# Problem Set 2

Prithivi Rana

September 18, 2023

**Abstract**

This problem set dealt with bit representation, dealing with round off errors, using multidimensional arrays, and unit testing

## 1 Problem 1

The first problem asked us to explore NumPy's 32-bit floating-point representation and determine how it represents the number 100.98763 in bits. Then we were tasked with calculating the difference between the actual number and its 32-bit floating-point representation.

My python program accomplishes this by converting 100.98763 to a 32-bit floating-point representation by using the np.float32() method. Then, I convert the 32-bit float to a 32-bit integer using .view(np.int32) to access its binary representation.

I broke down the binary representation into its sign, exponent, and mantissa components. Lastly, I converted the integer back to a 32-bit float and calculated the difference between this representation and the actual number (Figure 1).

```
prithivirana@PrithivsMacbook desktop % python3 float.py

Numpy's 32-bit floating point representation of  100.98763 :
        Sign:  0
        Exponent:  1000010
        Mantissa:  10010011111100110101011

Difference from actual num:  2.7514648479609605e-06
```

Figure 1: Output for Problem 1 exploring NumPy's floating bit representation.

## 2 Problem 2

The second problem asked us to calculate the Madelung constant for Sodium Chloride. The problem also asked for two solutions for the problem: one that uses loops, and one that accomplishes the result without using loops.

The first version of my solution uses three nested loops that iterate over the range -L to L (L was set to 100 for my solution). The loop passes over the iteration where i=j=k=0 and does

the calculation for all other values. This version outputs a value around -1.7418 for L and takes approximately 2.22 seconds to run as seen in the screenshot below (Figure 2).

The second version uses the np.meshgrid function to create a mesh of coordinates contained in a cuboid of length 2L around the origin (0,0,0). This removed the need for using loops and in turn makes the computation faster. This version outputs a similar value for L and takes approximately half the time ( 1.14 seconds).

```
Calculating Madelung Constant using a loop:
        L =  -1.7418198158396148
        Time =  2.228927404999922  seconds

Calculating Madelung Constant using a 2D array:
        L =  -1.741819815836106
        Time =  1.1448934150012064  seconds
```

Figure 2: Output for Problem 2 exploring the speed and efficiency of loops vs vectorized operations.

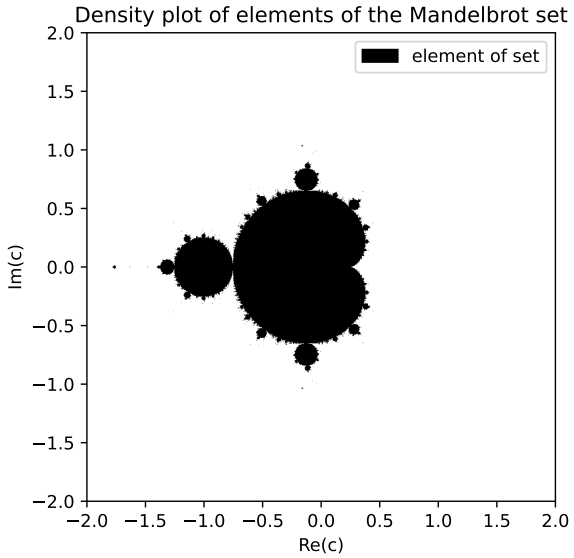
# 3 Problem 3

This problem tasked us with making an image of the Mandelbrot set. My code accomplishes this by making a meshgrid of NxN (N=1000 for the density plots in Figure 3) gridpoints between -2 and 2. The meshgrid represents a complex number C with Cx (x axis) being the real part and Cy (y axis) being the imaginary part.

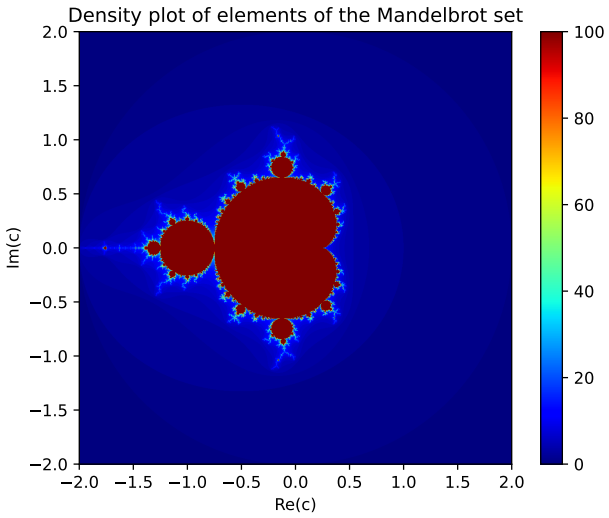
Two arrays, Zx and Zy, were declared and initialized to zeros. They represent the real and imaginary parts, respectively, of the iterated values Z. Then, a boolean array was declared and initialized to true and then used as a mask to track which grid points remain bounded as the iterations progress.

The main iteration loop computes values of Z' starting with z=0 and loops for 100 iterations. The magnitude of the new Z values is updated for points which have not yet been determined to be unbounded (as per the mask). If the magnitude of Z becomes greater than or equal to 2 for any point, that point is considered unbounded and its mask value is set to False, excluding it from the Mandelbrot set. The resulting density plot from this method was created using the "imshow" function (Figure 3a)).

Another variant of the program was made to keep track of the number of iterations for each point before the magnitude of Z became greater than 2. If the magnitude never surpassed 2, the maximum number of iterations was chosen for that element. The density plot of this variation of the problem can be seen in Figure 3b.




(a) Density plot from first version of program set




(b) Density plot showing number of iterations before magnitude of Z reaches 2

Figure 3: Outputs from the Mandelbrot Set problem

# 4 Problem 4

The last problem dealt with writing a function that computes the roots of a quadratic equation $ax^x + bx + c = 0$ accurately. This function determines what method to use depending on the sign of the coefficient b.

If b is positive, it will calculate the positive root using the formula $x = \frac{2c}{-b-\sqrt{b^2-4ac}}$. This formula is used to avoid roundoff error that would occur from subtracting $\sqrt{b^2-4ac}$ from b when b is very large and ac is small. The negative root is calculed using the normal form of the equation: $x = \frac{-b-\sqrt{b^2-4ac}}{2a}$.

If b is negative, it will use $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ for the positive root and $x = \frac{2c}{-b+\sqrt{b^2-4ac}}$ for the negative root.

A unit test was done on this function with the test_quadratic.py file that was provided and the module passed the unit test.

```
prithivirana@10-17-128-118 Desktop % pytest test_quadratic.py
============================= test session starts ==============================
platform darwin -- Python 3.11.4, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/prithivirana/Desktop
collected 1 item

test_quadratic.py .                                                      [100%]

============================== 1 passed in 0.11s ===============================
```

Figure 4: Output for Problem 4 showing the result of a unit test for the quadratic solver function