# Project 1: Properties of Floating Point Numbers

Harper Wang, Cameron Stephens

September 2022

## Contents

## 1 Introduction

We learn arithmetic in base 10, but digital computers perform all operations in base 2. Not all base 10 numbers can be represented exactly in base 2 with a finite number of digits. Digital computers have a finite amount of memory because each binary digit requires a physical space in memory. Due to this, many numbers must be either rounded or truncated before a computer can store and perform operations on them. This round-off error is present in almost every computation performed by a computer but is more obvious in certain cases where problems are ill-conditioned, or sensitive to these round off errors.

### 1.1 How Does a Computer Store Numbers?

Most computers store numbers as binary floating-point numbers. Floating point numbers are of the form

$$(-1)^s(.1d_2...d_t)_2 \times \beta^{e-m},$$

where $\beta$ is the base, s is the sign bit, $d_1...d_t$ is the mantissa, t is the number of bits in the mantissa, e is the exponent bits, and m is the bias of the exponent. Storing numbers in this form allows numbers to be stored (in approximate form) in a finite number of bits, but gives a set precision and range to the number that can be stored. Changing the ways you delegate memory to store floating point numbers can affect the range and precision of the number system. This can also influence how round off errors propagate throughout multi-step calculations. Much of this report will explore some of the consequences of storing numbers in this way.

# 2 Mathematical Issues

## 2.1 Range and Round Off Errors of Different Number Systems

This section is a comparison of the range and relative round off error of two different 32-bit machines. Both machines are base 2, with an implicit first bit. The first machine, F(2,25,-64,63), has 7 bits devoted to to the exponent with bias of 64 and 24 bits devoted to the mantissa. The second machine, F(2,24,-128,127), has 8 bits dedicated to the exponent with a bias of 128 and only 23 bits devoted to the mantissa.

### 2.1.1 Range in a General Base-Two Machine

The smallest possible number for a base-two machine with an implicit first bit corresponds to a mantissa with the first bit (the implicit bit) being one, and all explicit bits that require a physical space in memory being equal to one; and corresponds to all exponent bits being zero, or equal the negative of the bias.

### 2.1.2 Range of the First Machine

For the first machine, F(2,25,-64,63), the smallest positive number corresponds to a mantissa of $(.10000...)_2\dot{2}^{-64} \approx 2.7102 \times 10^{-20}$. The largest number corresponds to a mantissa of $(.11111...)_2 \times 2^{63} = (1 - 2^{-25}) \times 2^{63} \approx 9.22337 \times 10^{18}$. So the range of positive numbers for this floating point number system is $[.27102 \times 10^{-19}, .92234 \times 10^{19}]$.

### 2.1.3 Range of the Second Machine

For the second machine, F(2,24,-128,127), the smallest positive number corresponds to a mantissa of $(.10000...)_2\dot{2}^{-128} \approx 1.46937 \times 10^{-39}$. The largest number corresponds to a mantissa of $(.11111...)_2 \times 2^{127} = (1 - 2^{-24}) \times 2^{128} \approx 3.40282 \times 10^{38}$. So the range of positive numbers for this floating point number system is $[.14694 \times 10^{-38}, .34028 \times 10^{39}]$.

### 2.1.4 The Relative Round Off Error Between Consecutive Floating-Point Numbers

For consecutive floating point numbers of the same exponent, the value of the mantissa changes by $\beta^{-t}$, so the spacing between floating point numbers of the same exponent is $\beta^{e-t}$ once the mantissa is multiplied by the exponent. Knowing this, the maximum round-off error between two consecutive floating point numbers can be derived.
The relative error is the difference between the exact value and the rounded value divided by the exact value. For a floating-point number system that rounds instead of truncating, the maximum error for a number is $\frac{1}{2}\beta^{e-t}$ and

the value it is being compared to is $\beta^{e-t}$. So, the maximum relative error is bounded by

$$\begin{aligned}\sigma_r &\leq \frac{\frac{1}{2}\beta^{e-t}}{\beta^{e-t}} \\ &\leq \frac{1}{2}\beta^{1-t}\end{aligned}$$

## 2.2 The Spacing Between Consecutive Floating-Point Numbers

Because there is only a finite amount of memory a digital computer can have, there is also only a finite number of floating point values that can be represented by a number system used by a digital computer. The quantity of floating point numbers within a range $[\beta^{e-1}, \beta^{e})$ limited by the number of bits dedicated to the mantissa. Floating point numbers on the range $[2^{e-1}, 2^{e})$ all have $2^t$ equally spaced floating point numbers within that range. So as the exponent value increases, the spacing between floating point numbers increases because the range of numbers which the mantissa must span increases, but the number of possible mantissas remain the same. So, the precision of a floating point number system decreases as the exponent increases, causing the round-off error of floating point numbers to increase as the exponent increases. As $1 + \epsilon_1$ is in the range $[2^0, 2^1)$ and $0 + \epsilon_2$ is in the range $[2^{m-1}, 2^{m})$, $\epsilon_1$ will be much larger than $\epsilon_2$ because $m << 1$.

## 2.3 $F(2, 25, 64, 63)$ Representations of $\frac{2}{3}$ and $\frac{3}{5}$

Just as some fractions can not be represented exactly in base 10 and require rounding, eg. $\frac{2}{3} = 0.6\bar{6}6 \approx 0.667$, some fractions can not be representing exactly with a finite number of bits in base 2 as well. For this section we will examine how a 32-bit, base-2 floating-point number system that rounds and has an implicit first bit, i.e. F(2,25,-64,63), will represent the numbers $\frac{2}{3}$ and $\frac{3}{5}$.

### 2.3.1 $\frac{2}{3}$ Floating-Point Representation

Because $2^0 \leq \frac{2}{3} < 2^1$, we know the exponent bit will be equal to zero and the first bit of the mantissa will be 1. When we subtract out the $(.1)_2$ from $\frac{2}{3}$ we get a remainder of $\frac{1}{6} = \frac{1}{3} \times \frac{1}{2^1}$, so we know the second bit of the mantissa is 0 and the third bit is one. Now, subtracting out the $(.101)_2$ from $\frac{2}{3}$ gives us a remainder of $\frac{1}{24} = \frac{1}{6} \times \frac{1}{2^2}$, giving us and fourth and fifth bit of 0 and 1, respectively. Furthermore because we get the same remainder used to calculate the previous two bits multiplied by a factor of $2^{-2}$, we know that these last two bits will repeat indefinitely. Thus, the exact binary representation of $\frac{2}{3}$ is $(\bar{1}\bar{0})_2$. As the 25th bit is 1 and the 26th bit of this number is 0, the 25th bit will remain a 1 when this number is rounded.

$$\frac{2}{3} \texttt{ will be represented as } (.1010101010101010101010101)_2 \times 2^0$$

### 2.3.2 $\frac{3}{5}$ Floating-Point Representation

Because $2^0 \leq \frac{3}{5} < 2^1$, we know the exponent bit will be equal to zero and the first bit of the mantissa will be 1. When we subtract out the $(.1)_2$ from $\frac{3}{5}$ we get a remainder of $\frac{1}{10}$. Since $\frac{1}{2^4} \leq \frac{1}{10} < \frac{1}{2^3}$, the next non-zero bit is the fourth bit. When $(.1001)_2$ is subtracted from $\frac{3}{5}$ the remainder is $\frac{3}{80}$. At this point, the binary representation of $\frac{3}{5}$ begins to repeat every fifth bit because $\frac{3}{80} = \frac{3}{5} \times \frac{1}{2^4}$. So,

$$\begin{aligned}\frac{3}{5} &= (.1\bar{0}01)_2 \\ &\approx (.10011001...100110)_2, \texttt{to 26 bits.}\end{aligned}$$

Thus,

$$\frac{3}{5} \texttt{ will be represented as } (.10011001100110011001100110011)_2 \times 2^0.$$

## 2.4 An Example of How Round-Off Errors Propagate

Because of round off error, floating-point arithmetic can have some odd properties. To illustrate this, consider the following code.

$$\begin{aligned}H &= 1./2. \\ X &= 2./3. - H \\ Y &= 3./5. - H \\ E &= (X + X + X) - H \\ F &= (Y + Y + Y + Y + Y) - H \\ Q &= F/E\end{aligned}$$

### 2.4.1 Manual Calculation of this Code

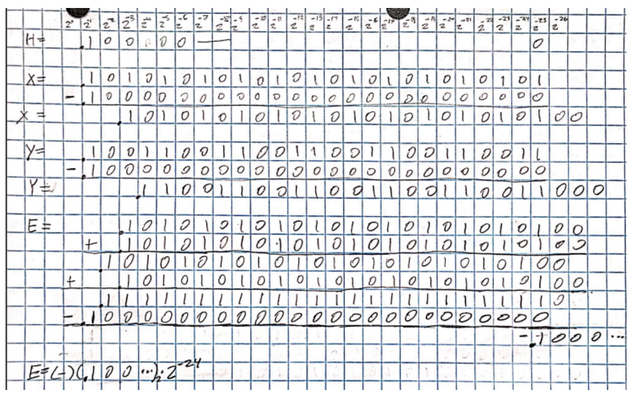

Figure 1: Manual Calculation of X, Y, and E

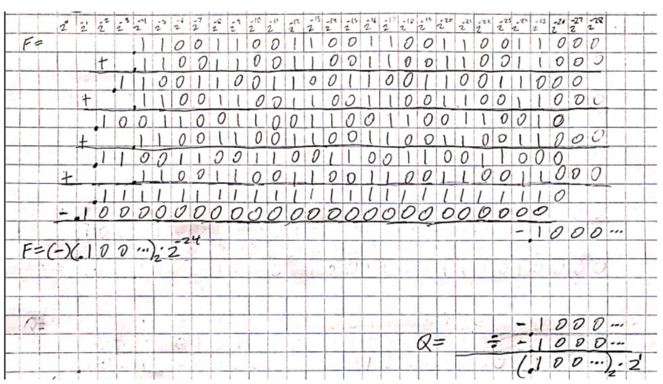

Figure 2: Manual Calculation of F and Q

The solution to this problem when solved manually is

$$Q = \frac{F}{E} = \frac{(-)(.1000...)_2 \times 2^{-24}}{(-)(.1000...)_2 \times 2^{-24}} = 1.$$

The final step in this calculation ends up being the round-off error in F divided by the round-off error in E. Since both E and F have the same round-off error in this particular number system, Q ends up being one. Because different calculators and programs use different floating-point number systems, the round off errors for both F and E are not necessarily the same and different calculators can end up getting very different values for Q depending on what number system they use.

# 3 Programming Issues

## 3.1 Computational Evaluation of $\epsilon$ at Different Points

```
format long
epsilon = 1;
count = 0;
while 1+epsilon > 1
    epsilon=epsilon.*.5;
    count = count +1;
end
% count = 53
% epsilon/2 = 1.110223024625157e-16

epsilon2 = 1;
count2 = 0;
while epsilon2 > 0;
    epsilon2=epsilon2.*.5;
    count2 = count2 +1;
end
% count2 = 1075
% epsilon2/2 ~ 2^-1075
```

The largest positive number $\epsilon_1$ for such that when $1{+}\epsilon_1$ is executed by MATLAB $1 + \epsilon_1 > 1$ holds true is approximately $\left(\frac{1}{2}\right)^{52} \approx 4.940656 \times 10^{-324}$.
The largest positive number $\epsilon_2$ for such that when $0{+}\epsilon_2$ is executed by MATLAB $0 + \epsilon_2 > 0$ holds true is approximately $\left(\frac{1}{2}\right)^{1074} \approx 2.220446 \times 10^{-16}$.

## 3.2 Odd Properties of Floating-Point Arithmetic

Because of round off error, floating-point arithmetic can have some odd properties. To illustrate this, consider the following code.

$$\begin{aligned}
H &= 1./2. \\
X &= 2./3. - H \\
Y &= 3./5. - H \\
E &= (X + X + X) - H \\
F &= (Y + Y + Y + Y + Y) - H \\
Q &= F/E
\end{aligned}$$

### 3.2.1 How does this code behave on different programs?

When this code is executed on an online FORTRAN compiler, the output is 2. When compiled in both Python and MATLAB, the code evaluates Q to be equal to 1.

## 4 Conclusion

### 4.1 How the Floating-Point Number System Used affects Range and Precision

The range and precision of a floating-point number system is determined by the number of bits dedicated to the mantissa, the number of bits dedicated to the exponent, and the bias on the exponent. While you can increase your range and precision indefinitely by simply adding more memory to the mantissa and exponent, this is not always the best option because memory is a very valuable resource in computation. So there is often a trade-off between range and precision. In other words, moving bits that are dedicated to the exponent to the mantissa will decrease the range of a number system but increase its precision.

### 4.2 Relative Precision of Floating-Point Numbers

The relative precision of a floating point number system (when comparing numbers with the same exponent value) is determined only by the base of the number system and the number of bits dedicated to the mantissa. A longer mantissa corresponds to less relative error.

### 4.3 Absolute Precision of a Floating Point Number System

The absolute precision of a floating point number is dependant on both the mantissa length and what the exponent. As the round-off error is bound by $\frac{1}{2}\beta^e - t$, the round-off for numbers of the same exponent is bound by the same value; but as the value of the exponent bit increases, the upper bound on absolute error increases. This demonstrates that, as the order of magnitude increases, the absolute error increases as well. Conversely, as long as a number is within the range of the number system, numbers of a smaller order of magnitude will have smaller absolute error.

### 4.4 How Round-Off Error Can Affect Ill-Conditioned Problems

When a problem is ill conditioned, such as in the "$\frac{0}{0}$" problem examined earlier, round-off error can become very obvious. In this case, there is no possible way for a binary floating-point number system with a finite amount of memory to exactly represent E and F; so this computation was designed to find the proportion of round-off error in F to round-off error in E. Because different compilers both use different floating-point number systems and do not round the same, different programs have the potential to output wildly different "solutions." If the number system used by a compiler is know, then the answer to a computation like this can actually be calculated manually.

## 4.5 How Understanding Machine Error Allows Us to Make Better Predictions

Consider Euler's Method for Numerical Integration. We generally consider the errors in this method to decrease proportional to $\Delta x$. This is true to a point, but if you consider the case where $\Delta x$ approaches $\epsilon$ then the errors will actually increase as $\Delta x$ decreases due to the propagation of round-off error. Furthermore, if we understand how uncertainty propagates throughout a calculation and what the uncertainty for a floating point number is; then we can estimate the error associated with a numerical calculation even if we do not know the exact solution.