Computation Physics Problem Set 1

Vedhasya Muvva

September 17, 2024

1 Problem 1

100.98763 can be represented in 32 bits by first being written in the binary equivalent of scientific notation as 1.57793171875×2^6. In this case, the 1.57793171875 is the mantissa and the 6 is the exponent. For easier conversion, this will now be represented in the following form:

$$(-1)^{\text{sign bit}}(1 + \text{fraction}) * 2^{\text{exponent - bias}}$$

The equivalent of 100.98763 in this form is as following:

$$(-1)^0(1 + 0.57793171875) * 2^{(6+127)\text{ - }127}$$

This includes a bias of 127, which is standard for single-precision (32-bit) floating point numbers.

Now to translate each of the values to binary. The sign bit will stay as 0. The exponent $6 + 127 = 133$ can be written in binary as 10000101. The fraction can be repeatedly multiplied by two to find the binary form. In this case it will look like the following:

$$
\begin{aligned}
0.57793171875 * 2 &= \mathbf{1}.1558634375 \\
0.1558634375 * 2 &= \mathbf{0}.311726875 \\
0.311726875 * 2 &= \mathbf{0}.62345375 \\
0.62345375 * 2 &= \mathbf{1}.2469075 \\
0.2469075 * 2 &= \mathbf{0}.493815 \\
0.493815 * 2 &= \mathbf{0}.98763 \\
0.98763 * 2 &= \mathbf{1}.97526 \\
0.97526 * 2 &= \mathbf{1}.95052 \\
0.95052 * 2 &= \mathbf{1}.90104 \\
0.90104 * 2 &= \mathbf{1}.80208 \\
0.80208 * 2 &= \mathbf{1}.60416 \\
0.60416 * 2 &= \mathbf{1}.20832 \\
0.20832 * 2 &= \mathbf{0}.41664 \\
0.41664 * 2 &= \mathbf{0}.83328 \\
0.83328 * 2 &= \mathbf{1}.66656 \\
0.66656 * 2 &= \mathbf{1}.33312 \\
0.33312 * 2 &= \mathbf{0}.66624 \\
0.66624 * 2 &= \mathbf{1}.33248 \\
0.33248 * 2 &= \mathbf{0}.66496 \\
0.66496 * 2 &= \mathbf{1}.32992 \\
0.32992 * 2 &= \mathbf{0}.65984 \\
0.65984 * 2 &= \mathbf{1}.31968 \\
0.31968 * 2 &= \mathbf{0}.63936 * * * \text{last digit rounds up to 1} \\
0.63936 * 2 &= \mathbf{1}.27872 \\
0.27872 * 2 &= \mathbf{0}.55744 \\
&\text{etc}
\end{aligned}
$$

This is the equivalent of finding the 2^{-1},2^{-2}, 2^{-3},..., 2^{-n}th terms. Reading the bolded values from top to bottom returns the binary value for the fraction.

In IEEE notation, the first bit in a 32-bit float is the sign, the next 8 bits represents the exponent, and the last (32-1-8 = 23) 23 bits represent the fraction. Only the first 23 terms from the fraction can be used for these space constraints with the last value rounded up from 0 to 1, which become the binary value 0.10010011111100110101011. Putting all of these values together returns **0 10000101 10010011111100110101011**. Because the fraction has been rounded to 23 bits, final bit went from $0.0 * 2^{-23} + 0.63936 * 2^{-24}$ to $1.0 * 2^{-23}$, meaning the actual value is off from the represented value by $(1.0 * 2^{-23} - 0.0 * 2^{-23} - 0.63936 * 2^{-24}) * 2^6 = 8.11004639e - 8 * 2^6 =$ **0.00000519042**.

2 Problem 2

In a similar manner as the first problem, lets first look at the following formula:

$$(-1)^{\text{sign bit}}(1 + \text{fraction}) * 2^{\text{exponent - bias}}$$

The biggest difference between the 32-bit representation and the 64-bit representation is the number of bits allocated for the fraction and the exponent. The 32-bit has 8 bits for the exponent and 23 bits for the fraction. The 64-bit has 11 bits for the exponent and 52 bits for the fraction. Accordingly, the 32-bit has a $2^{8-1} - 1 = 127$ bias for the exponent while the 64-bit has a $2^{11-1} - 1 = 1023$ bias for the exponent. Because of this space, the smallest the exponent can be is -127 for the 32-bit and -1023 for the 64-bit.

Since 1 would be represnted as $(-1)^0(1 + 0) * 2^0$ for both floating point values, the smallest value that can be added to 1.0 would have to be whichever fractional part of the equation was the smallest possible number using the available bits. For 32-bit, this would be 2^{-23} and for the 64-bit, this would be 2^{-52}. Converting both those values into decimal would be $2^{-23} =$**1.1920929e-7** for 32-bit and $2^{-52} =$**2.220446e-16**.

The minimum values will need to be the values where the sign bit, the fraction, and the exponent are all set to 0. This results in $1*2^{-127} =$**5.8774718e-39** for 32-bit floating numbers and $1*2^{-1023} =$ **1.112537e-308**.

The maximum values can be found when the sign bit is set to 0, the fraction has every available bit except for the last one set to 1 (because all of them set to 1 would result in inf), and the exponent is set to its highest value as well. fraction would be $\sum_{i=-(\text{number of bits-1})}^{-1} 2^i$ which would be slightly less than 1.0.

For the 32 bit, the exponent would be $2^8 - 1$. For the 64-bit, the fraction would be $2^{11} - 1$. Plugging these values in the equation, the largest value representable by a 32-bit float should be less than $(-1)^0(1 + 0.999) * 2^{2^8-1-2^{8-1}+1} = (1.999) * 2^{2^7} \simeq$ **e+38**. Similarly, the largest value representable by a 64-bit float should be less than $(-1)^0(1+0.999)*2^{2^{11}-1-2^{11-1}+1} = (1.999)*2^{2^{10}} \simeq$ **e+308**.

Summarizing this values results in:

	32-bit	64-bit
minimum to add to 1	2^{-23}	2^{-52}
minimum	2^{-127}	2^{-1023}
maximum	$2^{2^7+1} - 1$	$2^{2^{10}+1} - 1$

Table 1: Summary of values from Problem 2

3 Problem 3

I have written two versions of the code, one with a for loop and one without. I get a Madelung constant of approximately 1.74 with both versions, though the code without the for loops is significantly faster. With an L set to 70, the code with the for loop takes approximately 113 seconds while the code without the for loops takes approximately 0.568 seconds, meaning it is faster to code without for loops.

4 Problem 4

The plot of the mandelbrot set with a 500 x 500 grid and 100 iterations per point is showing the Figure 1.

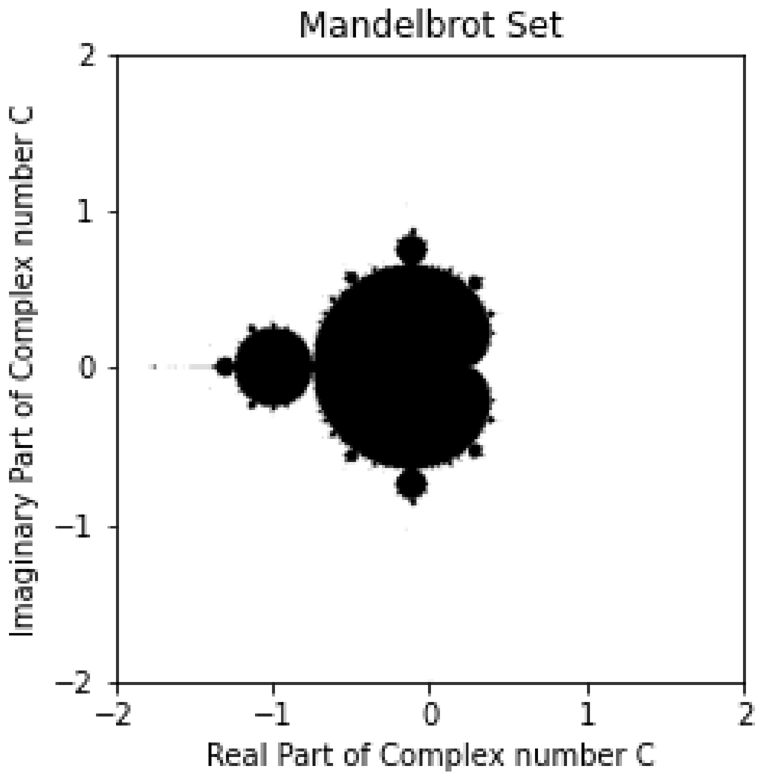

Figure 1: Coded plot of the Mandelbrot Set for each value of x and y within a range of [-2,2]

5 Problem 5

5.1 a

Using python and the given formula, the solutions to the equation are: -9.999999994736441314e-07 and -999999.9999989999792.

5.2 b

The previous equation can be rewritten in the following way.

$$\begin{aligned}
\frac{-b \pm \sqrt{b^2-4ac}}{2a} &= \frac{(-b \pm \sqrt{b^2-4ac})(-b \mp \sqrt{b^2-4ac})}{2a(-b \mp \sqrt{b^2-4ac})} \\
&= \frac{b^2-(b^2-4ac)}{2a(-b \mp \sqrt{b^2-4ac})} \\
&= \frac{4ac}{2a(-b \mp \sqrt{b^2-4ac})} \\
&= \frac{-2c}{(b \pm \sqrt{b^2-4ac})} \\
&= \frac{2c}{-b \mp \sqrt{b^2-4ac}}
\end{aligned}$$

The new results using these equations are -1.0000000000010000208e-06 and -1000000.0005263558689. The difference in results is most likely caused by rounding errors, specifically in computing b^2-4ac. Since the b^2 is so much larger than $4ac$, the details from $4ac$ get lost in rounding and effectively $\sqrt{b^2-4ac} \simeq b$. Because of this, $-b+\sqrt{b^2-4ac} \simeq 0$ and cancellation occurs. In the first equation, this gives you a small term on the numerator, but for the second equation, the term is in the denominator instead. Either way, one of the solutions to the problem will have cancellation problems no matter which equation you use.

5.3 c

The new solution to the quadratic formula problem is to get one solution from each equation, choosing the option that avoids the cancellation error.