Problem Set 2

Dario Loprete

September 17, 2024

1 Problem 1

Using the function `get_bits` from lecture, the 32-bit representation for the number $\mathbf{x = 100.98763}$ is given by:

$$01000010110010011111100110101011 \tag{1}$$

Then, we apply the formula which converts the 32-bit representation of a normal number to its value in base-10 :

$$\text{decimal value} = (-1)^{b_{31}} \text{ x } 2^{e-127} \text{ x } \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right), \tag{2}$$

where b_i is the i^{th} bit in the binary representation (with b_0 being the least significant digit, namely the one on the far right) and e is the exponent, which is given by $b_{30}b_{29}...b_{23}$. The function `float_to_dec` applies this formula using for loops and returns the corresponding value in base 10. The obtained value is:

$$100.98763275146484 \tag{3}$$

Finally, the difference between the inital value and that obtained using this function is given by:

$$2.7514648479609605 \text{ x } 10^{-6} \sim 2.75 \text{ x } 10^{-6} \tag{4}$$

This difference is due to the limited resolution of the floating point 32-bit representation, which is of order 10^{-6}.

2 Problem 2

2.1 Methodology

In order to find the smallest value you can add up to 1, we can look at the sum: 1 + step. If this sum returns a value which is equal to 1, it means that we have added to 1 a value which cannot be represented, namely it is below the precision of the chosen representation. So, we can start with step = 1, for which the sum = $2 \neq 1$, and divide the step by 2. We add this new value to 1 and

check the nature of the resulting sum. We repeat this logic over and over until we approximately reach the smallest value we can represent.

To approximately find the minimum value we can represent, we start from 1 and keep dividing it by 2. When a value which is smaller then the smallest representable positive number is encountered, it underflows to 0. Therefore, we can stop the loop when the resulting value is equal to zero.

To approximately find the maximum value we can represent, we start from 1 and keep multiplying it by 2. When we reach a value which is equal to infinity, it means that overflow has occurred. Therefore, the maximum value corresponds to the one we obtain before encountering overflow. (When running the code, a RuntimeWarning about overflow is displayed, but it does not stop us from obtaining an approximate value for the maximum representable number.

2.2 Results

32-bit precision
Smallest value you can add up to 1 $\sim$ 1.2 x 10^{-7}
Minimum positive number $\sim$ 1.0 x 10^{-45}
Maximum positive number $\sim$ 1.7 x 10^{38}

64-bit precision
Smallest value you can add up to 1 $\sim$ 2.2 x 10^{-16}
Minimum positive number $\sim$ 5.0 x 10^{-324}
Maximum positive number $\sim$ 9.0 x 10^{307}

These numbers are approximate, namely they may not coincide with those one obtains from using the functions `np.finfo(np.float32)` and `np.finfo(np.float64)`. Indeed, the limited precision and the presence of possible rounding errors in the performed operations may cause the obtained results to be slightly different from the exact ones.

3 Problem 3

Given the instructions and theory of Exercise 2.9, the Madelung constant for sodium chloride can be computed with a for loop or without. Eq. 5 and Eq. 6 report the formulae to be implemented:

$$M = \sum_{\substack{i,j,k=L \\ \text{not } i=j=k=0}}^{L} V(i,j,k), \tag{5}$$

with[1]

$$V(i,j,k) = \pm \frac{1}{\sqrt{i^2+j^2+k^2}} \tag{6}$$

[1] In the equations, the physical constants drop out.

where + corresponds to even $i+j+k$ (sodium atoms), and − corresponds to odd $i+j+k$ (chlorine atoms).

Implementation with for loop
We can compute the total potential felt at the origin by using three concatenated for loops, each of them running respectively through the indices i, j, k. This is performed by the function `madelung_loop`, in which, for each iteration, the contribution to the total potential due to the atom at position (i, j, k) is added. The function `potential_loop` evaluates the potential due to the atom at position (i, j, k), taking into account the positive or negative sign by means of an if condition.

For $L = 100$, the duration of the computation is $\sim 17s$, which surely seems significant. The obtained Madelung constant is ~ -1.74.

Implementation without for loop
Instead of using three concatenated for loops, we can make use of NumPy methods to speed up the computaion. The function `madelung_no_loop` creates a grid of coordinates for all the atoms in the lattice by means of `np.meshgrid`. In one single line, we can compute all the values $V(i, j, k)$ (see Eq. 6) using the function `potential_no_loop`, which takes as input the arrays of coordinates. Then we perform the sum using `np.sum`.

For $L = 100$, the duration of the computation in this case is $\sim 0.7s$, which is certainly faster and more efficient. The obtained Madelung constant is ~ -1.74.

4 Problem 4

In Exercise 3.7, we are asked to make an image of the Mandelbrot set, which is given by the repeated iteration of the following equation:

$$z' = z^2 + c, \tag{7}$$

where z and $c = x + iy$ are complex numbers. After a sufficient number of iterations, if the magnitude of the new number is less then 2, then that point belongs to the Mandelbrot set. For this problem, we consider values of c on an N x N grid spanning the region where $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. The code performs the iteration using a for loop. After creating a grid of values for c and z, each value of z gets updated using Eq. 7. If the magnitude of the new z is greater than 2, that value is set to a conventional one (we choose 3), so that we know that such z has already escaped the region such that $|z| \leq 2$. Additionally, we also save the number of iterations performed right before this threshold is crossed. After the for loop, we distinguish the z that are part of the Mandelbrot set from those which are not by associating 0 to the former and 1 to the latter.

The dimension N of the grid was set to 1500, while the number of iterations was set to 120. These values are sufficient to well represent the Mandelbrot set. Given the final grid with 0s and 1s, we can make an image by using the function `matplotlib.axes.Axes.imshow`, which is reported in Fig. 1.

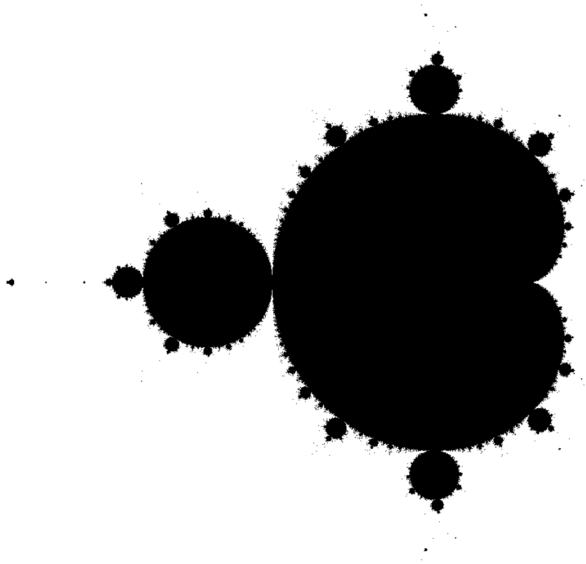

Figure 1: Mandelbrot set. In black, the points belonging to the set; in white, the points not belonging to the set.

If we color points based on the number of iterations performed right before the magnitude of z becomes greater then 2, we obtain the plot in Figure 2

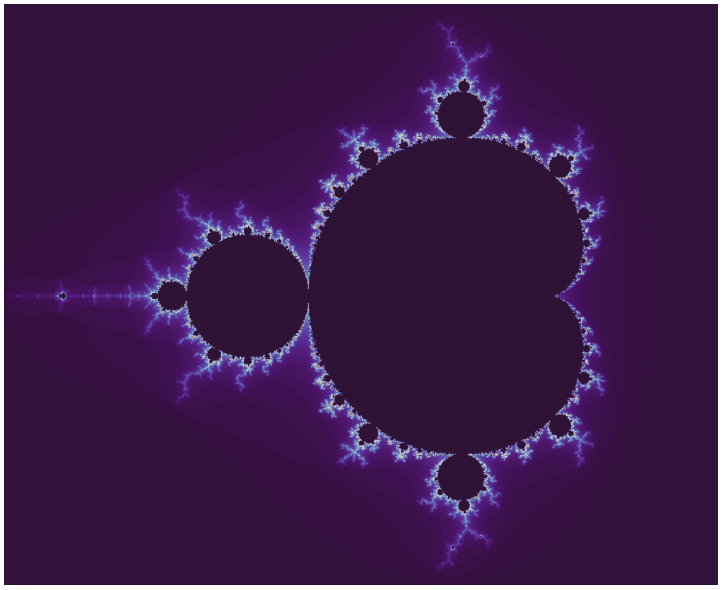

Figure 2: Mandelbrot set, colored version.

5 Problem 5

Exercise 4.2 asks to find the solutions for the equation

$$ax^2 + bx + c = 0, \tag{8}$$

with a = 0.001, b = 1000, c = 0.001. Method a) is based on the implementation of the standard formula

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{9}$$

For Method b) we notice the following:

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \frac{-b \mp \sqrt{b^2 - 4ac}}{-b \mp \sqrt{b^2 - 4ac}} = \tag{10}$$

$$= \frac{b^2 - b^2 + 4ac}{2a(-b \mp \sqrt{b^2 - 4ac})} = \tag{11}$$

$$= \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \tag{12}$$

The implementation of these two methods in the code `prob_5_ab.py` results in the following:

Method a)
$x_+ = -9.999894245993346$ x 10^{-7}
$x_- = -999999.999999$

Method b)
$x_+ = -1.000000000001$ x 10^{-6}
$x_- = -1000010.5755125057$

We can see that x_- from method a) and x_+ from method b) are closer to the exact results than the other values. Indeed, when computing x_+ with method a) and x_- with method b), the code needs to calculate the difference $-b + \sqrt{b^2 - 4ac}$. For the given values of a, b and c, this corresponds to a very small difference compared to the large numbers at play. This can lead to a loss of precision due to possible rounding errors. Hence, we obtain inaccurate results. Method c) provides a solution to this problem, which is implemented in the module `quadratic`. When the absolute value of the difference $-b + \sqrt{b^2 - 4ac}$ is smaller than a certain threshold (i.e. 0.001), then we should avoid using this difference in the computation, but instead obtain x_+ by means of method b) and x_- by means of method a). Both these numbers are evaluated by computing $-b - \sqrt{b^2 - 4ac}$, which does not give any problem. The function `quadratic` also provides the option of computing the two solutions via method a) when the difference $-b + \sqrt{b^2 - 4ac}$ is larger then the threshold, namely when we don't encounter loss of precision.