Problem Set 2

Evan Arch

September 18, 2024

Problem 1

The float32 binary value of 100.98763 is 01000010110010011111100110101011. Due to the imprecision of floats, this binary actually represents a slightly different number. We can convert it to a decimal number by first separating the sign(which we will neglect since the number is positive), the exponent and the mantissa. We can convert the exponent directly into decimal which gives 133. Subtracting 127 gives 6. This tells us that we need to place a decimal point 6 places over in the mantissa. Before that can be done, a decimal point must exist in the mantissa, so we add the implicit 1. to the front of it, 1.10010011111100110101011. Moving the decimal 6 places to the right gives 1100100.11111100110101011. We can now convert the left and right sides of the decimal point to decimal separately. The left side is simply and can be done in the standard way. However, since the right side represents a fraction, we must convert it using

$$\sum_i 2^{-1} * b_i, \tag{1}$$

where b_i is the bit at the ith position. This gives a final decimal representation of 100.9876327514648438. This differs from the initial float by $2.75146 * 10^{-6}$.

Problem 2

The smallest number that can be added to 1 and result in a number other than 1 can be found be adding successively smaller numbers to 1 and checking if the result is equal to 1. I wrote code to divide 1 by 10 until the result gives 1 when added to 1. A similar strategy can be used to find the smallest number before an underflow occurs, dividing 1 by 10 until the result equals 0. The largest value before an overflow can be found by multiplying 1 by 10 until the result equals np.inf. All of these processes were carried out for 32 bit and 64 bit numpy floats and the results are in the table below.

	float32	float64
Smallest value added to one	9.999999e-09	1.0000000000000001e-16
Smallest value before underflow	1e-45	1e-323
Largest value before overflow	1.0000001e+38	9.999999999999998e+307

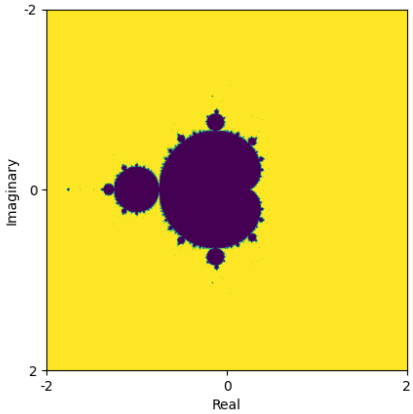

Figure 1: Mandelbrot set on a 1000x1000 grid from -2 to 2 with 100 iterations per point

Problem 3

The Madelung constant for NaCl M can be found using:

$$M = \sum_{ijk=-L}^{L} (-1)^{i+j+k}(i^2+j^2+k^2)^{-1/2}, \tag{2}$$

where i, j, k are the indices of points in a simple cubic lattice, and $i = j = k \neq 0$. This sum can be done in two ways, either using for loops, or by applying a vectorized function to a meshgrid that contains all the points and summing the resultant array. Both methods produce approximately the same value for the Madelung constant, $M \approx -1.74181$. With my implementation on my computer, using a grid size of 200x200x200, the for loop method takes 9.5s. The meshgrid method is about 140 times faster with a runtime of 67ms.

Problem 4

A complex number c is in the Mandelbrot set if the magnitude of $z' = z^2 + c$ is ever greater than 2 when applies recursively with z starting at 0. A plot generated with python is shown in figure 1.

The problem presents an interesting trade off, to apply a function to every element of a numpy array (a meshgrid in this case) the function must be vectorized. This means it cannot contain any if statements. This means that the Mandelbrot set equation cannot be applied recursively since there is no simple way to end the recursion. Thus, it must be applied iteratively, a fixed number of times. While this wastes some processing power iterating over some points that go to 2 quickly, it appears to make up said time through the benefits of vectorization.

Problem 5

Part A

Using the standard quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{3}$$

with $a = 0.001, b = 1000, c = 0.001$, the quadratic equation, implemented using numpy float32, gives the plus solution $-9.999894245993346e-07$, and the minus solution, -999999.999999. While the minus solution is quite accurate, the expected result is $-1 * 10^6$, the plus solution is not very good, the expected result is $-1 * 10^{-6}$.

Part B

The modified quadratic equation,

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}, \tag{4}$$

we appear to get better accuracy for smaller results than standard quadratic formula. Using the same values as before, $a = 0.001, b = 1000, c = 0.001$, the plus solution is $-1.000000000001e-06$, and the minus solution is -1000010.5755125057. Since the plus solution is more accurate with this formula, and the minus more accurate with the previous, the two could be combined to get a more accurate overall solution.

Part C

By combining the equations used in parts A and B, we can create a function that uses the most accurate one in each case. If we try using both equations with $a = 0.001, b = -1000, c = 0.001$, we find that the standard quadratic equation is more accurate for the plus condition, and the equation from part B is more accurate for the minus condition. Thus, we can pass the unit test by applying the equation from A for the plus solution and the equation from B for the minus solution when $b < 0$, and doing the opposite otherwise.