Computational Physics: PS 2

Marcus Hoskins

September 17, 2024

1 Discussion

1.1 Problem 1

In this problem we want to represent the number 100.98763 as a NumPy 32-bit floating point in bits. That is, we want to use the representation:

$$100.98763 = \left(1 + \sum_{i=0}^{22} f_i 2^{-(i-23)}\right) 2^{e-127},$$

which is the IEEE form of a 32-bit floating point number taken from the class notes. Then, we proceed as we would if we were in base 10. We want to get rid of the exponent on the right hand side and at the same time multiply the left hand side by a number that's of order 10^5, so that we can get a whole number. Of course here, being in base 2, this won't be fully possible. But, solving the equation $10^5 = 2^x$, we see that $x = -16.609$. So, using x=−16 and solving for our exponent, we find that $e = 111$. Then, we can solve for the mantissa. Doing so, we see that:

$$2^{16} \cdot 100.98763 = 1 + \sum_{i=0}^{22} f_i 2^{-(i-23)},$$

meaning for us (to the nearest whole number), the mantissa f is:

$$f = 6618324,$$

which takes 23 bits to describe, exactly the number of bits dedicated to the mantissa. Then, writing the exponent and mantissa in binary, and noting that the sign bit is 0 here, we find that the 32-bit floating point representation of 100.98763 is:

0	0	1	1	0	1	1	1	1	1	1	0	0	1	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	0	0

Technically, this bit representation describes the number 100.98761749..., which differs from the true number by 0.00002.

1.2 Problem 2

To begin, we wish to find the smallest value that one can add to 1 and get an answer that's different than 1. In 32-bit precision, I found that adding anything smaller than 10^{-7} gives me a number that's 1.0. Thus, the smallest such value in 32-bit is 10^{-7}. In 64-bit precision, I found the corresponding number to be 10^{-15}. Thus, we have found one example where 32-bit is less precise than 64-bit, as the smallest number greater than 0, which when added to 1 gives a value that's not 1, that we can represent in 32-bit is larger than in 64-bit, by a factor of 10^8.

We now wish to find approximately the minimum and maximum positive numbers that these two precision types can represent without overflow or underflow. For 32-bit, I found that the minimum positive number without underflow to be 1.7×10^{-45}, and the maximum positive number without overflow to be 1.7×10^{38}. Alternatively, for 64-bit I found the minimum positive number to be 2^{-1075} and the maximum positive number to be 2^{1023}.

1.3 Problem 3

The largest L value I could use such that the for loop method ran in around a minute was $L = 200$. So, for $L = 200$, I found the Madelung constant for sodium chloride to be:

$$M = \sum_{i,j,k=-L}^{L} \frac{1}{\sqrt{i^2 + j^2 + k^2}} = 382716.5, \qquad \text{exlcuding} \quad i = j = k = 0,$$

which for the method using for loops ran in 1 minute and 23.97 seconds, whereas for the method that didn't use for loops ran in 1.39 seconds: a startling difference!

1.4 Problem 4

My method for searching over the space of c values used for loops. So, my grid spacing was $N = 100$. Using this spacing, the Mandelbrot set can be seen in figure 1.

1.5 Problem 5

1.5.1 Part (a)

Using the program outlined, I found that the 2 roots of the equation $0.001x^2 + 1000x + 0.001 = 0$ are $-9.99989425 \times 10^{-7}$ and $-1.00000000 \times 10^{6}$.

1.5.2 Part (b)

Multiplying the usual solutions of a quadratic equation by the given values:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b \mp \sqrt{b^2 - 4ac}}{-b \mp \sqrt{b^2 - 4ac}} = \frac{b^2 - b^2 + 4ac}{2a(-b \mp \sqrt{b^2 - 4ac})} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}},$$

as expected. Then, using the outlined program, we find the roots of $0.001x^2 + 1000x + 0.001 = 0$ to be $-1.00000000 \times 10^{-6}$ and $-1.00001058 \times 10^{6}$. We notice that these roots are not the same

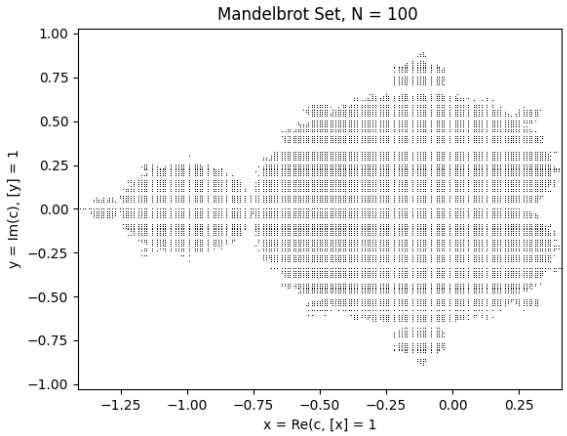

Figure 1: This is the Mandelbrot for $-2 \leq x, y \leq 2$ with lattice spacing $N = 100$.

as those found in part (a). Specifically, the root that's accurate here is the root that was slightly inaccurate in part (a), and vice versa. To explain this, we recall that larger machine precision errors occur when multiplying numbers of vastly different orders of magnitude. That is, in part (a) we see that $a = 0.001$, meaning $\frac{1}{2a}$, which multiplies the numerator, is large. Then, the root where the numerator is a small number (in magnitude), x_+, is that root that has some machine precision errors. However, the root where the numerator is itself a large number in magnitude, x_-, does not display any machine precision errors. Similarly, in part (b), we see that $c = 0.001$, a small number. So, the root where the denominator is a small number, x_-, or rather where the inverse of the denominator, which multiplies the numerator, is a large number (in magnitude), has some machine precision error. However, the root where the denominator is a large number (in magnitude), meaning the numerator is being multiplied by a number that is small in magnitude, has no such error, as we're multiplying 2 numbers that are small in magnitude.

1.5.3 Part (c)

Using what we've learned from part (b), we write this new code with the understanding that we wish to avoid cases where there is a large difference in magnitude between the numerator and the reciprocal of the denominator of the quadratic roots. How large this difference is before the code notices is something that we set by hand, and for me I set the minimum difference at which the program notices to be 100, or 2 orders of magnitude. My program also passes the given unit tests.