Extending TeX and METAFONT with floating-point arithmetic

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
WWW URL: `http://www.math.utah.edu/~beebe`
Telephone: +1 801 581 5254
FAX: +1 801 581 4148
Internet: `beebe@math.utah.edu`, `beebe@acm.org`, `beebe@computer.org`

Abstract

The article surveys the state of arithmetic in TeX and METAFONT, suggests that they could usefully be extended to support floating-point arithmetic, and shows how this could be done with a relatively small effort, *without* loss of the important feature of platform-independent results from those programs, and *without* invalidating any existing documents, or software written for those programs, including output drivers.

Contents

1 Dedication

This article is dedicated to Professors Donald Knuth (Stanford University) and William Kahan (University of California, Berkeley), with thanks for their many scientific and technical contributions, and for their writing.

2 Introduction

Arithmetic is a fundamental feature of computer programming languages, and for some of us, the more we use computers, the more inadequate we find their computational facilities. The arithmetic in TeX and METAFONT is particularly limiting, and this article explains why this is so, why it was not otherwise, and what can be done about it now that these two important programs are in their thirtieth year of use.

3 Arithmetic in TeX and METAFONT

Before we look at issues of arithmetic in general, it is useful to summarize what kinds of numbers TeX and METAFONT can handle, and how they do so.

TeX provides binary integer and fixed-point arithmetic. Integer arithmetic is used to count things, such as with TeX's `\count` registers. Fixed-point arithmetic is needed for values that have fractional parts, such as the `\dimen` dimension registers, the `\muskip` and `\skip` glue registers, and scale factors, as in `0.6\hsize`.

For portability reasons, TeX requires that the host computer support an integer data type of at least 32 bits. It uses that type for the integer arithmetic available to TeX programs. For fixed-point numbers, it reserves the lower 16 bits for the fractional part, and all but two of the remaining bits for the integer part. Thus, on the 32-bit processors that are commonly found in personal computers, 14 bits are available for the integer part. One of the remaining two bits is chosen as a sign bit, and the other is used to detect *overflow*, that is, generation of a number that is too large to represent.

When fractional numbers represent TeX dimensions, the low-order fraction bit represents the value 2^{-16} pt. While printer's points have been a common unit of measurement since well before the advent of computer-based typesetting, this tiny value is new with TeX, and has the special name *scaled point*. The value 1 sp is so small that approximately 100 sp is about the wavelength of visible light. This ensures that differences of a few scaled points in the positioning of objects on the printed page are completely invisible to human eyes.

The problem with fixed-point numbers in TeX is at the other end: 14 integer bits can only represent numbers up to 16383. As a dimension, that many points is about 5.75 m, which is probably adequate for printed documents, but is marginal if you are typesetting a billboard. The PDP-10 computers on which TeX and METAFONT were developed had 36-bit words: the four extra bits raised the maximum dimension by a factor of 16. Nevertheless, if TeX's fixed-point numbers are used for purposes other than page dimensions, then it is easy to exceed their limits.

TeX is a macro-extensible language for typesetting, and arithmetic is expected to be relatively rare. TeX has little support for numerical expressions, just verbose low-level operators, forcing the TeX programmer to write code such as this fragment from `layout.tex` to accomplish the multiply-add operation noted in the comment:

```
% MRGNOTEYA = 0.75*TEXTHEIGHT + FOOTSKIP
\T = \TEXTHEIGHT
\multiply \T by 75  % possible overflow!
\divide \T by 100
\advance \T by \FOOTSKIP
\xdef \MRGNOTEYA {\the \T}
```

Notice that the scale factor 0.75 could have been reduced from 75/100 to 3/4 in this example, but that is not in general possible. Similarly, here we could have written `\T = 0.75 \TEXTHEIGHT`, but that is not possible if the constant 0.75 is replaced by a variable in a register. The multiplication by 75 can easily provoke an overflow if `\T` is even as big as a finger length:

```
*\dimen1 = 220pt

*\dimen2 = 75\dimen1
! Dimension too large.

*\multiply \dimen1 by 75
! Arithmetic overflow.
```

See the LaTeX `calc` package for more horrors of fixed-point arithmetic.

TeX has, however, also seen use a scripting language, chosen primarily because of its superb quality, stability, reliability, and platform independence. TeX distributions now contain macro packages and utilities written in TeX for generating complex font tables, for packing and unpacking document archives, for scanning PostScript graphics files, and even for parsing SGML and XML.

TeX's arithmetic does not go beyond the four basic operations of *add*, *subtract*, *multiply*, and *divide*. In particular, no elementary functions (square root, exponential, logarithm, trigonometric and hyperbolic functions, and so on) are provided in TeX itself, even though, in principle, they can be provided with macro packages.

In TeX, overflow is detected in division and multiplication but not in addition and subtraction, as I described in my *TUG 2003* keynote address [4].

Input numbers in METAFONT are restricted to 12 integer bits, and the result of even trivial expressions can be quite surprising to users:

```
% mf expr
gimme an expr: 4095       >> 4095
gimme an expr: 4096
! Enormous number has been reduced.
>> 4095.99998
gimme an expr: infinity   >> 4095.99998
gimme an expr: epsilon    >> 0.00002
gimme an expr: 1/epsilon
! Arithmetic overflow.
>> 32767.99998
gimme an expr: 1/3        >> 0.33333
gimme an expr: 3*(1/3)    >> 0.99998
gimme an expr: 1.2 - 2.3  >> -1.1
gimme an expr: 1.2 - 2.4  >> -1.2
gimme an expr: 1.3 - 2.4  >> -1.09999
```

Notice that although 4096 is considered an overflow, internally METAFONT can generate a number almost eight times as large. Binary-to-decimal conversion issues produce the anomaly in $3 \times (1/3)$. The last line shows that even apparently simple operations are not so simple after all.

Overflows in METAFONT can also produce a report like this:

```
Uh, oh.  A little while ago one of the
quantities that I was computing got
too large, so I'm afraid your answers
will be somewhat askew.  You'll
probably have to adopt different
tactics next time.  But I shall try to
carry on anyway.
```

METAFONT provides a few elementary functions: `++` (Pythagoras), `abs`, `angle`, `ceiling`, `cosd`, `dir`, `floor`, `length`, `mexp`, `mlog`, `normaldeviate`, `round`, `sind`, `sqrt`, and `uniformdeviate`. They prove useful in the geometric operations required in font design.

4 Historical remarks

TEX and METAFONT are not the only systems that suffer from the limitations of fixed-point arithmetic. Most early computers were inadequate as well:

> It is difficult today to appreciate that probably the biggest problem facing programmers in the early 1950s was scaling numbers so as to achieve acceptable precision from a fixed-point machine.
>
> Martin Campbell-Kelly
> *Programming the Mark I: Early Programming Activity at the University of Manchester*
> Annals of the History of Computing,
> **2**(2) 130–168 (1980)

Scaling problems can be made much less severe if numbers carry an exponent as well as integer and fractional parts. We then have:

> Floating Point Arithmetic ... The subject is not at all as trivial as most people think, and it involves a surprising amount of interesting information.
>
> Donald E. Knuth
> *The Art of Computer Programming: Seminumerical Algorithms* (1998)

However, more than just an exponent is needed; the arithmetic system also has to be predictable:

> Computer hardware designers can make their machines much more pleasant to use, for example by providing *floating-point arithmetic* which satisfies simple mathematical laws.
>
> The facilities presently available on most machines make the job of rigorous error analysis *hopelessly difficult*, but properly designed operations would encourage numerical analysts to provide better subroutines which have certified accuracy.
>
> Donald E. Knuth
> *Computer Programming as an Art*
> *ACM Turing Award Lecture* (1973)

5 Why no floating-point arithmetic?

Neither TEX nor METAFONT have floating-point arithmetic natively available, and as I discussed in my *Practical TEX 2005* keynote address [3], there is a very good reason why this is the case. Their output needs to be identical on all platforms, and when they were developed, there were many different computer vendors, some of which had several incompatible product lines. This diversity causes several problems, some of which still exist:

- There is system dependence in *precision*, *range*, *rounding*, *underflow*, and *overflow*.
- The number base varies from 2 on most, to 3 (Setun), 4 (Illiac II), 8 (Burroughs), 10, 16 (IBM S/360), 256 (Illiac III), and 10000 (Maple).
- Floating-point arithmetic exhibits bizarre behavior on some systems:
  - $x \times y \neq y \times x$ (early Crays);
  - $x \neq 1.0 \times x$ (Pr1me);
  - $x + x \neq 2 \times x$ (Pr1me);
  - $x \neq y$ but $1.0/(x - y)$ gets zero-divide error;
  - wrap between underflow and overflow (e.g., C on PDP-10);
  - job termination on overflow or zero-divide (most).
- No standardization: almost every vendor had one or more distinct floating-point systems.
- Programming language dependence on available precisions:
  - Algol, Pascal, and SAIL (only `real`): recall that SAIL was the implementation language for the 1977–78 prototypes of TEX and METAFONT;
  - Fortran (`REAL`, `DOUBLE PRECISION`, and on some systems, `REAL*10` or `REAL*16`);
  - C/C++ (originally only `double`, but `float` added in 1989, and `long double` in 1999);
  - C# and Java have only `float` and `double` data types, but their arithmetic is badly botched: see Kahan and Darcy's *How Java's Floating-Point Hurts Everyone Everywhere* [28].
- Compiler dependence: multiple precisions can be mapped to just one, without warning.
- BSD compilers on IA-32 still provide no 80-bit format after 27 years in hardware.
- Input/output problem requires base conversion, and is *hard* (e.g., conversion from 128-bit binary format can require more than 11 500 decimal digits).
- Most languages do not guarantee exact base conversion.

Donald Knuth wrote an interesting article with the intriguing title *A simple program whose proof isn't* [29] about how TEX handles conversions between fixed-point binary and decimal. The restriction to fixed-point arithmetic with 16-bit fractional parts simplifies the base-conversion problem, and allows TEX to guarantee exact round-trip conversions of such numbers.

TeX produces the same line- and page-breaking across all platforms, because floating-point arithmetic is used only for interword glue calculations that could change the horizontal position of a letter by at most a few scaled points, but as we noted earlier, that is invisible.

METAFONT has no floating-point at all, and generates identical fonts on all systems.

6 IEEE 754 binary floating-point standard

With the leadership of William Kahan, a group of researchers in academic, government, and industry began a collaborative effort in the mid-1970s to design a new and much improved floating-point architecture. The history of this project is chronicled in an interview with Kahan [37, 38].

A preliminary version of this design was first implemented in the Intel 8087 chip in 1980, although the design was not finalized until its publication as IEEE Standard 754 in 1985 [23].

Entire books have beeen written about floating-point arithmetic: see, for example, Sterbenz [41] for historical systems, Overton [35] for modern ones, Omondi [34] and Parhami [36] for hardware, Goldberg [15, 17] for an excellent tutorial, and Knuth [30, Chap. 4] for theory. I am hard at work on writing two more books in this area. However, here we need only summarize important features of the IEEE 754 system:

- Three formats are defined: 32-bit, 64-bit, and 80-bit. A 128-bit format was subsequently provided on some Alpha, IA-64, PA-RISC, and SPARC systems.
- Nonzero normal numbers are *rational*: $x = (-1)^s f \times 2^p$, where the *significand*, f, lies in $[1, 2)$.
- Signed zero allows recording the direction from which an underflow occurred, and is particularly useful for arithmetic with complex (real + imaginary) numbers. The IEEE Standard requires that $\sqrt{-0}$ evaluate to -0.
- The largest stored exponent represents Infinity if $f = 0$, and either quiet or signaling NaN (Not-a-Number) if $f \neq 0$. A vendor-chosen significand bit distinguishes between the two kinds of NaN.
- The smallest stored exponent allows leading zeros in f for *gradual underflow* to *subnormal* values.
- The arithmetic supports a model of fast *nonstop* computing. Sticky flags record exceptions, and Infinity, NaN, and zero values automatically replace out-of-range values, without the need to invoke an exception handler, although that capability may also be available.
- Four rounding modes are provided:
  - *to nearest with ties to even* (default);
  - *to* $+\infty$;
  - *to* $-\infty$;
  - *to zero* (historical chopping).
- Values of $\pm\infty$ are generated from huge/tiny and finite/0.
- NaN values are generated from $0/0$, $\infty - \infty$, ∞/∞, and any operation with a NaN operand.
- A NaN is returned from functions when the result is undefined in real arithmetic (e.g., $\sqrt{-1}$), or when an argument is a NaN.
- NaNs have the property that they are unequal to anything, even themselves. Thus, the C-language inequality test `x != x` is true *if*, and *only if*, `x` is a NaN, and should be readily expressible in *any* programming language. Sadly, several compilers botch this, and get the wrong answer.

7 IEEE 754R precision and range

In any computer arithmetic system, it is essential to know the available range and precision. The precisions of the four IEEE 754 binary formats are equivalent to approximately 7, 15, 19, and 34 decimal digits. The approximate ranges as powers of ten, including subnormal numbers, are $[-45, 38]$, $[-324, 308]$, $[-4951, 4932]$, and $[-4966, 4932]$. A future 256-bit binary format will supply about 70 decimal digits, and powers-of-ten in $[-315\,723, 315\,652]$.

A forthcoming revision of the IEEE Standard will include decimal arithmetic as well, in 32-, 64-, and 128-bit storage sizes, and we can imagine a future 256-bit size. Their precisions are 7, 16, 34, and 70 decimal digits, where each doubling in size moves from n digits to $2n + 2$ digits. Their ranges are wider than the binary formats, with powers of ten in $[-101, 96]$, $[-398, 384]$, $[-6176, 6144]$, and $[-1\,572\,932, 1\,572\,864]$.

In each case, the range and precision are determined by the number of bits allocated for the sign and the significand, and for the decimal formats, by restrictions imposed by the compact encodings chosen for packing decimal digits into strings of bits.

It is highly desirable that each larger storage size increase the exponent range (many older designs did not), and at least double the significand length, since that guarantees that products evaluated in the next higher precision *cannot overflow*, and are *exact*. For example, the Euclidean distance $\sqrt{x^2 + y^2}$ is then trivial to compute; otherwise, its computation requires careful rescaling to avoid premature underflow and overflow.

8 Remarks on floating-point arithmetic

Contrary to popular misconception, even present in some books and compilers, floating-point arithmetic is *not fuzzy*:

- Results are *exact* if they are representable.
- Multiplication by a power of base is always exact, in the absence of underflow and overflow.
- Subtracting numbers of like signs and exponents is *exact*.

Bases other than 2 or 10 suffer from *wobbling precision* caused by the requirement that significands be normalized. For example, in hexadecimal arithmetic, $\pi/2 \approx 1.571 \approx 1.922_{16}$ has three fewer bits (almost one decimal digit) than $\pi/4 \approx 0.7854 \approx \mathrm{c.910}_{16}$. Careful coders on such systems account for this in their programs by writing

```
y = (x + quarter_pi) + quarter_pi;
```

instead of

```
y = x + half_pi;
```

Because computer arithmetic systems have finite range and precision, they are not *associative*, so commonly-assumed mathematical transformations do not hold. In particular, it is often necessary to control evaluation order, and this may be at odds with what the compiler, or even a high-performance CPU with dynamic instruction reordering, does with the code.

The presence of multiple rounding modes also invalidates common assumptions. For example, the Taylor series for the sine function begins

$$\sin(x) = x - (1/3!)x^3 + (1/5!)x^5 - \cdots.$$

If x is small enough, because of finite precision, one might expect that $\sin(x)$ could be computed simply as x. However, that is only true for the default rounding mode; in other modes, the correct answer could be one *ulp* (unit in the last place) higher or lower, so at least two terms must be summed. Similarly, the mathematical equivalence $-(xy+z) \equiv (-xy-z)$ does not hold in some rounding modes. Except for some special numbers, it is not in general permissible to replace slow division with fast multiplication by the reciprocal, even though many optimizing compilers do that.

Some of the common elementary functions are *odd* ones: they satisfy $f(x) = -f(-x)$. This relation does not in general hold computationally if a rounding direction of other than *round-to-nearest* is in effect. Software designers are then forced to decide whether obeying computer rounding modes is more important than preserving fundamental mathematical symmetries: in well-designed software, symmetry wins. Nevertheless, in some applications, like *interval arithmetic*, which computes upper and lower bounds for every numeric operation, precise control of rounding is imperative, and overrides symmetry.

See Monniaux [33] for a recent discussion of some of the many problems of floating-point evaluation. A good part of the difficulties described there arise because of higher intermediate precision in the Intel IA-32 architecture, the most common desktop CPU family today. Other problems come from unexpected instruction reordering or multiple threads of execution, and the incidence of these issues increases with each new generation of modern processors.

9 Binary versus decimal

Why should we care whether a computer uses binary or decimal arithmetic? Here are some reasons why a switch to decimal arithmetic has advantages:

- Humans are less uncomfortable with decimal arithmetic.
- In some case, binary arithmetic always gets the wrong answer. Consider this sales tax computation: 5% of 0.70 = 0.0349999... in *all* binary precisions, instead of the exact decimal 0.035. Thus, there can be significant cumulative rounding errors in businesses with many small transactions (food, music downloading, telephone, ...).
- Financial computations need fixed-point decimal arithmetic.
- Hand calculators use decimal arithmetic.
- Additional decimal rounding rules (eight instead of four) handle the financial and legal requirements of some jurisdictions.
- Decimal arithmetic eliminates most base-conversion problems.
- There is a specification of decimal arithmetic subsumed in the *IEEE 854-1987 Standard for Radix-Independent Floating-Point Arithmetic* [21].
- Older Cobol standards require 18D fixed-point.
- Cobol 2002 requires 32D fixed-point *and* floating-point.
- Proposals to add decimal arithmetic to C and C++ were submitted to the ISO language committees in 2005 and 2006.
- Twenty-five years of Rexx and NetRexx scripting languages give valuable experience in arbitrary-precision decimal arithmetic.

- The excellent IBM `decNumber` library provides *open source* decimal floating-point arithmetic with a billion (10^9) digits of precision and exponent magnitudes up to 999 999 999.
- Preliminary support in `gcc` for +, −, ×, and / became available in late 2006, based on a subset of the IBM `decNumber` library.
- The author's `mathcw` package [5] provides a C99-compliant run-time library for binary, and also for decimal, arithmetic (2005–2008), with hundreds of additional functions, and important and useful extensions of the I/O functions.
- IBM zSeries mainframes got IEEE 754 binary floating-point arithmetic in 1999, and decimal floating-point arithmetic in firmware in 2006.
- The IBM PowerPC version 6 chips announced on 21 May 2007 add hardware decimal arithmetic, probably the first mainstream processor to do so in more than four decades.
- Hardware support seems likely in future Intel IA-32 and EM64T (x86_64) processors, and the current family members are among the most widely-used in the world for general-purpose computing. Other chip vendors will have to offer similar facilities to remain competitive.

10 Problems with IEEE 754 arithmetic

Despite the many benefits of IEEE 754 floating-point arithmetic, there are many impediments to its effective use:

- Language access to features has been slow: more than 27 years have passed since the Intel 8087, and we are still waiting!
- Programmer unfamiliarity, ignorance, and inexperience.
- A deficient educational system, both in academia, and in textbooks, leaves most programmers with little or no training in floating-point arithmetic.
- Partial implementations by some vendors deny access to important features (e.g., subnormals may flush to zero, IA-32 has only one NaN, IA-32 and IA-64 have imperfect rounding, Java and C# lack rounding modes and higher precisions).
- Long internal registers are generally beneficial, but also produce many computational surprises and double rounding [33], compromising portability.
- Rounding behavior at underflow and overflow limits is unspecified by the IEEE standards, and thus, is vendor dependent.
- Overeager, or incorrect, optimizations by compilers may produce wrong results, and prevent obtaining similar results across different platforms, or between different compilers on the same system, or even from the same compiler with different options.
- Despite decades of availability of IEEE 754 arithmetic, some compilers still mishandle signed zeros and NaNs, and it can be difficult to convince compiler vendors of the significance of such errors (I know, because I've tried, and failed).

11 How decimal arithmetic is different

Programmers in science and engineering have usually only had experience with binary floating-point arithmetic, and some relearning is needed for the move to decimal arithmetic:

- Nonzero normal floating-point numbers take the form $x = (-1)^s f \times 10^p$, where f is an *integer*, allowing simulation of fixed-point arithmetic.
- Lack of normalization means multiple storage forms, but 1., 1.0, 1.00, 1.000, … compare equal, as long as floating-point instructions, rather than bitwise integer comparisons, are used.
- *Quantization* is detectable (e.g., for financial computations, 1.00 differs from 1.000).
- Signed zero and infinity, plus quiet and signaling NaNs, are detectable from the first byte, whereas binary formats require examination of all bits.
- There are *eight* rounding modes because of legal and tax mandates.
- Compact storage formats — Densely-Packed Decimal (DPD) [IBM] and Binary-Integer Decimal (BID) [Intel] — need fewer than BCD's four bits per decimal digit.

12 Software floating-point arithmetic

It may be better in some applications to have floating-point arithmetic entirely in software, as Apple once did with the no-longer-supported SANE (Standard Apple Numerics Environment) system. Here are some reasons why:

- TeX and METAFONT must continue to guarantee identical results across platforms.
- Unspecified behavior of low-level arithmetic guarantees *platform dependence*.
- Floating-point arithmetic is not associative, so instruction ordering (e.g., compiler optimization) affects results.

- Long internal registers on some platforms, and not on others, alter precision, and results.
- Multiply-add computes $x \times y + z$ with *exact* product and single rounding, getting different result from separate operations.
- Conclusion: only a single *software* floating-point arithmetic system in TEX and METAFONT can guarantee *platform-independent results*.

Software is often best enhanced by connecting two or more systems with a clean and simple interface:

> What if you could provide a seamlessly integrated, fully dynamic language with a conventional syntax while increasing your application's size by less than 200K on an x86? You can do it with *Lua*!
>
> Keith Fieldhouse

If we want to have floating-point arithmetic in TEX and METAFONT, then rather than modify those stable and reliable programs, including adding convenient expression syntax, and a substantial function library, there is a cleaner, and easier, approach:

- There is no need to modify TEX beyond what has already been done: LuaTEX interfaces TEX to a clean and well-designed scripting language — we just need to change the arithmetic and library inside `lua`.
- Scripting languages usually offer a single floating-point datatype, typically equivalent to IEEE 754 64-bit `double` (that is all that the C language used to have).
- `qawk` and `dnawk` are existing extensions by the author of `awk` for 128-bit binary and decimal arithmetic, respectively.
- Modern machines are fast and memories are big. We could adopt a 34D 128-bit format, or better, a 70D 256-bit format, instead as default numeric type.
- The author's `mathcw` package [5] is a highly-portable open-source library with support for *ten* floating-point precisions, including 256-bit binary and decimal.

Two more quotes from the father of the IEEE 754 design lead into our next points:

> The convenient accessibility of double-precision in many Fortran and some Algol compilers indicates that double-precision will soon be universally acceptable as a substitute for ingenuity in the solution of numerical problems.
>
> W. Kahan
> *Further Remarks on Reducing Truncation Errors*
> Comm. ACM **8**(1) 40, January (1965)

> Nobody knows how much it would cost to compute y^w correctly rounded for every two floating-point arguments at which it does not over/underflow. Instead, reputable math libraries compute elementary transcendental functions mostly within slightly more than half an ulp and almost always well within one ulp. Why can't y^w be rounded within half an ulp like SQRT? Because nobody knows how much computation it would cost.... No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits. Even the fact (if true) that a finite number of extra digits will ultimately suffice may be a deep theorem.
>
> W. Kahan
> *Wikipedia entry*

We need more than just the basic four operations of arithmetic: several dozen elementary functions, and I/O support, are essential.

- The *Table Maker's Dilemma* (Kahan) is the problem of always getting exactly-rounded results when computing the elementary functions. Here is an example of a hard case:
  `log(+0x1.ac50b409c8aeep+8) =`
  `0x60f52f37aecfcffffffffffffffeb...p-200`
  There are 62 consecutive 1-bits in that number, and at least $4 \times 13 + 62 + 1 = 115$ bits must be computed correctly in order to determine the correctly-rounded 53-bit result.

- Higher-than-needed-precision arithmetic provides a practical solution to the dilemma, as the Kahan quote observes.

- Random-number generation is a common portability problem, since algorithms for that computation are platform-dependent and vary in quality. Fortunately, several good ones are now known, and can be supplied in libraries, although careful attention still needs to be given to computer wordsize.

- The `mathcw` library gives *platform-independent* results for decimal floating-point arithmetic, since evaluation order is completely under programmer control, and identical everywhere, and the underlying decimal arithmetic is too.

13 How much work is needed?

I argue that decimal floating-point arithmetic in software, isolated in a separate scripting language, is an effective and reasonable way to extend TEX and METAFONT so that they can have access to floating-point arithmetic, and remove the limitations and nuisance of fixed-point arithmetic that they currently suffer.

It is therefore appropriate to ask what kind of effort would be needed to do this. In four separate experiments with three implementations of `awk`, and one of `lua`, I took two to four hours each, with less than 3% of the code requiring changes:

Program	Lines	Deleted	Added
`dgawk`	40 717	109	165
`dlua`	16 882	25	94
`dmawk`	16 275	73	386
`dnawk`	9 478	182	296
METAFONT in C	30 190	0	0
TEX in C	25 215	0	0

14 Summary

Had the IEEE 754 design been developed before TEX and METAFONT, it is possible that Donald Knuth would have chosen a software implementation of binary floating-point arithmetic, as he later provided for the MMIX virtual machine [2, 31, 32] that underlies the software analyses in newer editions of his famous book series, *The Art of Computer Programming*.

That did not happen, so in this article, I have shown how a different approach might introduce *decimal* floating-point arithmetic to TEX and METAFONT through a suitable scripting language, for which Lua [26, 25, 27] seems eminently suited, and has already been interfaced to TEX and is now in limited use for production commercial typesetting, and also for document style-file design. By selecting high working precision, at least 34 decimal digits and preferably 70, many numerical issues that otherwise compromise portability and reproducibility of typeset documents simply disappear, or at least, become highly improbable.

To make this workable, the compilers, the basic software arithmetic library, the elementary function library, and the I/O library need to be highly portable. The combination of the GNU `gcc` compiler family with the IBM `decNumber` library and the author's `mathcw` library satisfy all of these requirements. Within a year or two, we may therefore expect that decimal floating-point arithmetic in C could be available on all of the common platforms, allowing future TEX Live releases to build upon that foundation, and LuaTEX could become the TEX version of choice in many environments. LuaMETAFONT and LuaMETAPOST could soon follow.

References

[1] P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White. Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE floating-point arithmetic. *IBM Journal of Research and Development*, 43(5/6):723–760, 1999. ISSN 0018-8646. URL `http://www.research.ibm.com/journal/rd/435/abbott.html`. Besides important history of the development of the S/360 floating-point architecture, this paper has a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones [39, 7, 29, 6, 40].

[2] Heidi Anlauff, Axel Böttcher, and Martin Ruckert. *Das MMIX-Buch: ein praxisnaher Zugang zur Informatik. (German) [The MMIX Book: A practical introduction to computer science]*. Springer-Lehrbuch. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2002. ISBN 3-540-42408-3. xiv + 327 pp. EUR 24.95. URL `http://www.informatik.fh-muenchen.de/~mmix/MMIXBuch/`.

[3] Nelson Beebe. The design of TEX and METAFONT: A retrospective. *TUGboat*, 26(1):33–41, 2005. ISSN 0896-3207.

[4] Nelson H. F. Beebe. 25 Years of TEX and METAFONT: Looking back and looking forward — TUG 2003 keynote address. *TUGboat*, 25(1):7–30, 2004. ISSN 0896-3207.

[5] Nelson H. F. Beebe. *The `mathcw` Portable Elementary Function Library*. 2008. In preparation.

[6] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. *ACM SIGPLAN Notices*, 31(5):108–116, May 1996. ISSN 0362-1340. URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/231379/p108-burger/`. This paper offers a significantly faster algorithm than that of [39], together with a correctness proof and an implementation in Scheme. See also [7, 1, 40, 8].

[7] William D. Clinger. How to read floating point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, June 1990. ISBN 0-89791-364-7. ISSN 0362-1340. URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p92-clinger/`. See also output algorithms in [29, 39, 6, 1, 40].

[8] William D. Clinger. Retrospective: How to read floating point numbers accurately. *ACM SIGPLAN Notices*, 39(4):360–371, April 2004. ISSN 0362-1340. Best of PLDI 1979–1999. Reprint of, and retrospective on, [7].

[9] William J. Cody, Jr. Analysis of proposals for the floating-point standard. *Computer*, 14(3):63–69, March 1981. ISSN 0018-9162. See [23, 24].

[10] Jerome T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, 13(1):68–79, January 1980. ISSN 0018-9162. See errata in [11]. See [23, 24].

[11] Jerome T. Coonen. Errata: An implementation guide to a proposed standard for floating point arithmetic. *Computer*, 14(3):62, March 1981. ISSN 0018-9162. See [10, 23, 24].

[12] Jerome T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981. ISSN 0018-9162. See [23, 24].

[13] Charles B. Dunham. Surveyor's Forum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 24(3):319, September 1992. ISSN 0360-0300. See [15, 16, 45].

[14] W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1990. ISBN 0-387-97299-4. xix + 453 pp. LCCN QA76 .B326 1990.

[15] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. ISSN 0360-0300. URL `http://www.acm.org/pubs/toc/Abstracts/0360-0300/103163.html`. See also [16, 13, 45].

[16] David Goldberg. Corrigendum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 23(3):413, September 1991. ISSN 0360-0300. See [15, 13, 45].

[17] David Goldberg. Computer arithmetic. In *Computer Architecture—A Quantitative Approach*, chapter H, pages H–1–H–74. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, third edition, 2002. ISBN 1-55860-596-7. LCCN QA76.9.A73 P377 2003. US$89.95. URL `http://books.elsevier.com/companions/1558605967/appendices/1558605967-appendix-h.pdf`. The complete Appendix H is not in the printed book; it is available only at the book's Web site: `http://www.mkp.com/CA3`.

[18] David Gries. Binary to decimal, one more time. In Feijen et al. [14], chapter 16, pages 141–148. ISBN 0-387-97299-4. LCCN QA76 .B326 1990. This paper presents an alternate proof of Knuth's algorithm [29] for conversion between decimal and fixed-point binary numbers.

[19] David Hough. Applications of the proposed IEEE-754 standard for floating point arithmetic. *Computer*, 14 (3):70–74, March 1981. ISSN 0018-9162. See [23, 24].

[20] IEEE. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1985. ISSN 0362-1340. See [23].

[21] IEEE. *854-1987 (R1994) IEEE Standard for Radix-Independent Floating-Point Arithmetic*. IEEE, New York, NY, USA, 1987. ISBN 1-55937-859-X. 16 pp. US$44.00. URL `http://standards.ieee.org/reading/ieee/std_public/description/busarch/854-1987_desc.html`. Revised 1994.

[22] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1985. 18 pp. See [23].

[23] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 12, 1985. ISBN 1-55937-653-8. 20 pp. US$35.00. URL `http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html`; `http://standards.ieee.org/reading/ieee/std/busarch/754-1985.pdf`; `http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=cat-det.p&wartnum=019113`; `http://ieeexplore.ieee.org/iel1/2355/1316/00030711.pdf`. Revised 1990. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [9, 12, 10, 11, 19, 42, 43]. The final version was republished in [20, 22]. See also [44]. Also standardized as *IEC 60559 (1989-01) Binary floating-point arithmetic for microprocessor systems*.

[24] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [9, 12, 10, 11, 19, 42, 43]. Available from the IEEE Service Center, Piscataway, NJ, USA.

[25] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, Rio de Janeiro, Brazil, 2006. ISBN 85-903798-2-5. 328 (est.) pp.

[26] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.Org, Rio de Janeiro, Brazil, 2006. ISBN 85-903798-3-3. 112 (est.) pp.

[27] Kurt Jung and Aaron Brown. *Beginning Lua programming*. Wiley, New York, NY, USA, 2007. ISBN (paperback), 0-470-06917-1 (paperback). 644 (est.) pp. LCCN QA76.73.L82 J96 2007. URL `http://www.loc.gov/catdir/toc/ecip074/2006036460.html`.

[28] W. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. Technical report, Department of Mathematics and Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA, June 18, 1998. 80 pp. URL `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`; `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.ps`.

[29] Donald E. Knuth. A simple program whose proof isn't. In Feijen et al. [14], chapter 27, pages 233–242. ISBN 0-387-97299-4. LCCN QA76 .B326 1990.

This paper discusses the algorithm used in TeX for converting between decimal and scaled fixed-point binary values, and for guaranteeing a minimum number of digits in the decimal representation. See also [7, 8] for decimal to binary conversion, [39, 40] for binary to decimal conversion, and [18] for an alternate proof of Knuth's algorithm.

[30] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997. ISBN 0-201-89684-2. xiii + 762 pp. LCCN QA76.6 .K64 1997. US$52.75.

[31] Donald Ervin Knuth. *MMIXware: A RISC computer for the third millennium*, volume 1750 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. ISBN 3-540-66938-8 (softcover). ISSN 0302-9743. viii + 550 pp. LCCN QA76.9.A73 K62 1999.

[32] Donald Ervin Knuth. *The art of computer programming: Volume 1, Fascicle 1. MMIX, a RISC computer for the new millennium*. Addison-Wesley, Reading, MA, USA, 2005. ISBN 0-201-85392-2. 134 pp. LCCN QA76.6 .K64 2005.

[33] David Monniaux. The pitfalls of verifying floating-point computations. Technical report HAL-00128124, CNRS/École Normale Supérieure, 45, rue d'Ulm 75230 Paris cedex 5, France, June 29, 2007. 44 pp. URL `http://hal.archives-ouvertes.fr/docs/00/15/88/63/PDF/floating-point.pdf`.

[34] Amos R. Omondi. *Computer Arithmetic Systems: Algorithms, Architecture, and Implementation*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1994. ISBN 0-13-334301-4. xvi + 520 pp. LCCN QA76.9.C62 O46 1994. US$40.00.

[35] Michael 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. xiv + 104 pp. LCCN QA76.9.M35 O94 2001. US$40.00. URL `http://www.cs.nyu.edu/cs/faculty/overton/book/`; `http://www.siam.org/catalog/mcc07/ot76.htm`.

[36] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Walton Street, Oxford OX2 6DP, UK, 2000. ISBN 0-19-512583-5. xx + 490 pp. LCCN QA76.9.C62P37 1999. US$85.00.

[37] C. Severance. An interview with the old man of floating-point: Reminiscences elicited from William Kahan. World-Wide Web document., 1998. URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`. A shortened version appears in [38].

[38] Charles Severance. Standards: IEEE 754: An interview with William Kahan. *Computer*, 31(3): 114–115, March 1998. ISSN 0018-9162. URL `http://pdf.computer.org/co/books/co1998/pdf/r3114.pdf`.

[39] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):112–126, June 1990. ISSN 0362-1340. See also input algorithm in [7, 8], and a faster output algorithm in [6] and [29], IBM S/360 algorithms in [1] for both IEEE 754 and S/360 formats, and a twenty-year retrospective [40]. In electronic mail dated Wed, 27 Jun 1990 11:55:36 EDT, Guy Steele reported that an intrepid pre-SIGPLAN 90 conference implementation of what is stated in the paper revealed 3 mistakes:

1. Table 5 (page 124):
   insert `k <-- 0` after assertion, and also delete `k <-- 0` from Table 6.
2. Table 9 (page 125):
   for `-1:USER!("");`
   substitute `-1:USER!("0");`
   and delete the comment.
3. Table 10 (page 125):
   for `fill(-k, "0")`
   substitute `fill(-k-1, "0")`

[40] Guy L. Steele Jr. and Jon L. White. Retrospective: How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 39(4):372–389, April 2004. ISSN 0362-1340. Best of PLDI 1979–1999. Reprint of, and retrospective on, [39].

[41] Pat H. Sterbenz. *Floating-point computation*. Prentice-Hall series in automatic computation. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1973. ISBN 0-13-322495-3. xiv + 316 pp. LCCN QA76.8.I12 S77 1974.

[42] David Stevenson. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):51–62, March 1981. ISSN 0018-9162. See [23, 24].

[43] David Stevenson. *A proposed standard for binary floating-point arithmetic: draft 8.0 of IEEE Task P754*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1981. 36 pp. See [23, 24].

[44] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Reinhart, and Winston, New York, NY, USA, 1982. ISBN 0-03-060571-7. xvii + 308 pp. LCCN TK7895 A65 W37 1982. Master copy output on Alphatype CRS high-resolution phototypesetter. This book went to press while the IEEE 754 Floating-Point Standard was still in development; consequently, some of the material on that system was invalidated by the final Standard (1985) [23].

[45] Brian A. Wichmann. Surveyor's Forum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 24(3):319, September 1992. ISSN 0360-0300. See [15, 16, 13].