Preface

Programming in *Mathematica*

In its brief history, the world of programming has undergone a remarkable evolution. Those of us old enough to remember boxes of punch cards and batch jobs couldn't be happier about some of these changes. One could argue that the limitations, physical and conceptual, of the early programming environments helped to focus that world in a very singular manner. Eventually, efforts to overcome those limitations led to a very visible and broad transformation of the world of computer programming. We now have a plethora of languages, paradigms, and environments to choose from. At times this embarrassment of riches can be a bit overwhelming, but I think most would agree that we are fortunate to have such variety in programming languages with which to do our work.

I learned about *Mathematica* as I suspect many people have – after using several languages over the years, a colleague introduced me to a new and very different tool, *Mathematica*. I soon realized that it was going to help me in my work in ways that previous languages could not. Perhaps the most notable feature was how quickly I could translate the statement of a problem to a working program. This was no doubt due to having a functional style of programming at my fingertips but also being able to think in terms of rules and patterns seemed to fit well with my background in mathematics.

Well, *Mathematica* is no longer a young up-start in the programming world. It has been around now for over 25 years, making it, if not an elder statesman, certainly a mature and familiar player. And one that is used by people in fields as varied as linguistics, bioinformatics, engineering, and information theory. Like myself, many people are first introduced to it in an academic setting. Many more are introduced through a colleague at work. Still others have seen it mentioned in various media and are curious as to what it is all about. After using it to do basic or more advanced computation, most users soon find the need to extend the default set of tools that come with *Mathematica*. Programming is the ticket.

So what makes *Mathematica* such a useful programming tool? First, it is a well-designed language, one whose internal logic will be quite apparent as you get to know it. It is not only easier to use but it also provides an enjoyable programming experience. Second, it is a multi-paradigmatic language, meaning several different styles of programming are available under one roof: functional programming (like LISP or HASKELL), procedural programming (like C, FORTRAN, JAVA, PERL), logic programming (like PROLOG), rule-based programming (SNOBOL and some of PROLOG), and it has a rich

string pattern language, including support for regular expressions (like PERL). In no other language can you program in so many styles and mix them at will.

The generality of the *Mathematica* language is in sharp contrast to what are called *domain-specific languages*. As the name implies, these are languages designed to solve problems for a specific application domain. HTML is a classic example – it is a markup language that is only really useful for its intended purpose, marking up web pages. Similarly, TEX, used for page composition of technical material and GRAPHML, for representing, formatting, and operating on graph objects, can both be thought of as domain-specific languages.

Mathematica takes a different approach. The language is general enough that it can be used in a variety of different disciplines to represent and solve computational problems. As for speed, it is fast enough for many problems you will encounter. When the need to increase performance arises, options are available: parallel programming, compilation, connecting to external programs.

Given the generality of *Mathematica* and the fact that you can choose different programming paradigms with which to write your programs, the novice is often left feeling a little bewildered. Which style should you use for a particular problem? Which is fastest? Which approach makes the most sense for your particular domain? In general, although good *Mathematica* programmers rely on a functional style to a great degree, your *Mathematica* programs will contain a combination of styles. They tend to be more compact, easier to read, and easier to debug (although not always) than more traditional procedural implementations. Well, perhaps I am showing my bias. It may be that I chose *Mathematica* because it fit well with how I think; or perhaps my adoption of *Mathematica* has shaped the form of programs I write. Perhaps this is just another instance of the Sapir–Whorf hypothesis in linguistics, which states that a spoken language influences the patterns of thought of the speaker of that language. Rather than go down that rabbit hole, let's just say that you will program differently with *Mathematica*, and, I think, more efficiently and more enjoyably.

Who is this book for?

This book is designed for two overlapping audiences. First, it is intended as a self-contained, self-study book for anyone who wants to learn how to use the *Mathematica* language to solve problems in their domain. Typically, this includes researchers, academics, students, and even hobbyists who are accustomed to picking up and learning about any tools that will help them do what they do better. Comments from users of my previous books have persuaded me that an example-based book, such as this, is appreciated by many.

Second, this book can also be used as a text in a course on *Mathematica* programming such as is used in many schools and universities as part of, or in preparation for, a computational science course. Toward that end, each section includes many exercises to check basic understanding of the concepts in that section, as well as providing extended and (hopefully) interesting examples in their own right. A complete set of solutions in both PDF format and as *Mathematica* notebooks is available at the publisher's website www.cambridge.org/wellin_essentials.

The examples and exercises in this book draw from many different fields, including:

- *Bioinformatics*. Analysis of nucleotide sequences, computing GC ratios, displaying blocks of genetic information, searching for subsequences, protein–protein interaction networks, dot plots, displaying amino acid sequences, space-filling plots.
- *Computer science*. Encoding/encryption, sorting, adjacency structures, Collatz sequences, Hamming numbers, Hamming weight, Tower of Hanoi, Fibonacci numbers, Fibonacci words, Euler numbers, root finders, Horner polynomial representation, inverse permutations, random number algorithms, sieving, associative arrays, Monte Carlo simulations, comparisons with other languages.
- *Data and signal analysis*. Filtering signals (smoothing, clipping, removing spikes), entropy of signals, Benford's law, stem plots, lag plots, statistical analysis of data, random walks, visualizing extent of data, Hamming distance, cluster analysis.
- *Finance and economics*. Time-series analysis of economic and financial data, trend plots, stock screens.
- *Geometry*. Convex hull, diameter of point sets, point-in-polygon problems, traveling salesman-type problems, hypocycloids and epicycloids, area and perimeter problems, boundaries of regions, Heron's formula, triangle medians and centers.
- *Graphs and networks*. Random graphs, regular graphs, bond percolation, connected components, dense graphs, directed acyclic graphs, neighborhood graphs, random walk on graphs.
- *Image processing*. Resizing, filtering, segmentation.
- *Mathematics*. Palindromic numbers, triangular numbers, pyramidal numbers, truth tables, prime gaps, Vandermonde and Jacobian matrices, Mersenne numbers, Venn diagrams, geometric transformations.
- *Solar physics and atmospheric science*. Analyzing and visualizing sunspot activity using time series, analyzing global sea-surface temperature data, solar magnetic field cycles.
- *Textual analysis and natural language processing*. Corpus linguistics, word stemming, stop words, comparative textual analysis, scraping websites for data, sorting strings, bigrams and *n*-grams, word games (anagrams, blanagrams, palindromes), filtering text.

The background necessary to successfully navigate and learn from this book includes some introductory undergraduate mathematics (basic linear algebra, geometry). The science topics and problems introduced in this book are mostly self-contained and, where necessary, they point to available references in the literature. Although no prior experience with programming is assumed, even the most basic experience with writing, running, or debugging programs will be helpful in getting you up and running more quickly.

This book is not a tutorial introduction to *Mathematica*. For that, fortunately, there are many useful resources available, including in-product tutorials, online courses, and many good books. Refer to the Wolfram Research website for a comprehensive listing of these materials.

How to use this book

For those new to *Mathematica*, Chapter 1 will give you a sense of what it means to program, what it means to program in *Mathematica*, and introduces you to some of the basic tools every *Mathematica* user should know about: entering and evaluating expressions, dealing with errors, getting help.

The real meat of the book starts with Chapter 2, which provides a semi-formal description of the *Mathematica* language with a particular focus on expressions. Chapters 3 through 5 (lists, patterns and rules, functions) are the core of the book. Lists are a fundamental data type in *Mathematica* and provide a relatively simple structure for much of what follows. Pattern matching and the use of transformation rules are an essential and somewhat unique aspect that every *Mathematica* programmer must master. Functional programming, introduced in Chapter 5, will likely be new to you, but once you understand it, the speed and efficiency of what you can do will increase dramatically.

Chapter 6 introduces localization constructs, options and messages enabling your programs to look and behave much like the built-in functions. Although you could skip this chapter if you were solely focused on solving particular computational problems, spending some time with these topics will help you to start thinking like a programmer. How will your programs be used by others? How will they likely interact with your programs? What should happen when something goes wrong?

Strings are used of course in linguistics, but they also have broad applicability in computer science, web search, and bioinformatics. They are discussed in Chapter 7, which includes an introduction to regular expressions. If you do little or no work with strings you could safely skim or skip this chapter but there is a lot of useful material there if you do dive in.

Chapter 8 on graphics and visualization gives an introduction to the *Mathematica* graphics language as well as the dynamic expression language as it relates to graphics. Topics on optimizing your graphics code are included.

With all the different approaches available in *Mathematica* to solve problems, it is sometimes difficult to determine the "best" approach. Chapter 9 on program optimization addresses this issue from the point of view of speed and efficiency; that is, which approaches will help your programs run faster and use fewer system resources.

Chapter 10 provides a framework for those who wish to turn their programs into packages that can be shared with colleagues, students, or clients. It includes an extended example, random walks, so that you can begin to see examples of larger programs that include modularization, namespace constructs, messaging, options, testing, and other meta-programming issues that are important to think about as your programming skills advance.

To return to the question about which style of programming you should adopt for a particular problem, I hope that this book will convince you that this is the wrong question to ask. As you become more familiar with the *Mathematica* programming language, you will start to reword and formulate your programs and implement them using all of the tools you have under your belt.

Learning different paradigms does not point to an either/or approach, but allows you to use combinations of programming styles to solve your problems as efficiently as possible.

With that in mind, do not take the formal structure of this book too literally. Although books are pretty linear physical objects – one page or chapter follows another – learning is not linear. It is fine to jump around as the need arises. In fact, you are encouraged to do so. Although, in general, latter material presupposes knowledge of what has come before, several examples and exercises use concepts that are developed later in the book and if you are not already familiar with them, you should feel free to jump ahead.

Numerous references are made throughout the book to tutorials and other help documents that are found in *Mathematica*'s help system, the Wolfram Language and System Documentation Center, referred to as WLDC. This comprehensive help system is available in *Mathematica* under the Help menu and online at reference.wolfram.com.

The exercises (over 350 of them) are an integral part of this book. Some test your knowledge of basic concepts in each section; some extend those concepts; and some use those concepts to solve more substantial and interesting problems. "Life is not a spectator sport" and neither is programming. Try to do as many of the exercises as you can. There is much to learn there.

Resource limitations have prevented the inclusion of the solutions to the exercises with this book (printed, they would stretch over 240 pages). Fortunately, the solutions to the exercises are available online at the publisher's website (www.cambridge.org/wellin_essentials) and the author's website (www.programmingmathematica.com) in both *Mathematica* notebook format as well as PDF. These sites also contain most of the programs and packages developed in this book, the book's bibliography with live clickable references, lists of errata, and some extra material that did not make it into the printed book.

Numerous data files are used throughout this book, many imported from the web and some residing in a project directory for this book. To set up your environment so that import works seamlessly, download the data files from the author's or publisher's website as indicated in the previous paragraph. Then, wherever you put the directory of data files, use something like the following to add that directory to your path. You may find it convenient to put this command in a Kernel/init.m file so that it is evaluated each time you start a new session.

```
PrependTo[$Path,
  ToFileName[{$UserBaseDirectory, "Applications", "EPM", "Data"}]];
```

Finally, a word about *Mathematica* itself. As this book was being developed, a bifurcation occurred and where there was one name, *Mathematica*, now there are two: the *Wolfram Language* and *Mathematica*. The *Wolfram Language* is the name of the underlying language and *Mathematica* is the wider product in which that language occurs. Since I have used both the language and the product throughout the book, in an attempt to simplify, I have opted to refer to the software simply by using the broader term *Mathematica* here.

Acknowledgments

I have been using *Mathematica* for over 25 years now, but I still run into plenty of puzzling things that I would not be able to disentangle without the assistance of others. Thanks are due to Rob Knapp for help in understanding some of the internals of the looping constructs and gaining a deeper understanding of compiling functions. Dan Lichtblau provided some insights into the internals of the Fibonacci computations. Ulises Cervantes-Pimentel helped with region and mesh functionality as well as the structure of the internal representations of these objects. Harry Calkins answered numerous questions on compiling and parallel computation. Charles Pooh provided answers to several of my questions regarding graph functionality and internals. Also, Andre Kuzniarek, Glenn Scholebo, Jeremiah Cunningham, and Larry Adelston helped with a number of production and front end issues. Dennis Coleman provided assistance with some graphic design.

Thanks are also due to Frank E. Blokland, of the Dutch Type Library, for help in working around some OpenType font issues. Frank is the typographer responsible for creating the digital version of the text font used in this book, Albertina (the original design was due to Chris Brand).

David Tranah, my editor at Cambridge University Press, has been extremely helpful in finding the right mix of topics and presentation for such a book. Robert Judkins, in the production department at Cambridge was as efficient and professional as any author could wish. Thanks to Clare Dennison at Cambridge who has quickly and pleasantly handled all those extra little administrative tasks associated with such a project. Thanks are also due to the reviewers of this book, who provided some focused and insightful comments and suggestions on content and style.

The one person who knows better than anyone what it is like to undertake and complete such a project is my wife Sheri. It may have been doable without her patience, support, and love, but it would not have been as enjoyable.

Paul R. Wellin
programmingmathematica@gmail.com