Computational Precision and Floating-Point Arithmetic: A Teacher's Guide to Answering FAQ 7.31

Richard M. Heiberger

rmh@temple.edu

Department of Statistics
Temple University
Fox School of Business

Beginners ask questions like,

> "Why does .3 + .6 not equal .9?"

Experts always reply,

> "See FAQ 7.31."

The answer in the FAQ is probably not helpful to the Beginner.

I show several simple arithmetic and algebra statements whose *machine-calculated values* differ from *the real number system values*. I show the floating-point bit patterns for the numbers, and show why the calculated answer is correct in the floating-point system.

1 Examples

1.1 Addition

```
> (3 + 6) == 9
[1] TRUE
```

```
> (.3 + .6) == .9
[1] FALSE
```

1.2 Factoring: $(a + b) \times (a - b) = a^2 - b^2$

```
> d <- 1 + 2^-2

> (d+1)*(d-1) == d^2 - 1
[1] TRUE
```

```
> a <- 1 + 2^-27

> (a+1)*(a-1) == a^2 - 1
[1] FALSE
```

2 Representations of Numbers

2.1 Real Numbers in Base 10

Any real number can be expressed as the infinite sum

$$\pm \left(a_0 10^0 + a_1 10^{-1} + a_2 10^{-2} + \ldots + a_i 10^{-i} \right) \times 10^p$$

where p can be any integer, the values a_i are digits selected from the decimal digits $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and i is any positive integer.

For example, the decimal number 3.3125 is expressed as

$$\begin{aligned} 3.3125 &= (3 \times 10^0 + 3 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3} + 5 \times 10^{-4}) \times 10^0 \\ &= (3 \times \tfrac{1}{1} + 3 \times \tfrac{1}{10} + 1 \times \tfrac{1}{100} + 2 \times \tfrac{1}{1000} + 5 \times \tfrac{1}{10000}) \times 1 \end{aligned}$$

2.2 Finite Precision Base-10, 2-digit Arithmetic

We look at a simple example of finite-precision arithmetic with 2 significant decimal digits.

Calculate the sum of squares of three numbers in 2-digit base-10 arithmetic. For concreteness, use the example

$$2^2 + 11^2 + 15^2$$

This requires rounding to 2 significant digits at *every* intermediate step. The steps are easy. Putting your head around the steps is hard.

We rewrite the expression as a fully parenthesized algebraic expression, so we don't need to worry about precedence of operators at this step.

$$((2^2) + (11^2)) + (15^2)$$

Now we can evaluate the parenthesized groups from the inside out.

```
( (2^2)  +  (11^2) )  +  (15^2) ## parenthesized expression
(  (4)   +  (121)  )  +  (225)  ## square each term
(   4    +   120   )  +   220   ## round each term to two significant decimal digits
(       124        )  +   220   ## calculate the intermediate sum
(       120        )  +   220   ## round the intermediate sum to two decimal digits
                     340        ## sum the terms
```

Compare this to the full precision arithmetic

```
( (2^2)  +  (11^2) )  +  (15^2) ## parenthesized expression
(   4    +   121   )  +   225   ## square each term
(       125        )  +   225   ## calculate the intermediate sum
                     350        ## sum the terms
```

We see immediately that two-decimal-digit rounding at each stage gives an answer that is not the same as the one from familiar arithmetic with real numbers.

2.3 Decimal Fractions that DON'T Come Out Even

Fraction	Real Number	Two-Digit Decimal Number
1/3	0.33333333 . . .	0.33
2/3	0.66666666 . . .	0.67
1	1.00000000 . . .	1.00
1/3 + 1/3	0.66666666 . . .	0.66

2.4 Finite Precision Floating Point Numbers in Base 2

Floating point arithmetic in computers uses a finite-precision base-2 (binary) system for representation of numbers. Most computers today use the 53-bit IEEE 754 system, with numbers represented by the finite sum

$$\pm\,(a_0 \times 2^{0} + a_1 \times 2^{-1} + a_2 \times 2^{-2} + \ldots + a_{52} \times 2^{-52}) \times 2^{p}$$

where p is an integer in the range -1022 to 1023 (expressed as decimal numbers), the values a_i are digits selected from $\{0, 1\}$, and the subscripts and powers i are decimal numbers selected from $\{0, 1, \ldots, 52\}$. The decimal number 3.125_{10} is 11.0101_2 in binary.

$$\begin{aligned}
3.3125_{10} &= (1 \times 2^{0} + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}) \times 2_{10} \\
&= (1 + \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{0}{16} + \frac{1}{32}) \times 2 \\
&= \;\; 1.10101_2 \times 2_{10} \\
&= 11.0101_2
\end{aligned}$$

2.5 4-Bit Binary of Small Integers

```
> FourBits <- mpfr(matrix(0:31, 8, 4), precBits=4)
> dimnames(FourBits) <- list(0:7, c(0,8,16,24))
> FourBits
'mpfrMatrix' of dim(.) =  (8, 4) of precision  4   bits
                    > showBin(FourBits, shift=TRUE)
  0  8 16 24        0              8              16             24
0 0  8 16 24        0 +0b_____0.000 +0b__1000.___ +0b_1000_.___ +0b_1100_.___
1 1  9 16 24        1 +0b_____1.000 +0b__1001.___ +0b_1000_.___ +0b_1100_.___
2 2 10 18 26        2 +0b____10.00_ +0b__1010.___ +0b_1001_.___ +0b_1101_.___
3 3 11 20 28        3 +0b____11.00_ +0b__1011.___ +0b_1010_.___ +0b_1110_.___
4 4 12 20 28        4 +0b___100.0__ +0b__1100.___ +0b_1010_.___ +0b_1110_.___
5 5 13 20 28        5 +0b___101.0__ +0b__1101.___ +0b_1010_.___ +0b_1110_.___
6 6 14 22 30        6 +0b___110.0__ +0b__1110.___ +0b_1011_.___ +0b_1111_.___
7 7 15 24 32        7 +0b___111.0__ +0b__1111.___ +0b_1100_.___ +0b1000__.___
```

2.6 Finite Precision Floating Point Numbers in Hexadecimal Representation of Base 2

The IEEE 754 standard requires the base $\beta = 2$ number system with $p = 53$ base-2 digits.

The numbers (except for 0) in internal representation are always *normalized* with the leading bit always 1. Since the leading bit is always 1, there is no need to store it. Only 52 bits are actually needed for 53-bit precision.

A string of `0` and `1` is difficult for humans to read. Therefore every set of 4 bits is represented as a single hexadecimal digit, from the set {`0 1 2 3 4 5 6 7 8 9 a b c d e f`}, representing the decimal values {0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15}. The 52 stored bits can be displayed with 13 hex digits.

Since the base is $\beta = 2$, the exponent of an IEEE 754 floating point number must be a power of 2. The double-precision computer numbers contain 64 bits, allocated 52 for the significand, 1 for the sign, and 11 for the exponent. The 11 bits for the exponent can express $2^{11} = 2048$ unique values. These are assigned to range from 2^{-1022} to 2^{1023}.

The number 3.3125_{10} is represented in hexadecimal (base-16) notation as

$$3.3125_{10} = (1 \times 16^0 + a_{16} \times 16^{-1} + 8_{16} \times 16^{-2}) \times 2_{10}$$

$$= 1.a8_{16} \times 2_{10}$$

$= \texttt{0x1.a8p1 \#\# two hex digits after the binary point}$

$$= \left(1 + \frac{10}{16} + \frac{8}{256}\right) \times 2^1$$

$$= \left(1 + \frac{1010_2}{16} + \frac{1000_2}{256}\right) \times 2^1$$

$$= 1.10101000_2 \times 2_{10} = \texttt{0b1.10101000p1} = \texttt{0b11.0101000}$$

The R function `showHex` (based on the base R `sprintf`) specifies hexadecimal printing.

The R function `showBin` (also based on the base R `sprintf`) specifies binary printing.

```
> x <- 3.3125

> showHex(x)
[1] +0x1.a800000000000p+1

> showBin(3.3125)
[1] +0b1.1010100000000000000000000000000000000000000000000000p+1
```

3 Addition — 53 bits

Several 53 bit numbers are shown here in decimal and hexadecimal notation.

```
  decimal          decimal.17         hexadecimal
1  0.0625 0.06250000000000000 +0x1.0000000000000p-4
2  0.1000 0.10000000000000001 +0x1.999999999999ap-4   ## rounded up
3  0.3000 0.29999999999999999 +0x1.3333333333333p-2   ## rounded down
4  0.3125 0.31250000000000000 +0x1.4000000000000p-2
5  0.5000 0.50000000000000000 +0x1.0000000000000p-1
6  0.6000 0.59999999999999998 +0x1.3333333333333p-1   ## rounded down
7  0.9000 0.90000000000000002 +0x1.ccccccccccccdp-1   ## rounded up
8  1.0000 1.00000000000000000 +0x1.0000000000000p+0
9  3.3125 3.31250000000000000 +0x1.a800000000000p+1
```

3.1 Rmpfr — Multiple Precision Floating-Point Reliable — 13 bits here

```
> nums <- c(.0625, .1, .3, .3125, .5,  .6,  .9,  1, 3.3125)
> nums13 <- Rmpfr::mpfr(nums, precBits=13)

                D13          H13                     B13
0.0625 0.06250 +0x1.000p-4 +0b1.000000000000p-4
0.1    0.10001 +0x1.99ap-4 +0b1.100110011010p-4 ## rounded up
0.3    0.29999 +0x1.333p-2 +0b1.001100110011p-2 ## rounded down
0.3125 0.31250 +0x1.400p-2 +0b1.010000000000p-2
0.5    0.50000 +0x1.000p-1 +0b1.000000000000p-1
0.6    0.59998 +0x1.333p-1 +0b1.001100110011p-1 ## rounded down
0.9    0.90002 +0x1.ccdp-1 +0b1.110011001101p-1 ## rounded up
1      1.00000 +0x1.000p+0 +0b1.000000000000p+0
3.3125 3.31250 +0x1.a80p+1 +0b1.101010000000p+1
```

4 Factoring

```
> b6 <- Rmpfr::mpfr(1, precBits=6); a6 <- b6 + 1/8
> b7 <- Rmpfr::mpfr(1, precBits=7); a7 <- b7 + 1/8
```

	Dec6	Bin6	Dec7	Bin7
a	1.125000	+0b1.00100p+0	1.125000	+0b1.001000p+0
a^2	1.250000	+0b1.01000p+0	1.265625	+0b1.010001p+0
b	1.000000	+0b1.00000p+0	1.000000	+0b1.000000p+0
b^2	1.000000	+0b1.00000p+0	1.000000	+0b1.000000p+0
P=(a+b)*(a-b)	0.265625	+0b1.00010p-2	0.265625	+0b1.000100p-2
D=a^2 - b^2	0.250000	+0b1.00000p-2	0.265625	+0b1.000100p-2
P - D	0.015625	+0b1.00000p-6	0.000000	+0b0.000000p+0

5 Mod

```
> Mod(3+4i)
[1] 5

> Mod(30+40i)
[1] 50

> Mod(3e153+4e153i)
[1] 5e+153

> Mod(3e154+4e154i)
[1] 5e+154
```

```
> sqrt(3^2 + 4^2)
[1] 5

> sqrt(30^2 + 40^2)
[1] 50

> sqrt(3e153^2 + 4e153^2)
[1] 5e+153

> sqrt(3e154^2 + 4e154^2)
[1] Inf
```

```
> Mod(3e307+4e307i)          > sqrt(3e307^2 + 4e307^2)
[1] 5e+307                   [1] Inf

> Mod(3e307+4e307i)          > sqrt((3e307/4e307)^2 + (4e307/4e307)^2)*4e307
[1] 5e+307                   [1] 5e+307
```

```
.Machine[c(1,3,4)]
                  numeric                    hex
double.eps   2.220446e-16    +0x1.0000000000000p-52
double.xmin  2.225074e-308   +0x1.0000000000000p-1022
double.xmax  1.797693e+308   +0x1.fffffffffffffp+1023
```

```
> .Machine$double.xmax
[1] 1.797693e+308
> .Machine$double.xmax * (1 + .Machine$double.eps)
[1] Inf
```

5.1 Variance

```
> x <- 1:3          ## 1 2 3

> n <- length(x)

> sum((x-mean(x))^2)       ## two-pass algorithm (numerically good)
[1] 2

> sum(x^2) - n * mean(x)^2 ## one-pass algorithm (dangerous)
[1] 2
```

```
> y <- x + 10^2    ## 101 102 103

> sum((y-mean(y))^2) ## two-pass algorithm (numerically good)
[1] 2

> sum(y^2) - n * mean(y)^2 ## one-pass algorithm (dangerous)
[1] 2

> z <- x + 10^8    ## 100000001 100000002 100000003


> sum((z-mean(z))^2) ## two-pass algorithm (numerically good)
[1] 2

> sum(z^2) - n * mean(z)^2 ## one-pass algorithm (dangerous)
[1] 0
```

Illustrate VARIANCE with 3-bit and 4-bit numbers

```
> (x3 <- mpfr(1:3, precBits=3))
3 'mpfr' numbers of precision 3 bits
[1] 1 2 3

> (M3 <- (x3[1] + x3[2] + x3[3]) / 3)
1 'mpfr' number of precision 3 bits
[1] 2

> (xm2 <- (x3-M3)^2)
3 'mpfr' numbers of precision 3 bits
[1] 1 0 1
## two-pass algorithm (numerically good)
> xm2[1] + xm2[2] + xm2[3]
1 'mpfr' number of precision 3 bits
[1] 2
```

```
> (x4 <- mpfr(1:3, precBits=4))
3 'mpfr' numbers of precision 4 bits
 [1] 1 2 3

> (M4 <- (x4[1] + x4[2] + x4[3]) / 3)
1 'mpfr' number of precision  4 bits
[1] 2

> (xm2 <- (x4-M4)^2)
3 'mpfr' numbers of precision 4 bits
[1] 1 0 1

> xm2[1] + xm2[2] + xm2[3]
1 'mpfr' number of precision 4 bits
[1] 2
```

```
> ## one-pass algorithm (dangerous)
> (x32 <- x3^2)
3 'mpfr' numbers of precision 3 bits
[1] 1 4 8
## 8 = +0b1.00p+3

>(x32sum <- x32[1] + x32[2] + x32[3])
1 'mpfr' number of precision 3 bits
[1] 12
## 13 = 12 = +0b1.10p+3
> mpfr(1:16, 3)
16 'mpfr' numbers of precision  3   bits
[1]  1  2  3  4  5  6  7  8  8 10 12 12 12 14 16 16

> x32sum - 3*M3^2
1 'mpfr' number of precision 3 bits
[1] 0
```

```
> (x42 <- x4^2)
3 'mpfr' numbers of precision 4 bits
[1] 1 4 9
## 9 = +0b1.001p+3

> (x42sum <- x42[1] + x42[2] + x42[3])
1 'mpfr' number of precision 4 bits
[1] 14
## 14 = +0b1.110p+3

> x42sum - 3*M4^2
1 'mpfr' number of precision 4  its
[1] 2
```

5.2 .Machine

```
                        numeric                         hex
double.eps         2.220446e-16      +0x1.0000000000000p-52
double.xmin       2.225074e-308    +0x1.0000000000000p-1022
double.xmax       1.797693e+308    +0x1.fffffffffffffp+1023
double.base                   2             +0x1.00000000p+1
double.digits                53             +0x1.a8000000p+5
double.exponent              11             +0x1.60000000p+3
double.min.exp            -1022             -0x1.ff000000p+9
double.max.exp             1024            +0x1.00000000p+10
integer.max          2147483647            +0x1.fffffffcp+30 ## 2^31 - 1
```

5.3 Square Root $\left(\sqrt{2}\right)^2$

```
> sqrt(2)^2 == 2
[1] FALSE

> showHex(sqrt(2)^2)
[1] +0x1.0000000000001p+1

> showHex(2)
[1] +0x1.0000000000000p+1
```

6 Forthcoming Book

This talk is based on the appendix "Computational Precision and Floating-Point Arithmetic" in the forthcoming Second Edition of my book (with Burt Holland) *Statistical Analysis and Data Display: An Intermediate Course with Examples in R*. It will be available soon, probably in early September.