by communicating partial results (message passing).
Fortran 2008 introduced an alternative approach using coarrays which extend Fortran to provide a more natural way for executables to interchange data. This approach leads to many of the calls used by MPI to send and receive data from one process to another being performed via an assignment operator. A good introduction to coarrays may be found in Reid and Numrich [79].

We present examples that perform the same computations using both MPI and coarrays.

OpenMP, or Open Multiprocessing, describes a technology that enables each executable to use its internal CPUs, or “cores,” for improved performance.

These software methodologies require study and practice in order to understand and use them successfully in applications. Our coverage of MPI and OpenMP is brief and resources other than just a Fortran compiler may be required to run the sample codes. We do not discuss in detail the use of MPI in numerical software.

Excellent tutorials on MPI and OpenMP are available (see, e.g., Barney [10, 11]).

ParL: a standard-conforming interface to MPI, examples to compute $\pi$ and compute parallel matrix-vector products; see Chapter 13.

ParL: use of coarrays, examples of duplicate MPI computation of $\pi$ and matrix-vector products; see Chapter 14.

ParL: some guidelines and tips for using OpenMP; see Chapter 15.

0.4.7 Getting the Bugs Out

A developer of numerical software will often spend more time removing bugs or achieving accuracy and performance than writing the source code in the first place. We offer some topics that may help to improve productivity in this area. But the human capacity for error and unfortunate choice is unlimited! There is no fix for this fact of life.

Obsol: identify and remove obsolescent features from source code; see Chapter 16.

Tests: what testing is, types of testing, constructing test data; see Chapter 17.

CompL: using compiler options for optimization, run-time checks, clutter removal; see Chapter 18.

Times: portable timing for CPU use, elapsed time; see Chapter 1.

Tools: using make, version control with SVN, documentation systems, debuggers, profilers, pretty printers, sample output and error processors; see Chapter 19.
0.4. Moving to Interesting Topics

0.4.8 Getting to the Source (Code)

A primary intent of this book is to provide program units for use by the reader that illustrate the lessons we have learned. To use this material a developer can turn directly to Chapter 20. There one can find the URL to the SIAM web site that has these files available for download. The subdirectories for the code suites are listed in Chapter 20 with a longer description of their contents.

- **Chapter02** Packaging for collections of Fortran 77 codes.
- **Chapter03** Generic packaging of Airy functions.
- **Chapter04** Defined operations and overloaded assignment for sparse matrix operations.
- **Chapter05** Class objects, interior to a numerical code.
- **Chapter06** A framework and packaging for the Quicksort sorting algorithm.
- **Chapter07** Modernizing QUADPACK routine *qag* to use Fortran 2003.
- **Chapter08** Documenting the use of the modernized QUADPACK routines.
- **Chapter09** Computing the *l*₂ norm of a vector using the IEEE intrinsic modules for handling exceptions.
- **Chapter10** Demonstrating standardized interlanguage calls between Fortran and C.
- **Chapter11** Defined operations and overloaded assignment for solving square, sparse matrix linear systems with SuperLU.
- **Chapter12** Defined operations developed for sparse matrices, illustrated with two applications.
- **Chapter13** Direct interface to a core suite of the C library of MPI routines, with two examples.
- **Chapter14** The two examples using MPI are repeated using coarray Fortran.
- **Chapter15** Program units that illustrate usage of OpenMP directives.
- **Chapter16** Before (Fortran 77) and after (Fortran 90) versions of a simple Newton method program.