The Mathematical-Function Computation Handbook

Nelson H.F. Beebe

The Mathematical-Function Computation Handbook

Programming Using the MathCW Portable Software Library

Springer

Nelson H.F. Beebe
Department of Mathematics
University of Utah
Salt Lake City, UT
USA

ISBN 978-3-319-64109-6 ISBN 978-3-319-64110-2 (eBook)
DOI 10.1007/978-3-319-64110-2

Library of Congress Control Number: 2017947446

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Dedication

This book and its software are dedicated to three Williams: Cody, Kahan, and Waite. They taught us that floating-point arithmetic is interesting, intricate, and worth doing right, and showed us how to do it better.

This book and its software are also dedicated to the DEC PDP-10 computer, on which outstanding interactive computing and the Internet were built, and on which this author spent a dozen productive years.

Preface

WRITE YOUR OWN STORY.
DON'T LET OTHERS WRITE IT FOR YOU.

— CHINESE FORTUNE-COOKIE ADVICE.

A *preface* IS GENERALLY SHORTER THAN AN *introduction*, WHICH CONTAINS MATTER KINDRED IN SUBJECT, AND ADDITIONAL OR LEADING UP TO WHAT FOLLOWS; WHILE A *preface* IS USUALLY CONFINED TO PARTICULARS RELATING TO THE ORIGIN, HISTORY, SCOPE, OR AIM OF THE WORK TO WHICH IT IS PREFIXED.

— *New Century Dictionary* (1914).

This book documents a large library that supplies the mathematical functions required by several programming languages, including at least these:

- the 1983 ANSI and 1995 and 2012 ISO Standards [Ada83, Ada95, Ada12] for the Ada programming language;
- the 1990, 1999, and 2011 ISO Standards for C [C90, C99, C11b];
- the 1998, 2003, and 2011 ISO Standards for C++ [C++98, BSI03b, C++03a, C++11a];
- the 2002 and 2006 ECMA and 2006 ISO Standards for C#® (pronounced *C-sharp*) [ECM06a, HWG04, CLI05, C#06a, CLI06];
- the 1978 ANSI and 1991, 1997, and 2010 ISO Standards for Fortran [ANSI78, FTN91, FTN97, FTN10];
- the widely used, but not yet standardized, Java® programming language [AG96, AG98, CLK99, AGH00, GJSB00, GJSB05, GJS+13, GJS+14]; and
- the 1990 ISO Extended Pascal Standard [PAS90, JW91].

Numerous scripting languages, including awk, ECMAScript®, hoc, JavaScript®, Julia, Lua®, Perl®, PHP, Python®, Rexx, Ruby, and Tcl, offer a subset of mathematical functions that are usually drawn from the venerable Fortran and C repertoires. Many other current and historical programming languages, among them Algol 60, Algol 68, COBOL, D, Go, Lisp, Modula, Oberon, OCaml, PL/1, Rust, and Scheme, as well as the Adobe® PostScript® page-description language, also provide subsets of the Fortran functions.

Although the needs of those languages are a valuable guide for the design of this library, this author's view has been a wider one, and several additional elementary and special functions are provided, including ones for the detection of integer overflow, a sadly neglected area in most architectures and almost all programming languages.

Most current computers, as well as the virtual machines for C#, Java, and PostScript, are based on the *IEEE 754 Standard for Binary Floating-Point Arithmetic* published in 1985, but developed several years earlier, and implemented in hardware in the Intel 8087 coprocessor in 1980. Software intended to be run only on current systems could therefore limit itself to the IEEE 754 architecture. However, three considerations led to the broader view adopted in this book:

- Decimal floating-point arithmetic is part of the 2008 revision of the IEEE 754 Standard [IEEE08, ISO11], and there are proposals to include it in future versions of the ISO C and C++ Standards. Two main reasons to provide decimal arithmetic are its familiarity to humans, and its widespread use in databases.

  IBM is a strong proponent of decimal arithmetic, and has produced a firmware implementation on its mainframe systems, and added hardware support in the PowerPC version 6 and later chips. This arithmetic is

based on more than two decades of experience with software decimal arithmetic, notably in the Rexx programming language [Cow85, Cow90] defined by ANSI Standard X3.274-1996 [REXX96], and the NetRexx language [Cow97].

Symbolic-algebra languages, such as Maple®, Mathematica®, Maxima, MuPAD®, PARI/GP, and REDUCE, provide arbitrary-precision arithmetic. The Maple language uses decimal arithmetic, but the others use binary arithmetic.

- There are excellent, and readily available, virtual machines for several historical architectures, including most early microprocessors, the DEC PDP-10, PDP-11, and VAX architectures, the IBM System/360, and even the modern Intel IA-64.

  A recent book on virtual machines [SN05] notes that they provide a good way to prototype new machine designs, and have important advantages for security when users can be isolated in private virtual machines.

  The great success of the Java programming language has in large part been due to its definition in terms of a virtual machine, called the *Java Virtual Machine (JVM)*® [LY97, LY99, SSB01, LYBB13, Set13, LYBB14], and a standard library that provides a uniform environment for Java programs that is independent of the underlying hardware and operating system.

  Microsoft's related C# language is also defined in terms of a virtual machine that forms the core of the Microsoft .NET Framework, and the associated free-software reimplementations of .NET in the DotGNU Project[1] and the Mono Project.[2]

  Once compilers make programming languages available for the virtual machine, applications, and indeed, even entire operating systems, can be lifted from hardware to virtual machines. This broadens markets for software vendors, removes their dependence on hardware suppliers, and allows them to reach future hardware platforms more quickly.

  The commercial Parallels®, VirtualBox®, Virtual Iron®, VMware®, and Xen® systems have demonstrated that virtual machines adapted to particular hardware can support multiple operating systems on a single CPU with little overhead. Hewlett–Packard, IBM, and Sun Microsystems have all introduced products that allow processors to be logically split and shared among multiple operating systems, or multiple instances of the same operating system. GNU/LINUX *containers* and FREEBSD *jails* provide similar capabilities.

- The Cody and Waite *Software Manual for the Elementary Functions* [CW80] that inspired some of the work described in this book was careful to address the algorithmic issues raised by floating-point designs with number bases other than two. Their book also addresses computations of the elementary functions in *fixed-point* arithmetic, a topic that we largely ignore in this book.

Most books and courses on programming languages spend little time on floating-point arithmetic, so student programmers may consequently conclude that floating-point programming must be trivial, uninteresting, and/or unimportant. This author's view is that floating-point programming does not receive the attention that it deserves, and that there are interesting problems in the design of mathematical software that are not encountered in other types of programming.

However, because a substantial portion of the human population seems to have a fear of mathematics, or at least finds it uncomfortable and unfamiliar, this book takes care to minimize the reader's exposure to mathematics. There are many research papers on the elementary and special functions that provide the mathematical detail that is a necessary foundation for computer implementations, but in most cases, we do not need to deal with it directly. As long as you have some familiarity with programming in at least one common language, and you can recall some of your high-school algebra, and are willing to accept the results of a few short excursions into calculus, you should be able to understand this book, and learn much from it.

On the other hand, professional numerical analysts should find this book of interest as well, because it builds on research by the mathematical software community over the last four decades. In some areas, notably those of converting floating-point numbers to whole numbers, and finding remainders, we present cleaner, simpler, and more portable solutions than are found in existing libraries.

[1] See `http://www.gnu.org/projects/dotgnu/`.

[2] See `http://www.mono-project.com/`.

An important recent research topic has been the development for some of the elementary functions of mathematically proven algorithms that guarantee results that are always correctly rounded. In most cases, these advanced algorithms are quite complex, and beyond the scope of this book. However, for some of the simpler elementary functions, we show how to produce always, or almost always, correctly rounded results.

You might wonder why anyone cares about the correctness of the last few digits of a finite approximation to, say, the square root of 27. Certainly, few humans do. However, it is important to remember that computer hardware, and software libraries, are the foundations of all other software. Numerical computing in particular can be remarkably sensitive to the characteristics and quality of the underlying arithmetic system. By improving the reliability and accuracy of software libraries, we strengthen the foundation. By failing to do so, we leave ourselves open to computational disasters.

Acknowledgements

This author has had the interest and opportunity to have programmed in scores of programming languages on dozens of operating systems, and most of the major CPU architectures, since his first encounter with the IBM 7044 and IBM System/360 model 50 as an undergraduate student.

At the University of Utah Department of Mathematics, we have a large computing facility that serves thousands of students, as well as departmental staff and faculty, and invited guests. We have made it a point to provide, and preserve, access to a wide range of physical and virtual hardware, and operating systems, to encourage and facilitate software portability testing. A powerful tool, `build-all`, developed by this author and presented in another book [RB05a], makes it easy to automate the building and validating of software packages in parallel on multiple systems.

In recent years, guest access to computing facilities in other locations has been invaluable for development and testing on even more platforms. Those organizations include:

- the University of Utah Center for High-Performance Computing (formerly, the Utah Supercomputing Institute);
- the Henry Eyring Center for Theoretical Chemistry in the Department of Chemistry at the University of Utah;
- the University of Utah Department of Electrical Engineering;
- the University of Utah Department of Physics and Astronomy;
- the Emulab Network Emulation Testbed of the Flux Research Group in the School of Computing at the University of Utah; .
- the Hewlett–Packard Test Drive Laboratory;
- the IBM Linux Community Development System; and
- the University of Minnesota Supercomputing Institute.

This author is grateful to the management and staff of all of those excellent facilities, as well as to Intel Corporation for the grant to his department of an IA-64 server.

The mailing list of the IEEE 754 committee, and numerous exchanges with list members, continue to stimulate my interest in floating-point arithmetic. In particular, IBM Fellow Mike Cowlishaw provided useful comments on some of the material on decimal arithmetic in this book, and helped to clarify the history and rationale of decimal arithmetic in computers, and its implementation in modern compilers.

Virtual machines used during the development of this book and its software include:

- the fine commercial VMware system on AMD64, EM64T, and IA-32;
- Ken Harrenstien's outstanding KLH10 implementation of the influential and venerable PDP-10 architecture on which this author worked happily and productively for a dozen years, and the far-ahead-of-its-time DEC TOPS-20 operating system kindly made available to hobbyists by its original vendor;
- Bob Supnik's amazing `SIMH` simulator of about thirty historical architectures, including the Interdata 8/32 (target of the first port of UNIX from the DEC PDP-11), and the DEC VAX;

- Roger Bowler's `Hercules` emulator for the IBM System/370, ESA/390, and z/Architecture; and
- The QEMU (Quick EMUlator) hypervisor and KVM (Kernel-based Virtual Machine) that run on most desktop operating systems, including at least Apple MAC OS X and MACOS, DRAGONFLYBSD, FREEBSD, GHOSTBSD, GNU/LINUX, HAIKU, HARDENEDBSD, MIDNIGHTBSD, Microsoft WINDOWS, NETBSD, OPENBSD, PAC BSD, PCBSD, OPENSOLARIS, REACTOS, and TRUEOS.

This book, and all of the software that it describes, is a single-person production by this author. Years of following the scientific research literature, and published books, in broad areas of computer science, theoretical chemistry, computational physics, and numerical mathematics, and a lifetime spent in programming in scientific computing and other areas, gave the experience needed to tackle a project with the breadth of scope of the software library described here.

The Unix family

The Open Group currently owns the all-caps registered trademark UNIX® for a descendant of the operating system developed at AT&T Bell Laboratories starting about 1969, and permits vendors whose implementations pass an extensive test suite to license use of that trademark. Because of legal wrangling, ownership of the specific name UNIX was long disputed in US courts. That led computer manufacturers to rebrand their customized versions under other trademarked names, and similarly, in the free-software world, each O/S distribution seems to acquire its own unique name. In this book, we use the capitalized name UNIX to refer to the entire family.

The GNU® system, where *GNU* stands for the infinitely recursive phrase *GNU is Not Unix*, is, for the purposes of this book, also a UNIX-like system. If this matters to you, just mentally add the suffix *-like* to every mention of UNIX in this book.

Filesystem implementations, operating-system kernel details, software packaging practices, and system-management procedures and tools, differ, sometimes dramatically so, across different members of the UNIX family. However, for the ordinary *user* of those systems, they are all familiar and similar, because most of their commonly used commands are nearly identical in behavior and name. The notions of files as byte streams, devices, kernel data, and networks treated as files, command shells for interactive use and scripting, simple I/O redirection mechanisms, and pipes for easy connection of many smaller programs into larger, and more powerful, ones, are critical features of the UNIX environment. A common window system, X11, provides mechanism but not policy, and separates program, window display, and window management, allowing each to run on separate computers if desired, but joined by secure encrypted communications channels.

Many UNIX distributions have long been available as open-source software, allowing programmers all over the world to contribute documentation, enhancements, and sometimes, radical new ideas. Internet mailing lists allow them to keep in regular contact and develop long-time electronic friendships, even though they may never have a chance to meet in person. The UNIX world is truly a global community.

Trademarks, copyrights, and property ownership

Because it deals with real computers, operating systems, and programming languages, both historic and current, this book necessarily contains many references to names that are copyrighted, registered, or trademarked, and owned by various firms, foundations, organizations, and people. Except in this preface, we do not clutter the book text with the traditional superscript symbols that mark such ownership, but we acknowledge it here, and we index every reference to model, product, and vendor names. Among the commercial software systems most frequently mentioned in this book are Maple, Mathematica, MATLAB®, and MuPAD. In general, we should expect almost any commercial entity, or commercial product, to have a name that is registered or trademarked. The computing industry that has so changed human history is the work of many companies and many people from many countries, and we should view it proudly as part of our shared modern heritage.

The International Organization for Standardization (ISO) kindly gave permission to cite in this book (usually brief) snippets of portions of the ISO Standards for the C language. Because of the language precision in Standards, it is essential for correct software development to be guided by the original exact wording, rather than working from existing practice, hearsay, imperfect human memory, or paraphrased text.

To show code, or not

Authors of books about software have to make a choice of whether to show actual code in a practical programming language, or only to provide imprecise descriptions in flowcharts or pseudocode.

Although the latter approach seems to be more common, it has the serious drawback that the code cannot be tested by a compiler or a computer, and long experience says that untested code is certain to contain bugs, omissions, and pitfalls. Writing software is hard, writing floating-point software is harder yet, and writing such software to work correctly on a broad range of systems is even more difficult. Software often outlives hardware, so portability is important.

There are many instances in this book where subtle issues arise that *must* be properly handled in software, and the only way to do so is to use a real programming language where the code can undergo extensive testing. The floating-point programmer must have broad experience with many architectures, because obscure architectural assumptions and platform dependencies can all too easily riddle code, making it much less useful than it could be, if more care were paid to its design.

This author has therefore chosen to show actual code in many sections of this book, and to index it thoroughly, but not to show all of the code, or even all of the commentary that is part of it. Indeed, a simple printed listing of the code in a readable type size is several times longer than this book.

When the code is routine, as it is for much of the validation testing and interfacing to other programming languages, there is little point in exhibiting it in detail. All of the code is freely available for inspection, use, and modification by the reader anyway, because it is easily accessible online. However, for many shorter routines, and also for complicated algorithms, it is instructive to display the source code, and describe it in prose.

As in most human activities, programmers learn best by hands-on coding, but they can often learn as much, or more, by reading well-written code produced by others. It is this author's view that students of programming can learn a lot about the subject by working through a book like this, with its coverage of a good portion of the run-time library requirements of one of the most widely used programming languages. This is *real* code intended for *real* work, for portable and reliable operation, and for influencing the future development of programming languages to support more numeric data types, higher precision, more functionality, and increased dependability and security.

To cite references, or not

Although research articles are generally expected to contain copious citations of earlier work, textbooks below the level of graduate research may be devoid of references. In this book, we take an intermediate approach: citations, when given, are a guide to further study that the reader may find helpful and interesting. When the citations are to research articles or reports, the bibliography entries contain Web addresses for documents that could be located in online archives at the time of writing this book.

Until the advent of computers, tracking research publications was difficult, tedious, and time consuming. Access to new journal issues often required physical visits to libraries. Photocopiers made it easier to abandon the long-time practice of sending postcards to authors with requests for article reprints. The world-wide Internet has changed that, and many journals, and some books, now appear only electronically. Internet search engines often make it possible to find material of interest, but the search results still require a lot of labor to record and re-use, and also to validate, because in practice, search results are frequently incomplete and unreliable. Most journal publishers have Web sites that provide contents information for journal volumes, but there is no common presentation format. Also, a search of one publisher's site is unlikely to find related works in journals produced by its competitors.

The BIBTEX bibliography markup system developed by Oren Patashnik at Stanford University as part of the decade-long TEX Project has provided an important solution to the problem of making publication information *reusable*. This author has expended considerable effort in writing software to convert library and publisher data into BIBTEX form, and developed many tools for checking, curating, ordering, searching, sorting, and validating BIBTEX data.

Two freely available collections, the *BibNet Project* archives and the *TEX User Group* archives, both hosted at the author's university, and mirrored to other sites, provide a view into the research literature of selected areas of chemistry, computer science, computer standards, cryptography, mathematics, physics, probability and statistics, publication metrics, typesetting, and their associated histories.

Those archives provide a substantial pool of data from which specialized collections, such as the bibliography for this book, can be relatively easily derived, *without* retyping publication data. BIBTEX supports citation, cross

referencing, extraction, sorting, and formatting of publication data in hundreds of styles. Additional software written by this author extends BIBTEX by automatically producing the separate author/editor index that appears in the back matter of this book, and enhancing bibliography entries with lists of page numbers where each is cited. Thus, a reader who remembers *just one author* of a cited work can quickly find both the reference, and the locations in the book where it is cited.

The MathCW Web site

This book is accompanied by a Web site maintained by this author at

`http://www.math.utah.edu/pub/mathcw/`

That site contains

- source code for the book's software;
- a BIBTEX database, `mathcw.bib`, from a subset of which all references that appear in the bibliography in this book's back matter are automatically derived and formatted;
- related materials developed after the book has been frozen for publication;
- compiled libraries for numerous systems; and
- pre-built C compilers with support for decimal arithmetic on several operating systems.

It is expected to be mirrored to many other sites around the world, to ensure wide availability, and protection against loss.

The mathcw software is released in versioned bundles, and its history is maintained in a revision control system to preserve a record of its development and future evolution, as is common with most large modern software projects.

Contents

List of figures

List of tables

Quick start

THIS QUICK START FEATURE IS NOT COMPREHENSIVE:
THERE MAY BE ADDITIONAL OSHA STANDARDS AND
GUIDANCE MATERIALS THAT ALSO APPLY.

— WEB SITE DISCLAIMER, U.S. DEPARTMENT OF LABOR,
OCCUPATIONAL SAFETY & HEALTH ADMINISTRATION.

The mathcw library implements the elementary mathematical functions mandated by C89 and C99, normally supplied by the `-lm` library on most UNIX systems. **Table 3.4** on page 58 summarizes the mathcw library contents.

Much more information is provided in the remaining chapters of this book, but if you are only interested in library installation, then this short section provides enough information for that task.

To build, validate, and install this library on almost any modern UNIX or POSIX-compatible system, this widely used GNU recipe does the job:

```
% ./configure && make all check install
```

On Hewlett–Packard HP-UX on IA-64 (Intel Architecture 64-bit), change the target `all` to `all-hp` to get library support for two additional precisions available on that platform; see **Section 4.26** on page 100 for details. The corresponding `check-hp` target tests all five supported precisions.

To change the default installation tree from the GNU-standard default of `/usr/local`, define `prefix` to an equivalent value at install time:

```
% make prefix=/depot install
```

That would create the library file `/depot/lib/libmcw.a`.

The mathcw library can be used like this with either C or C++ compilers:

```
% cc [ flags ] [ -I$prefix/include ] file(s) [ -L$prefix ] -lmcw
```

The `-I` option is needed if the code includes this package's `mathcw.h` header file. That may not be necessary, if `<math.h>` is included. Depending on the local conventions, the `-L` option may or may not be required to define a load-library path. `$prefix` must be replaced by the local default, such as `/usr/local`.

Caution is needed on systems for which the C header file `<math.h>` redefines library function names, or where compilers produce inline code for elementary functions, such as on the Intel IA-32 (formerly, x86) architecture, which has single hardware instructions for several of them. For the GNU compiler family, use the option `-fno-builtin` to permit library routines to replace inline code.

Caution is also needed when the host default floating-point behavior is not IEEE-754 conformant. GNU/LINUX and OSF/1 operating systems with Alpha processors have this defect: underflows flush abruptly to zero, and overflows and operations with NaN immediately terminate the process. To get mostly conformant behavior, with GNU compilers, use the `-mieee` flag, and with native compilers, use the `-ieee` flag. To get full conformance, use `-mieee-with-inexact` or `-mieee-conformant`, plus `-mfp-rounding-mode=d`, with GNU compilers, and `-ieee_with_inexact` and `-mfp-rounding-mode=d` with native ones.

On IBM AIX 4.2, `long double` is compiled as `double`, unless the native compiler name is changed from `cc` to `cc128`, or the `-qlongdouble` option is specified. Run-time library support of 128-bit `long double` requires linking with `cc128`, or explicit specification of the `-lc128` library. The easiest way to build the mathcw library is then like this:

```
% make CC=cc128 all check
```

The GNU compiler `gcc` version 2.95.3 does not support a 128-bit `long double` at all, and no newer version builds successfully on AIX 4.2.

It is possible to compile a subset of the package, either manually to select replacements for deficient implementations in the native `-lm` library, or by precision, with the targets `float`, `double`, and `longdouble`. For example, pre-C99

implementations of the C language may lack some or all of the `float` and `long double` elementary functions, providing only the `double` versions.

To clean up after a build, do this:

```
% make clean
```

To return to the original state of a freshly unpacked distribution, use the command

```
% make distclean
```

More details about the interpretation of the test results are given later in **Chapter 22** on page 763.

The mathcw library distribution includes interfaces that make the library accessible from several other major programming languages. The interfaces are described in appendices of this book, beginning on page 911.

For up-to-date information about building the library, run the command

```
% make help
```

to get further information about platform-specific targets, and about building shared libraries.