Extending TEX and METAFONT with Floating-Point Arithmetic

Nelson H. F. Beebe

Department of Mathematics

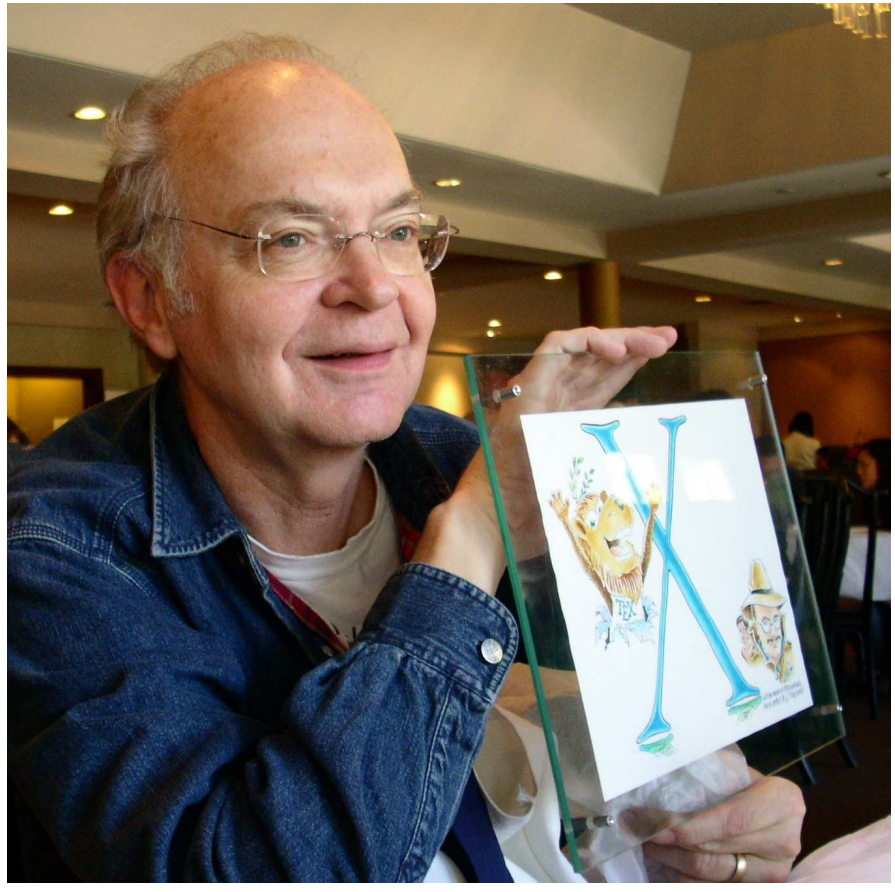
University of Utah

Salt Lake City, UT 84112-0090

USA

Dedication

Professor **Donald Knuth** (Stanford)
Professor **William Kahan** (Berkeley)



Arithmetic in TEX and METAFONT

- Binary integer arithmetic with $\geq$ 32 bits (TEX `\count` registers)
- Fixed-point arithmetic with sign bit, overflow bit, $\geq$ 14 integer bits, and 16 fractional bits (TEX `\dimen`, `\muskip`, and `\skip` registers)
- Overflow detected on division and multiplication but not on addition (flaw (NHFB), feature (DEK))
- Gyrations sometimes needed in METAFONT to work with fixed-point numbers

```
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.
```

Arithmetic in METAFONT

METAFONT restricts input numbers to 12 integer bits:

```
% 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
```

Historical remarks

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)

Historical remarks [cont]

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)

Historical remarks [cont]

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)

Why no floating-point arithmetic?

- System dependence in *precision*, *range*, *rounding*, *underflow*, *overflow*
- Base varies: 2, 3 (Setun), 4 (Illiac II), 8 (Burroughs), 10, 16 (IBM S/360), 256 (Illiac III), 10000 (Maple)
- Bizarre behavior when TEX was developed:
  - $x \times y \neq y \times x$ (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 (PDP-10)
  - job termination on overflow or zero-divide (most)
- No standardization: almost every vendor had unique floating-point system

Why no floating-point . . . [cont]?

- Language dependence:
  - Algol, Pascal, and SAIL (`real`)
  - Fortran (`REAL`, `DOUBLE PRECISION`, and sometimes `REAL*16`)
  - C/C++ (`double`, `float` added in 1989, `long double` in 1999)
  - Java and C# (only `float` and `double`, but arithmetic system is badly botched: see Kahan and Darcy's ***How Java's Floating-Point Hurts Everyone Everywhere***)
- Compiler dependence: multiple precisions mapped to just one
- BSD compilers still provide no 80-bit format after 27 years in hardware

Why no floating-point . . . [cont]?

- 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)
- DEK wrote ***A simple program whose proof isn't*** (1990) about TEX's conversions between fixed-point binary and decimal
- Most languages do not guarantee exact base conversion
- TEX guarantees identical line-breaking and page-breaking across all platforms (floating-point arithmetic used only for interword glue calculations)
- METAFONT has no floating-point at all, and generates identical fonts on all systems

IEEE 754 binary standard (1985)

- Preliminary version first implemented in Intel 8087 chip (1980)
- Three formats defined: 32-bit, 64-bit, and 80-bit. 128-bit format available on some Alpha, IA-64, PA-RISC, and SPARC systems.
- Nonzero normal numbers are *rational*: $x = (-1)^s f \times 2^p$, where $f \in [1, 2)$
- Signed zero
- Largest stored exponent represents Infinity when $f = 0$, quiet and signaling NaN (Not-a-Number) when $f \neq 0$
- Smallest stored exponent allows f to have leading zeros with *gradual underflow* to *subnormal* values

IEEE 754 binary standard [cont]

- Nonstop computing model: sticky flags record exceptions
- Four rounding modes:
  - *to nearest with ties to even* (default)
  - *to* $+\infty$
  - *to* $-\infty$
  - *to zero* (historical chopping)
- $\pm\infty$ generated from large/small and finite/0
- NaN generated from 0/0, $\infty - \infty$, ∞/∞, and any operation with a NaN operand
- NaN returned from functions when result is undefined in real arithmetic (e.g., $\sqrt{-1}$)

IEEE 754R Precision and range

Binary				
32-bit	24b (≈ 7D)	1e-45	1e-38	3e+38
64-bit	53b (≈ 15D)	4e-324	2e-308	1e+308
80-bit	64b (≈ 19D)	3e-4951	3e-4932	1e+4932
128-bit	113b (≈ 34D)	6e-4966	3e-4932	1e+4932
256-bit	234b (≈ 70D)	2e-315 723	5e-315 653	3e+315 652
Decimal				
32-bit	7D	1e-101	1e-95	1e+96
64-bit	16D	1e-398	1e-383	1e+384
128-bit	34D	1e-6176	1e-6143	1e+6144
256-bit	70D	1e-1 572 932	1e-1 572 863	1e+1 572 864

Remarks on floating-point arithmetic

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

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

Bases other than 2 or 10 suffer from wobbling precision: in hexadecimal arithmetic, $\pi/2 \approx 1.571 \approx 1.922_{16}$ has 3 fewer bits (almost one decimal digit) than $\pi/4 \approx 0.7854 \approx \mathrm{c}.910_{16}$.

Binary versus decimal

- humans less uncomfortable with decimal arithmetic
- sales tax: 5% of $0.70 = 0.0349999\ldots$ in *all* binary precisions, instead of exact decimal 0.035. Thus, significant cumulative rounding errors in businesses with many small transactions (food, telephone, . . .)
- financial computations need fixed-point decimal arithmetic
- hand calculators use decimal arithmetic
- additional decimal rounding rules (8 instead of 4)
- decimal arithmetic eliminates most base-conversion problems

Binary versus decimal [cont]

- ***IEEE 854 Standard for Radix-Independent Floating-Point Arithmetic*** (1987, 1994)
- 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++ (2005, 2006)
- 25 years of Rexx and NetRexx scripting languages give valuable experience in arbitrary-precision decimal arithmetic
- 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

Binary versus decimal [cont]

- Preliminary support in `gcc` for $+$, $-$, $\times$, and $/$ (late 2006) based on subset of IBM `decNumber` library
- `mathcw` package provides C99-compliant run-time library for binary, and also for decimal, arithmetic (NHFB 2005–2007)
- Three sizes defined for IEEE 754R: 32-bit (7D), 64-bit (16D), and 128-bit (34D)
- IBM zSeries mainframes get IEEE 754 binary f.p. (1999), and decimal f.p. in firmware (2006)
- IBM PowerPC chips add hardware decimal arithmetic (21 May 2007)
- Hardware support likely in future Intel IA-32 and EM64T (x86_64)

Problems with IEEE 754 arithmetic

- Language access to features slow: 27+ years and still waiting!
- Programmer unfamiliarity, ignorance, and inexperience
- Deficient educational system
- Partial implementations by some vendors (e.g., subnormals 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 generally beneficial, but also produce many computational surprises and double rounding, compromising portability
- Rounding behavior at underflow and overflow limits unspecified, and vendor dependent

How decimal arithmetic is different

- Nonzero normal numbers $x = (-1)^s f \times 2^p$, where f is an *integer*: can simulate fixed-point arithmetic
- Lack of normalization means multiple storage forms, but 1., 1.0, 1.00, 1.000, . . . compare equal
- *Quantization* detectable (e.g., for financial computations, 1.00 differs from 1.000)
- Signed zero and Infinity, plus quiet and signaling NaNs detectable from first byte (binary formats require examination of all bits)
- Eight rounding modes (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

Software floating-point arithmetic

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

Side excursion

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

Software floating-point [cont.]

- No need to modify TEX beyond what has already been done: LuaTEX interfaces TEX to a clean and well-designed scripting language — just need to change 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 C used to have)
- `qawk` and `dnawk`: `awk` for 128-bit binary and decimal
- Machines are fast and memories are big: adopt 34D 128-bit format, or better, 70D 256-bit format, instead as default numeric type.
- `mathcw` has highly-portable open-source library support for *ten* floating-point precisions, including 256-bit binary and decimal.

Software floating-point [cont.]

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)

Software floating-point [cont.]

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

Software floating-point [cont.]

- Table Maker's Dilemma (Kahan) is the problem of always getting exactly-rounded results when computing the elementary functions:
  log(+0x1.ac50b409c8aeep+8) = 0x60f52f37aecfcfffffffffffffeb...p−200
  (62 consecutive 1's)
- Higher-than-needed-precision arithmetic provides a practical solution, as Kahan quote observes
- Random-number generation is a portability problem, since algorithms are platform-dependent and vary in quality
- **mathcw** library provides *platform-independent* results for decimal floating-point arithmetic

How much work?

Changing scripting languages from binary to decimal floating-point arithmetic took two to four hours each, with relatively few modifications:

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

Floating-point arithmetic and typesetting

- TEX's smallest dimension is 2^{-16}pt $= 1$sp, while wavelength of visible light is about 100sp (TEXbook, p. 58): rounding errors are *invisible*
- TEX's largest dimension is 2^{14}pt $= 5.75$m, not quite billboard size
- macro notation painful (`layout.tex`):

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

- reduction $75/100 \longrightarrow 1/4 \times 3$ possible here, but not in general

Floating-point arithmetic [cont]

- Overflow detection unreliable in TEX (integer arithmetic in most programming languages is worse!)
- No elementary functions available in TEX, not even square root
- METAFONT offers ++ (Pythagoras), `abs`, `angle`, `ceiling`, `cosd`, `dir`, `floor`, `length`, `mexp`, `mlog`, `normaldeviate`, `round`, `sind`, `sqrt`, and `uniformdeviate`
- Floating-point simplifies computation of fractions, scaling, and rotation: see LATEX `calc` package for horrors of fixed-point arithmetic
- Interface from TEX to scripting language allows conventional numeric programming

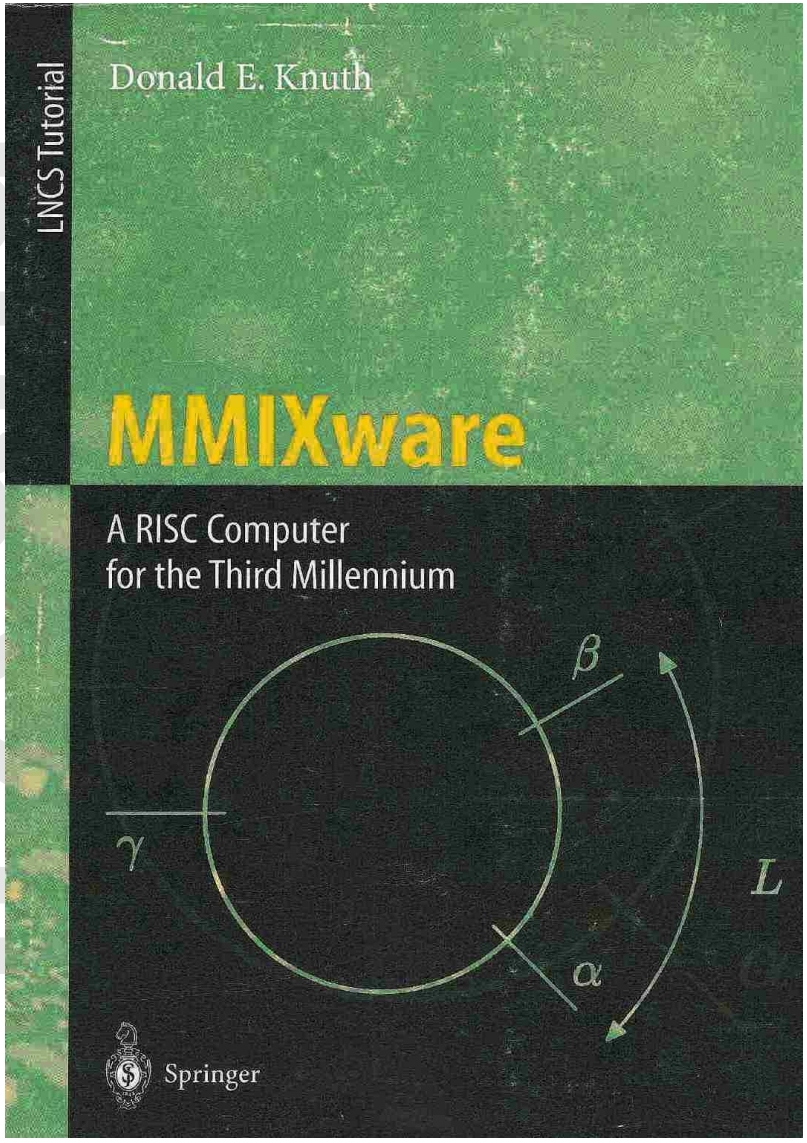
MMIX and NNIX

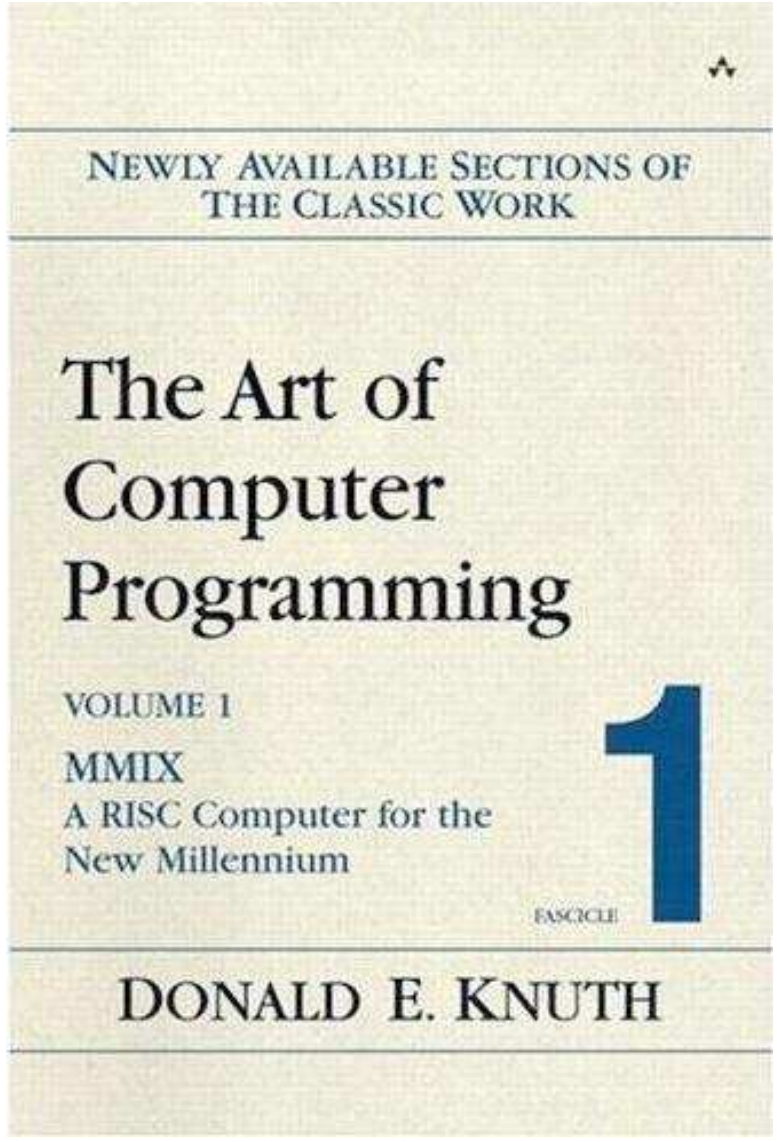
Whenever anybody has asked if I will be writing a book about operating systems, my reply has always been “Nix.” Therefore the name of MMIX’s operating system, NNIX, should come as no surprise.

Donald E. Knuth

MMIXware:
A RISC Computer for the Third Millenium

MMIX and NNIX [cont]





MMIX and NNIX [cont]

MMIX and NNIX [cont]

- MMIX is a modern virtual machine used in recent volumes of DEK's famous series ***The Art of Computer Programming***
- MMIX is written as *literate program* in published books
- arithmetic is IEEE 754 64-bit binary in software using only 32-bit unsigned integers
- 17 floating-point instructions: `FADD`, `FCMP`, `FCMPE`, `FDIV`, `FEQL`, `FEQLE`, `FINT`, `FIX`, `FIXU`, `FLOT`, `FLOTU`, `FMUL`, `FREM`, `FSQRT`, `FSUB`, `FUN`, and `FUNE`
- `gcc` version 3.2 can be built for MMIX system
- NNIX is a (still virtual) Unix-like O/S for MMIX
- `mathcw` library port to MMIX: 10 new + 12 header bug workaround, out of 250,000 lines

The End

THE BEATLES
JULY/AUGUST 1969
[1969 = YEAR OF FIRST EDITION OF
DONALD KNUTH'S
Seminumerical Algorithms
(*TAOCP* VOLUME 2)]