# New directions in floating-point arithmetic [Extended abstract]

Nelson H. F. Beebe

*University of Utah*
*Department of Mathematics, 110 LCB*
*155 S 1400 E RM 233*
*Salt Lake City, UT 84112-0090*
*USA*

**Abstract.** This article briefly describes the history of floating-point arithmetic, the development and features of IEEE standards for such arithmetic, desirable features of new implementations of floating-point hardware, and discusses work-in-progress aimed at making decimal floating-point arithmetic widely available across many architectures, operating systems, and programming languages.

**Keywords:** binary arithmetic; decimal arithmetic; elementary functions; fixed-point arithmetic; floating-point arithmetic; interval arithmetic; `mathcw` library; range arithmetic; special functions; validated numerics
**PACS:** 02.30.Gp Special functions

## DEDICATION

This article is dedicated to N. Yngve Öhrn, my mentor, thesis co-advisor, and long-time friend, on the occasion of his retirement from academia. It is also dedicated to William Kahan, Michael Cowlishaw, and the late James Wilkinson, with much thanks for inspiration.

## WHAT IS FLOATING-POINT ARITHMETIC?

*Floating-point arithmetic* is a technique for storing and operating on numbers in a computer where the base, range, and precision of the number system are usually fixed by the computer design.

Conceptually, a floating-point number has a *sign*, an *exponent*, and a *significand* (the older term *mantissa* is now deprecated), allowing a representation of the form $(-1)^{\text{sign}} \times \text{significand} \times \text{base}^{\text{exponent}}$. The *base point* in the significand may be at the left, or after the first digit, or at the right. The point and the base are implicit in the representation: neither is stored.

The sign can be compactly represented by a single bit, and the exponent is most commonly a biased unsigned bitfield, although some historical architectures used a separate exponent sign and an unbiased exponent. Once the sizes of the sign and exponent fields are fixed, all of the remaining storage is available for the significand, although in some older systems, part of this storage is unused, and usually, ignored. On modern systems, the storage order is conceptually sign, exponent, and significand, but addressing conventions on byte-addressable systems (the *big endian* versus *little endian* theologies) can alter that order, and some historical designs reordered them, and sometimes split the exponent and significand fields into two interleaved parts. Except when the low-level storage format must be examined by software, such as for binary data exchange, these differences are handled by hardware, and are rarely of concern to programmers.

The data size is usually closely related to the computer word size. Indeed, the venerable Fortran programming language mandates a *single-precision* floating-point format occupying the same storage as an integer, and a *double-precision* format occupying exactly twice the space. This requirement is heavily relied on by Fortran software for array dimensioning, argument passing, and in `COMMON` and `EQUIVALENCE` statements for storage alignment and layout. Some vendors later added support for a third format, called *quadruple-precision*, occupying four words. Wider formats have yet to be offered by commercially-viable architectures, although we address this point later in this article.

Floating-point arithmetic can be contrasted with *fixed-point* arithmetic, where there is no exponent field, and it is the programmer's responsibility to keep track of where the base point lies. Address arithmetic, and signed and unsigned integer arithmetic, are special cases of fixed-point arithmetic where the base point is always at the right, so only whole numbers can be represented, and for address and unsigned integer arithmetic, the storage occupied by the sign is given to the number field.

Floating-point number systems have limited precision and range, and their arithmetic is *not* associative. These properties are at odds with mathematical arithmetic, and often require great care to handle correctly in software. They often produce large gaps between a problem's mathematical solution and a practical, and accurate, computational solution.

Also, compiler optimizations, instruction reordering, and use of higher precision for intermediate computations, can all produce unpleasant surprises in floating-point software. Consequently, numerical programmers must be highly experienced, eternally vigilant, and fanatic about testing on every accessible platform.

## DECIMAL ARITHMETIC

The Rexx and NetRexx scripting languages [1, 2] developed by IBM Fellow Michael Cowlishaw provide a software implementation of decimal floating-point arithmetic with up to a billion ($10^9$) digits, and a huge exponent range of $10^{\pm 999,999,999}$. Based on long experience with those languages, in 2006, IBM researchers developed a firmware implementation of IEEE-like decimal arithmetic for the z9 mainframe [3, 4], and on May 21, 2007, IBM announced the POWER6 processor with the first hardware implementation since the 1960s, outside of handheld calculators, of decimal floating-point arithmetic. For a survey of historical decimal systems, see [5].

In 2005, ISO committees for the standardization of the C and C++ languages received proposals [6, 7, 8, 9] for the incorporation of decimal floating-point arithmetic. In late 2006, GNU developers of compilers for those languages added preliminary support on one CPU platform for the new data types, and in 2007, more CPU platforms followed. The underlying arithmetic is supplied by Cowlishaw's `decNumber` library [10], but no debugger or library support for I/O and mathematical functions is included. The decimal formats are 32-bit, 64-bit, and 128-bit, with precisions of 7, 16, and 34 digits, respectively. Their nonzero magnitude ranges are approximately $[10^{-101}, 10^{97}]$, $[10^{-398}, 10^{385}]$, and $[10^{-6176}, 10^{6145}]$, somewhat wider than the corresponding binary formats.

The `decNumber` library provides a superset of the features of the IBM hardware implementation, including support for *eight* rounding modes (additional modes are needed to meet rounding rules in financial computations mandated by various legal jurisdictions), along with all of the other features of IEEE 754 arithmetic.

Unlike the binary format, where the point lies after the first digit, the decimal significand is an *integer coefficient*, and thus, trailing zeros allow multiple representations of the same numeric value. This design choice was intentional, because it allows decimal fixed-point arithmetic, historically required for financial computations, to be done in decimal floating-point arithmetic, and additional library functions make it possible to get and set, the scale, or *quantization*. In financial work, these operations are expected to be common.

One important ramification for programmers is that multiplication by 1., 1.0, 1.00, ... each produce different quantization, and similarly, multiplication by 1000 differs in quantization from multiplication by $1 \times 10^3$. Since most people are taught in school to write at least one digit following a decimal point, programmers of decimal arithmetic need to learn to avoid introducing unwanted trailing zeros so as to preserve quantization.

Another significant point is that financial computations need large numbers of digits: older COBOL standards required support for 18 decimal digits, and the 2002 ISO COBOL Standard [11] mandates 32 digits. Thus, the 128-bit format is expected to be widely used, and market pressure will force it to be in hardware, rather than in software as is currently the case for the binary format of that size on several platforms.

The IBM hardware and software implementations of decimal arithmetic use an encoding called *DPD (Densely-Packed Decimal)* [10, 12], which represents three decimal digits in ten bits. DPD is more compact than older schemes, such as Binary-Coded Decimal (BCD), that have long been used for fixed-point decimal arithmetic in many processors.

Intel researchers have developed a competing encoding called *BID (Binary Integer Decimal)* and implemented it in a prototype reference library [13]. BID is designed to allow chip designers to reuse circuitry from integer arithmetic units, but appears to make correct decimal rounding difficult. Fortunately, detailed analysis has made it possible to develop computational algorithms that can guarantee correct decimal rounding for the basic operations of add, subtract, multiply, and divide. From the programmer's point of view, the two encodings offer identical floating-point characteristics, but there are small differences in the representable range that could make it possible for software to distinguish between them, and thus, make assumptions that limit portability.

In mid 2007, the GNU compilers were extended to generate code for both the IBM and Intel libraries, and it is expected that Intel's own compilers will soon have support for decimal arithmetic as well. It seems likely that future Intel processors will provide decimal arithmetic in hardware, once sufficient software experience with both BID and DPD encodings has been accumulated to guide the final choice of encoding. Since most desktop computers use CPUs from the Intel architecture families, in just a few years, decimal floating-point arithmetic may be widespread, and competitive with binary floating-point arithmetic in efficiency.

Since 2005, this author has been developing a large highly-portable elementary function library, called the `mathcw` library, that includes all of the mathematical repertoire of the 1999 ISO C Standard, plus a great deal more. This library has been designed to provide a comfortable C99 environment on all current platforms, as well as to run on historical architectures of the 1970s, such as the PDP-10, PDP-11, and VAX, which remain available via software virtual machines that, thanks to advances in processor technology, can be an order of magnitude faster than the hardware ever was. The library currently supports six binary types, and four decimal types, covering almost every significant computing platform of the last thirty years.

In addition, the `mathcw` library is designed to pave the way for future *octuple-precision* arithmetic in a 256-bit format offering a significand of 235 bits in binary, and 70 digits in decimal. The approximate nonzero magnitude ranges are $[10^{-315\,653}, 10^{315\,652}]$ in binary and $[10^{-1\,572\,932}, 10^{1\,572\,863}]$ in decimal.

While it is relatively straightforward to add support for new data types and machine instructions (or interfaces to software implementations thereof) in modern compilers, there is a huge hurdle to overcome in providing a powerful run-time library for I/O and mathematical functions. The `mathcw` library removes that obstacle, and makes decimal arithmetic as comfortable, and accessible, as binary arithmetic.

Fast and compact implementations of many of the elementary and special functions are based on a combination of range reduction, and evaluation of a rational polynomial that accurately represents the function itself, or more commonly, an auxiliary function from which the desired function can be easily obtained, but only over a limited argument interval. In particular, this means that the polynomial approximation must be adapted to both the interval, and the required precision, and it is then impossible to port the software to systems with differing precision, or to modify it for use on a different interval, without having the means to generate new polynomial approximations. To eliminate this problem, which is widespread in previous mathematical software libraries, the `mathcw` library uses polynomials output in formats suitable for use in C, Fortran, and `hoc` by auxiliary software written in the Maple symbolic-algebra language. The Maple output is manually copied into the library's header files. To facilitate algorithm modification, and use with even higher precision in the distant future, all of the Maple programs are included in the `mathcw` software distribution.

The I/O part of the library includes important extensions to allow handling of numbers in any base from 2 to 36 (e.g., `8@3.11037554@e+0`, `10@3.14159265@e+0`, `16@3.243f6c@e+0`, and `36@3.53i5g@e+0` all represent 32-bit approximations to $\pi$), control over exponent widths (a feature available in Fortran since the 1978 ISO Standard, but still lacking in the most-recent C, C++, and Java run-time libraries), and digit grouping with separating underscores, making long digit strings much more readable, so that

```
static const decimal_long_double PI = 3.141_592_653_589_793_238_462_643_383_279_503;
```

becomes a valid initialization in C or C++. Such numbers can be used in input and output files, and once compilers are trivially extended, also in programming-language source code.

The `mathcw` library includes an extensive collection of functions for *pair-precision arithmetic*, which simulates double-length significands in every supported precision. It is sometimes the case that somewhat higher precision in just a small part of a large computation can make a dramatic improvement in overall accuracy, and these functions fill that need.

The `mathcw` library can interfaced to many other programming languages, and interfaces to Ada, C#, C++, Fortran, Java, and Pascal are provided with the system.

As additional proof of concept, *five* scripting language compilers/interpreters (three for `awk`, plus `hoc` and `lua`) have been adapted to use decimal, rather than binary, floating-point arithmetic, and allow digit-separating underscores in source code. All of them pass their full validation suites as well as they do for the original versions that use binary arithmetic. In each case, less than 3% of the source code needed changes for decimal arithmetic, so this author is confident that the scores of other popular scripting languages implemented in C or C++, such as `icon`, `javascript`, `perl`, `php`, `python`, `ruby`, the Unix shells, and so on, can be similarly updated, each in less than a day's work, making decimal floating-point arithmetic the norm almost everywhere within just a few years. With highly-portable underpinnings in the form of the `mathcw` library, these languages could easily offer a much richer mathematical

function repertoire, including access to all of the features in the IEEE 754 Standard, instead of the rather limited function set chosen for Fortran more than 50 years ago.

The `mathcw` library is described in a forthcoming book [14], and is to be released under a software license that ensures free and unrestricted source-code access to everyone, in the tradition of the numerical mathematics community, and the free software movement.

## FURTHER READING

Overton's 100-page book [15] provides a useful introduction to more details of floating-point arithmetic than we can provide in this short article. This author's book [14] gives much more information, including guidance for creating numerical programs that are independent of precision and range, and usually independent of base.

There are numerous other books and technical papers on the subject of computer arithmetic, and the best advice is to consult the comprehensive on-line bibliography that is actively maintained by this author at `http://www.math.utah.edu/pub/tex/bib/index-table.html#fparith`.

## REFERENCES


1. M. F. Cowlishaw, *The REXX language: a practical approach to programming*, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1985, ISBN 0-13-780735-X (paperback).
2. M. F. Cowlishaw, *The NetRexx language*, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1997, ISBN 0-13-806332-X, see also supplement [16].
3. *Preliminary Decimal-Floating-Point Architecture*, IBM Corporation, San Jose, CA, USA (2006), URL `http://publibz.boulder.ibm.com/epubs/pdf/a2322320.pdf;http://www-03.ibm.com/servers/eserver/zseries/zos/bkserv/r3pdf/zarchpops.html`, form number SA23-2232-00.
4. A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, *IBM Journal of Research and Development* **51**, 217–227 (2007), ISSN 0018-8646, URL `http://www.research.ibm.com/journal/rd/511/duale.html`.
5. M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: proceedings: Santiago de Compostela, Spain, June 15–18, 2003*, edited by J. C. Bajard, and M. Schulte, IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2003, pp. 104–111, ISBN 0-7695-1894-X, ISSN 1063-6889, URL `http://www.dec.usc.es/arith16/papers/paper-107.pdf`.
6. R. Klarer, Decimal types for C++: Second draft, Report C22/WG21/N1839 J16/05-0099, IBM Canada, Ltd., Toronto, ON, Canada (2005), URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1839.html`.
7. ISO, ISO/IEC JTC1 SC22 WG14 N1154: Extension for the programming language C to support decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1154.pdf`.
8. ISO, ISO/IEC JTC1 SC22 WG14 N1161: Rationale for TR 24732: Extension to the programming language C: Decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1161.pdf`.
9. ISO, ISO/IEC JTC1 SC22 WG14 N1176: Extension for the programming language c to support decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://open-std.org/jtc1/sc22/wg14/www/docs/n1176.pdf`.
10. M. Cowlishaw, *The decNumber C library*, IBM Corporation, San Jose, CA, USA (2007), URL `http://download.icu-project.org/ex/files/decNumber/decNumber-icu-340.zip`, version 3.40.
11. International Organization for Standardization, *ISO/IEC 1989:2002: Information technology — Programming languages — COBOL*, International Organization for Standardization, Geneva, Switzerland, 2002, ISBN ????, URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=28805`.
12. M. F. Cowlishaw, *IEE Proceedings. Computers and Digital Techniques* **149**, 102–104 (2002), ISSN 1350-2387.
13. Anonymous, Reference software implementation of the IEEE 754R decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://cache-www.intel.com/cd/00/00/29/43/294339_294339.pdf`.
14. N. H. F. Beebe, *The `mathcw` Portable Elementary Function Library*, 2008, in preparation.
15. M. Overton, *Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001, ISBN 0-89871-482-6, URL `http://www.siam.org/catalog/mcc07/ot76.htm,http://www.cs.nyu.edu/cs/faculty/overton/book/`.
16. M. F. Cowlishaw, *NetRexx Language Supplement*, IBM UK Laboratories, Hursley Park, Winchester, England (2000), URL `http://www-306.ibm.com/software/awdtools/netrexx/nrlsupp.pdf`, version 2.00. This document is a supplement to [2].