A new math library

Nelson H. F. Beebe

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

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

20 March 2009

Common misconceptions about computer arithmetic

- Integer arithmetic is always exact
- Integer overflows are caught
- Floating-point arithmetic is *fuzzy*
- Floating-point equality comparisons are unreliable
- Floating-point precision and range are adequate for everyone
- Rounding errors accumulate
- Computers execute arithmetic code in the order and precision in which it is written
- Underflows are harmless
- Overflows are disastrous
- Sign of zero does not matter
- Arithmetic exceptions should cause job termination

Historical floating-point arithmetic

- Konrad Zuse's Z1, Z3, and Z4 (1936–1945): 22-bit (Z1 and Z3) and 32-bit Z4 with exponent range of $2^{\pm 63} \approx 10^{\pm 19}$
- Burks, Goldstine, and von Neumann (1946) argued against floating-point arithmetic
- *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 (1980)
- IBM mainframes from mid-1950s supplied floating-point arithmetic
- IEEE 754 Standard (1985) proposed a new design for binary floating-point arithmetic that has since been widely adopted
- IEEE 754 design first implemented in Intel 8087 coprocessor (1980)

Historical flaws on some systems

Floating-point arithmetic can make error analysis difficult, with behavior like this in some older designs:

- $u \neq 1.0 \times u$
- $u + u \neq 2.0 \times u$
- $u \times 0.5 \neq u/2.0$
- $u \neq v$ but $u - v = 0.0$, and $1.0/(u - v)$ raises a zero-divide error
- $u \neq 0.0$ but $1.0/u$ raises a zero-divide error
- $u \times v \neq v \times u$
- underflow wraps to overflow, and vice versa
- division replaced by reciprocal approximation and multiply
- poor rounding practices increase cumulative rounding error

IEEE 754 binary floating-point arithmetic

	s	exp	significand	
bit	0	1 … 9	… 31	single
	0	1 … 12	… 63	double
	0	1 … 16	… 79	extended
	0	1 … 16	… 127	quadruple
	0	1 … 22	… 255	octuple

- *s* is sign bit (0 for +, 1 for −)

IEEE 754 binary floating-point arithmetic

	s	exp		significand	
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field

IEEE 754 binary floating-point arithmetic

s	exp	significand

bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field
- smallest exponent: zero and *subnormals* (formerly, *denormalized*)

IEEE 754 binary floating-point arithmetic

s	exp	significand		

bit					
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field
- smallest exponent: zero and *subnormals* (formerly, *denormalized*)
- largest exponent: Infinity and NaN (Not a Number)

IEEE 754 binary floating-point arithmetic

	s	exp	significand		
bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field
- smallest exponent: zero and *subnormals* (formerly, *denormalized*)
- largest exponent: Infinity and NaN (Not a Number)
- significand has implicit leading 1-bit in all but 80-bit format

IEEE 754 binary floating-point arithmetic

s	exp	significand

bit	0	1	9	31	single
	0	1	12	63	double
	0	1	16	79	extended
	0	1	16	127	quadruple
	0	1	22	255	octuple

- *s* is sign bit (0 for $+$, 1 for $-$)
- *exp* is unsigned biased exponent field
- smallest exponent: zero and *subnormals* (formerly, *denormalized*)
- largest exponent: Infinity and NaN (Not a Number)
- significand has implicit leading 1-bit in all but 80-bit format
- ± 0, $\pm\infty$, signaling and quiet NaN

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, x op NaN, ...

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{\ op\ NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~} \mathrm{NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero

IEEE 754 binary floating-point arithmetic

- NaN from 0/0, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{\ op\ NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero
- precisions in bits: 24, 53, 64, 113, 235

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{\ op\ NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero
- precisions in bits: 24, 53, 64, 113, 235
- approximate precisions in decimal digits: 7, 15, 19, 34, 70

IEEE 754 binary floating-point arithmetic

- NaN from $0/0$, $\infty - \infty$, $f(\mathrm{NaN})$, $x \mathrm{~op~} \mathrm{NaN}$, ...
- NaN $\neq$ NaN is distinguishing property, but botched by 10% of compilers
- $\pm\infty$ from big/small, including nonzero/zero
- precisions in bits: 24, 53, 64, 113, 235
- approximate precisions in decimal digits: 7, 15, 19, 34, 70
- approximate ranges (powers of 10): $[-45, 38]$, $[-324, 308]$, $[-4951, 4932]$, $[4966, 4932]$, $[-315\,723, 315\,652]$

IEEE 754 binary floating-point arithmetic

- nonstop computing model

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: $\boxed{\text{underflow}}$, $\boxed{\text{overflow}}$, $\boxed{\text{zero divide}}$, $\boxed{\text{invalid}}$, and $\boxed{\text{inexact}}$

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard
- some architectures implement only subsets (e.g., no subnormals, or only one rounding mode, or only one kind of NaN, or in embedded systems, neither Infinity nor NaN)

IEEE 754 binary floating-point arithmetic

- nonstop computing model
- five sticky flags record exceptions: underflow, overflow, zero divide, invalid, and inexact
- four rounding modes: to-nearest-with-ties-to-even (default), to-plus-infinity, to-minus-infinity, and to-zero
- traps versus exceptions
- fixups in trap handlers impossible on heavily-pipelined or parallel architectures (since IBM System/360 Model 91 in 1968)
- no language support for advanced features until 1999 ISO C Standard
- some architectures implement only subsets (e.g., no subnormals, or only one rounding mode, or only one kind of NaN, or in embedded systems, neither Infinity nor NaN)
- some platforms have nonconforming rounding behavior

Why the base matters

- accuracy and run-time cost of conversion between internal and external (usually decimal) bases
- effective precision varies when the floating-point representation uses a radix larger than 2 or 10
- reducing the exponent width makes digits available for increased precision
- for a fixed number of exponent digits, larger bases provide a wider exponent range, and reduce incidence of rounding
- for a fixed storage size, granularity (the spacing between successive representable numbers) increases as the base increases
- in the absence of underflow and overflow, multiplication by a power of the base is an *exact* operation, and this feature is *essential* for many computations, in particular, for accurate elementary and special functions

Why the base matters [cont.]

Consider evaluation of $z = x/(2y)$:

- In the *binary* base, optimum form is $\mathbf{z = x/(y + y)}$.
- In a *nonbinary* base, compute $\mathbf{z = 0.5 \times (x/y)}$.

These alternatives avoid introducing unnecessary additional rounding error, and the second sacrifices speed for accuracy.

Base conversion problem

- exact in one base may be inexact in others (e.g., decimal 0.9 is hexadecimal `0x1.cccccccccccccccccccccc...p-1`)
- 5% sales-tax example: binary arithmetic: $0.70 \times 1.05 = 0.734999999\ldots$, which rounds to 0.73; correct decimal result 0.735 may round to 0.74
- Goldberg (1967) and Matula (1968) showed how many digits needed for *exact round-trip* conversion
- exact conversion may require *many* digits: more than 11 500 decimal digits for binary-to-decimal conversion of 128-bit format,
- base-conversion problem not properly solved until 1990s
- few (if any) languages guarantee accurate base conversion

Decimal floating-point arithmetic

- Absent in most computers from mid-1960s to 2007
- IBM Rexx and NetRexx scripting languages supply decimal arithmetic with arbitrary precision (10^9 digits) and huge exponent range ($10^{\pm 999\,999\,999}$)
- IBM `decNumber` library provides portable decimal arithmetic, and leads to hardware designs in IBM zSeries (2006) and PowerPC (2007)
- GNU compilers implement low-level support in late 2006
- business processing traditionally require 18D fixed-point decimal, but COBOL 2003 mandates 32D, and requires floating-point as well
- four additional rounding modes for legal/tax/financial requirements
- *integer*, rather than *fractional*, coefficient means redundant representation, but allows emulating fixed-point arithmetic
- *quantization* primitives can distinguish between 1, 1.0, 1.00, 1.000, etc.
- trailing zeros significant: they change quantization

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1 6	9		31	single
	0	1 6	12		63	double
	0	1 6	16		127	quadruple
	0	1 6	22		255	octuple

- IBM Densely-Packed Decimal (DPD) and Intel Binary-Integer Decimal (BID) in 32-bit, 64-bit, 128-bit, and 256-bit formats provide $3n + 1$ digits: 7, 16, 34, and 70

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- IBM Densely-Packed Decimal (DPD) and Intel Binary-Integer Decimal (BID) in 32-bit, 64-bit, 128-bit, and 256-bit formats provide $3n + 1$ digits: 7, 16, 34, and 70
- wider exponent ranges in decimal than binary: $[-101, 97]$, $[-398, 385]$, $[-6176, 6145]$, and $[-1\,572\,863, 1\,572\,865]$

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- IBM Densely-Packed Decimal (DPD) and Intel Binary-Integer Decimal (BID) in 32-bit, 64-bit, 128-bit, and 256-bit formats provide $3n + 1$ digits: 7, 16, 34, and 70
- wider exponent ranges in decimal than binary: $[-101, 97]$, $[-398, 385]$, $[-6176, 6145]$, and $[-1\,572\,863, 1\,572\,865]$
- *cf* (combination field), *ec* (exponent continuation field), (cc) (coefficient combination field)

Decimal floating-point arithmetic

	s	cf	ec	cc		
bit	0	1	6	9	31	single
	0	1	6	12	63	double
	0	1	6	16	127	quadruple
	0	1	6	22	255	octuple

- IBM Densely-Packed Decimal (DPD) and Intel Binary-Integer Decimal (BID) in 32-bit, 64-bit, 128-bit, and 256-bit formats provide $3n+1$ digits: 7, 16, 34, and 70
- wider exponent ranges in decimal than binary: $[-101, 97]$, $[-398, 385]$, $[-6176, 6145]$, and $[-1\,572\,863, 1\,572\,865]$
- *cf* (combination field), *ec* (exponent continuation field), (cc) (coefficient combination field)
- Infinity and NaN recognizable from first byte (not true in binary formats)

Library problem

- Need *much* more than ADD, SUB, MUL, and DIV operations
- `mathcw` library provides full C99 repertoire, including `printf` and `scanf` families, plus hundreds more [but not functions of type `complex`]
- code is portable across all current platforms, and several historical ones (PDP-10, VAX, S/360, ...)
- supports *six* binary and *four* decimal floating-point datatypes
- separate algorithms cater to base variations: 2, 8, 10, and 16
- pair-precision functions for even higher precision
- fused multiply-add (FMA) via pair-precision arithmetic
- programming languages: Ada, C, C++, C#, Fortran, Java, Pascal
- scripting languages: gawk, hoc, lua, mawk, nawk

Whatever your figurework requirements, there's a MMIX Station exactly suited to your needs. Designed by Prof. D. E. Knuth of Stanford, this ingenious all electric machine has more than two hundred registers and is the fastest producer of useful, accurate answers just when business is needing more and more figures. Available in a broad color range.

Fused multiply-add

- $a \times b + c$ is a common operation in numerical computation (e.g., nested Horner polynomial evaluation and matrix/vector arithmetic)
- `fma(a,b,c)` computes $a \times b + c$ with *exact* double-length product and addition with *one* rounding
- `fma(a,b,c)` recovers error in multiplication:

$$\begin{aligned} \mathtt{d} &\leftarrow \mathtt{fl(a * b)} \\ \mathtt{err} &\leftarrow \mathtt{fma(a,b,-d)} \\ \mathtt{a} \times \mathtt{b} &= \mathtt{fl(a * b)} + \mathtt{err} \end{aligned}$$

- `fma()` in some native hardware [IBM PowerPC (32-bit and 64-bit only), HP/Intel IA-64 (32-bit, 64-bit, 80-bit), and some HP PA-RISC and MIPS R8000]
- `fma()` is a critical component of many algorithms for accurate computation

Fused multiply-add

- Markstein's book shows how `fma()` leads to accurate and compact elementary functions, as well as provably-correctly-rounded software division and square root
- Nievergelt [TOMS 2003] proved that `fma()` leads to matrix arithmetic provably accurate to the penultimate digit
- See `fparith.bib` for many other applications of `fma()`

The MathCW Library

```
__acs _cvtinf _cvtnan _cvtrnd _cvtrn _ipow _prd _pr _pxy _red _re _rp _rph acosdeg acos acosh acosp
acospi adx agm annuity asindeg asin asinh asinp asinpi atan2deg atan2 atan2p atandeg atan atanh
atanp atanpi bi0 bi1 bin bis0 bis1 bisn bk0 bk1 bkn bks0 bks1 bksn cad cbrt ceil chisq compound
compoun copysign cosdeg cos cosh cosp cospi cotandeg cotan cotanp cotanpi cvtia cvtib cvtid cvti
cvtig cvtih cvtio cvtod cvto cvtog cvtoi cvton cxabs cxadd cxad cxarg cxconj cxcopy cxdiv cximag
cxmul cxneg cxproj cxreal cxset cxsub dfabs dfadd dfad dfdiv dfmul dfneg dfsqrt dfsub echeb ellec
elle ellkc ellk ercw ereduce erfc erf eriduce exp10 exp10m1 exp16 exp16m1 exp2 exp2m1 exp8 exp8m1
exp expm1 fabs fdim floor fma fmax fmin fmod fmo fmul fpclassify frexp frexph frexpo gamib gamic
gami hypot ichisq ierfc ierf ilogb infty intxp iphic iphi ipow isfinite isgreater isgreaterequal
isinf isless islessequal islessgreater isnan isnormal isqnan issnan issubnormal isunordered
isunordere j0 j1 jn ldexp ldexph ldexpo lgamma lgamma_r llrint llround llroun log101p log10 log161p
log16 log1p log21p log21p log2 log81p log8 logb log lrcw lrint lround lroun mchep modf nan nearbyint
nextafter nexttoward nexttowar normalize nrcw ntos pabs pacos pacosh padd pad pasin pasinh patan2
patan patanh pcbrt pcmp pcon pcopy pcopysign pcos pcosh pcotan pdiv pdot peps peval pexp10 pexp16
pexp2 pexp8 pexp pexpm1 pfdim pfmax pfmin pfrexp pfrexph phic phi phigh phypot pierfc pierf pilogb
pin pinfty pipow pisinf pisnan pisqnan pissnan pldexp pldexph plog101p plog1p plogb plog plow pmul2
pmul pneg pout pow pprosum pqnan pscalbln pscalbn pset psi psignbit psiln psin psinh psnan psplit
psqrt psub psum2 psum ptan ptanh qert qnan quantize remainder remquo rint round roun rsqrt
samequantum sbi0 sbi1 sbin sbis0 sbis1 sbisn sbj0 sbj1 sbjn sbk0 sbk1 sbkn sbks0 sbks1 sbksn sby0
sby1 sbyn scalbln scalbn second secon setxp signbit sincos sincosp sincospi sindeg sin sinh sinp
sinpi snan sqrt tandeg tan tanh tanp tanpi tgamma trunc urcw1 urcw1_r urcw2 urcw2_r urcw3 urcw3_r
urcw4 urcw4_r urcw urcw_r vagm vbi vbis vbj vbk vbks vby vercw vercw_r vlrcw vlrcw_r vnrcw vnrcw_r
vsbi vsbis vsbj vsbk vsbks vsby vsum vurcw1 vurcw1_r vurcw2g vurcw2_r vurcw3 vurcw3_r vurcw4
vurcw4_r vurcw vurcw_r y0 y1 yn
```

$$\ldots \times 10 \approx 3660 \textbf{ functions} \approx \textbf{2M lines}$$

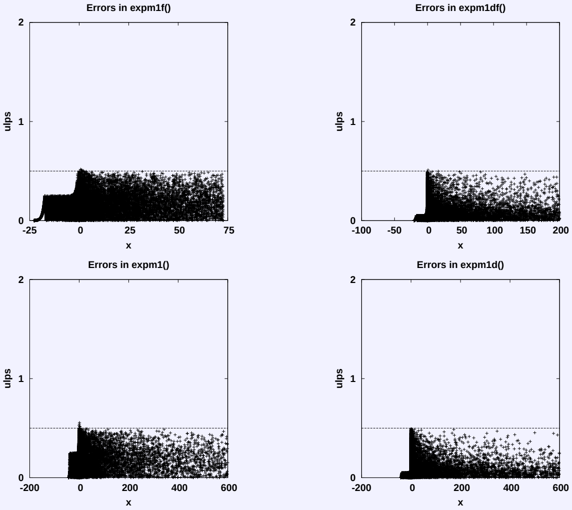
MathCW design goals

- Complete C99 and POSIX support with many enhancements
- Portable across past, present, and future platforms
- Binary *and* decimal arithmetic fully supported
- *Ten* floating-point formats, including single (7D), double (16D), quadruple (34D), and octuple precision (70D)
- IEEE 754 (1985 and 2008) and 854 feature access
- Free software and documentation under GNU licenses
- Documented in manual pages and forthcoming treatise
- Interactive access in `hoc`
- Interfaces to Ada, C, C++, C#, Fortran, Java, Pascal
- Replace native binary arithmetic in all scripting languages with high-precision decimal arithmetic

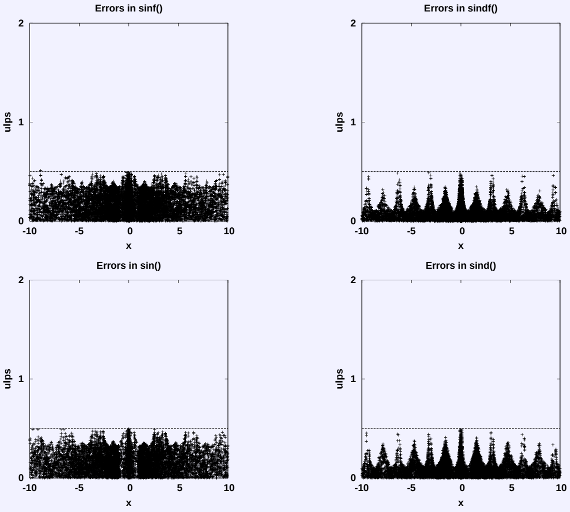
MathCW design goals [cont.]

- Separate data from code
- Abstract data types: `fp_t` and `hp_t`, and `FP()` and `FUNC()` wrappers
- Make algorithm files base-, precision-, and range-independent when feasible
- No platform software configuration needed
- Offer static, shared, fat (multi-architecture), and wrapper libraries
- Provide high relative accuracy: target is two *ulps* (units in the last place), but exponential, log, root, and trigonometric families return results that are (almost) always correctly rounded (and much better than Intel IA-32 rounding of 80-bit to 64-bit results)
- Provide exact function argument reduction [Payne/Hanek at DEC (1982) and Corbett at Berkeley (1983)]

Error plots

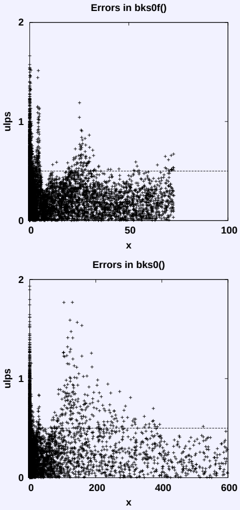

Errors in expm1f()
Errors in expm1df()
Errors in expm1()
Errors in expm1d()
ulps
x
-25
0
25
50
75
-100
-50
0
50
100
150
200
-200
0
200
400
600
0
1
2

Error plots



Errors in sinf()
Errors in sindf()
Errors in sin()
Errors in sind()
ulps
x
2
1
0
-10
-5
0
5
10

Error plots



Errors in bks0f()
ulps
x
Errors in bks0df()
ulps
x
Errors in bks0()
ulps
x
Errors in bks0d()
ulps
x

Q&A and discussion