Problem Set 2

Debendro Mookerjee
GitHub Account Name: Debendro-dm5790

September 18, 2023

Abstract

In this homework set we will be discussing two types of problems: computational problems that can be performed by using iterations and the numerical accuracy of numbers stored in, and calculations performed on, a computer. The second and third problems fall into the former category and the first and fourth problems fall into the latter one. After introducing the four problems and the methods used to solve them, we provide our code outputs which include plots and unit test outputs on the terminal.

1 Introduction

In order to understand numerical inaccuracies in computers, it is important to know how floating point numbers are stored in them. Although such numbers are stored with varying degrees of precision such as 32-bit precision and 64-bit precision, the focus of the first exercise is on the former type of floating point number. This exercise considers the decimal number 100.98763. We convert this number into a 32-bit float by using IEEE's standard formula. The resulting string of 32 binary digits will not be equal to the original number. The number the string of bits corresponds to is called a machine number. For a given precision, the machine numbers are the numbers the computer can store exactly.

The previous decimal number is an example of a rational number and these numbers are often present when the computer performs computations; however, there are several times when the computer performs operations that yield irrational numbers. Since these numbers have a non terminating decimal representation, the machine numbers they are stored to are guaranteed to approximate the irrational numbers. Numerical errors are abundant.

An example of an operation that often yields irrationals is the square root operation. The numerical errors due to the square root operation is the focus of the fourth exercise, which considers two ways to calculate the roots of a quadratic: the standard formula which has the square root in the numerator and the conjugate formula which had the square root in the denominator. Numerical errors are guaranteed to happen. Furthermore, if the square root operation leads to the addition of two numbers with similar magnitudes and opposite signs, errors will be amplified. This will be illustrated by running pytest on the terminal.

The second and third exercises explore loops. Although numerical errors are present in such

iterative algorithms, these problems do not focus on them. The second problem considers the sum of potentials on an infinite simple cubic lattice whose sites alternate between one with a net positive unit charge and one with a net negative unit charge. Sodium chloride is an example of such a solid. The sum converges to a constant that is proportional to another constant known as the Madelung constant. Although the computer cannot perform an infinite sum, the infinite sum can be estimated by computing a finite sum. The focus of the third exercise is to approximately determine the Mandelbrot set in the region of the complex plane where $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. This continuous region is approximated by a $N \times N$ grid and these N^2 numbers are considered a representative sample all the complex numbers in the aforementioned region. For each such complex number c in the grid we repeatedly apply the formula $z^{'} = z^2 + c$ where the starting z value is 0. If the magnitude ever is greater than 2, the point is not considered a part of the Mandelbrot set.

2 Methods

2.1 Exercise 1

We convert the decimal number 100.98763 to a 32-bit float as follows. We convert the decimal number into binary and shift the binary point until the binary number is normalized. We then apply the following formula to get the 32-bit IEEE representation:

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

The term b_i represents the value of the i^{th} binary digit. This formula can be found in chapter one of *Computational Physics: Simulation of Classical and Quantum Systems* by Philipp Scherer. Finally, we compute the difference between the decimal number and the machine number.

2.2 Exercise 2

The goal of this exercise is to compute the Madelung constant, which reveals the total electric potential felt by an atom in a solid. We consider a three dimensional infinite simple cubic lattice, which we approximate as a finite lattice, where the sites alternate between one with net charge $+e$ and one with a net charge $-e$. We let the lattice spacing be a and consider each atom as a site with Cartesian coordinates (i, j, k). The atom at the origin will therefore have the coordinates $(i = 0, j = 0, k = 0)$. We consider the atom at the origin and calculate the Madelung constant, M, as follows:

$$M = \sum_{i,j,k=-L, i\neq 0, j\neq 0, k\neq 0}^{L} \frac{(-1)^{i+j+k}}{\sqrt{i^2 + j^2 + k^2}}$$

Here L is the number of atoms in all directions.

We compute the sum using two methods. In the first method, we use a nested for loop in order to compute the constant. We define a function called madelungForLoop whose input parameter is L. We store the parameter as a 32-bit integer and use linspace to construct a numpy array consisting of the integers from $-L$ to L, inclusive. We create a nested for loop whose iterative variables are called i, j, and k and their values range from $-L$ to L. We compute the aforementioned sum but do not consider the case where $i = j = k = 0$ by implementing

a conditional within the nested for loop. We print out the Madelung constant and the time needed to execute the nested for loop for a given L. We use another for loop to calculate and display the various approximations of the Madelung constant for various values of L. The goal was to determine convergence towards the theoretical value of -1.74. See, for instance, *An In-Depth Look at the Madelung Constant for Cubic Crystal Systems* by Grosso et al.; however, we stopped the program when execution times became too long.

For the second method, we implement a function called madelungMesh whose input parameter is the number of atoms in all directions. We use the input parameter and numpy's meshgrid to implement a three dimensional cube. We compute

$$\frac{(-1)^{i+j+k}}{\sqrt{i^2+j^2+k^2}}$$

on each point in the three dimensional mesh and calculate the sum to approximate the Madelung constant. We use another for loop to calculate and display the various approximations of the Madelung constant for various values of L. Execution times were faster and our code quickly computed a Madelung constant of about -1.74.

2.3 Exercise 3

First, we implement a $N \times N$ grid in the region $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. We then initialize arrays cList, the array of all c values which are just the grid points $x + iy$, and booleanArray, which will contain 1 (0) if the c value is (is not) in the Mandelbrot set. After that we define the constant magLimit, which is the maximum allowed limit of the magnitude after each iteration and initialize zlist, the array of all resulting complex numbers after each iteration of $z = z^2 + c$. This array will get updated in the for loop used to execute the iterations. During each iteration of the for loop, we define a local array called rightWrongMag. Its elements are 0 (1) if the transformation of c has magnitude greater than (less than or equal to) the magnitude limit 2. Transformations whose magnitudes are greater than 2 must not be considered in future iterations. This is done by setting the corresponding z-values and c-values to 0 so that any future iterations of $z = z^2 + c$ on these values have no effect and by setting the corresponding boolean value in booleanArray to 0. When everything is finished, we plot the Mandelbrot set.

2.4 Exercise 4

Here, we consider two ways to compute the roots of a quadratic polynomial: the standard formula and the conjugation of it. We then perform a unit test on the former method by using pytest. Application of pytest on the terminal was successful for the function that determined the roots of a quadratic polynomial by the standard formula.

3 Results

3.1 Exercise 1

We first convert the decimal numbers 100 and 0.98763 to binary numbers. We see that $100_{10} = 2^6 + 2^5 + 2^2 = 1100100_2$ and $0.98763_{10} \approx 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-9} + 2^{-10} + 2^{-12} + 2^{-14} + 2^{-16} + 2^{-18} + 2^{-20} = 1111110011010101010101_2$. Therefore we see that

$$100.98763_{10} \approx 1100100.11111100110101010101 = 1.10010011111100110101010101 \times 2^6$$

We see that there are 26 bits after the binary point and so we need to truncate the last three bits 101_2. We therefore have to round the bit before it, i.e. 0, to 1. Thus, the 23-bit mantissa is

$$10010011111100110101011_2$$

We also see that the exponent part of the single precision floating point number is $6 + 127 = 133 = 2^7 + 2^2 + 2^0$. Thus the 8-biy exponent is

$$10000101_2$$

Of course the sign bit is 0. **Hence 100.98763_{10} is stored as the single precision machine number**

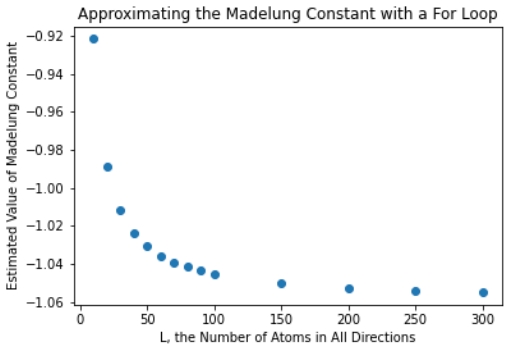
$$0 \;\; 10000101 \;\; 10010011111100110101011$$

This corresponds to the number $1.10010011111100110101011 \times 2^6 = 1100100.11111100110101011_2 = 100 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-9} + 2^{-10} + 2^{-12} + 2^{-14} + 2^{-16} + 2^{-17} = 100.9876327515$

The absolute value of the difference is about 2.751464×10^{-6}

3.2 Exercise 2

Below is a plot of estimated Madelung constants versus the number of atoms in all directions. The numbers do not approach the theoretical value for these small values of L, but the execution times become too long for me to run the for loop method on my laptop.



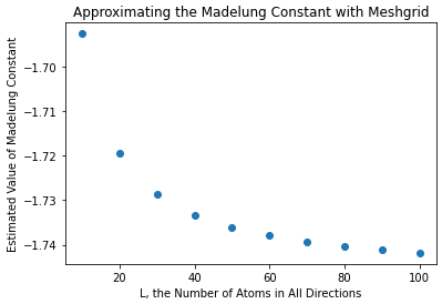
Plot of Madelung constants versus L with nested for loops.

Here is sample of the for loop execution times for the two largest L values. These times are around 16 minutes and 30 minutes, respectively.

```
Now starting L = 250
Time to run for loop is 982.567023399999 seconds.
Madelung Constant for L = 250 is
-1.05405759086443

Now starting L = 300
Time to run for loop is 1690.9607940999995 seconds.
Madelung Constant for L = 300 is
-1.0550174318932939
```

Execution times for L = 250 and 300 with Nested For Loops

Below is a similar plot for the meshgrid method. Even for small values of L the estimated values approach the theoretical value. The execution times are noticeably quicker. **For loops in python can be extremely slow.**



Plot of Madelung constants versus L with Numpy's Meshgrid.

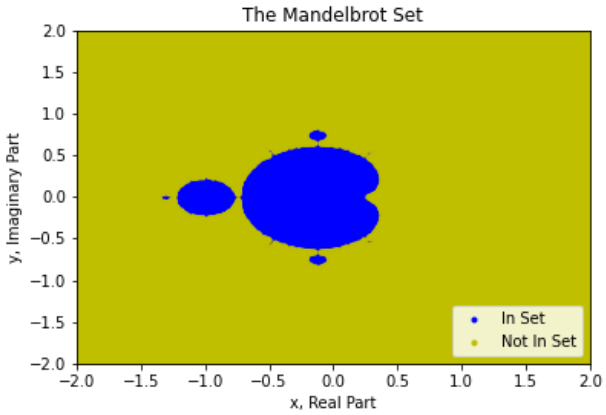
Here are the execution times for each L in the for loop. **Using meshgrid is much faster**.

```
It takes 0.000731199997909879 seconds to compute a Madelung constant of
-1.6925974

It takes 0.0029094999999870197 seconds to compute a Madelung constant of
-1.7193974

It takes 0.009987200000068697 seconds to compute a Madelung constant of
-1.7286364

It takes 0.02538639999966108 seconds to compute a Madelung constant of
-1.7332764

It takes 0.04300040000043737 seconds to compute a Madelung constant of
-1.7360691

It takes 0.07424120000041512 seconds to compute a Madelung constant of
-1.7379616

It takes 0.12756209999997736 seconds to compute a Madelung constant of
-1.7393326

It takes 0.21085549999997966 seconds to compute a Madelung constant of
-1.7403601

It takes 0.24693149999984598 seconds to compute a Madelung constant of
-1.7411863

It takes 0.3614910000001146 seconds to compute a Madelung constant of
-1.7418116
```

Execution times for L = 10,20,30,40...,80,90,100 with Meshgrid. The times are each a fraction of a second.

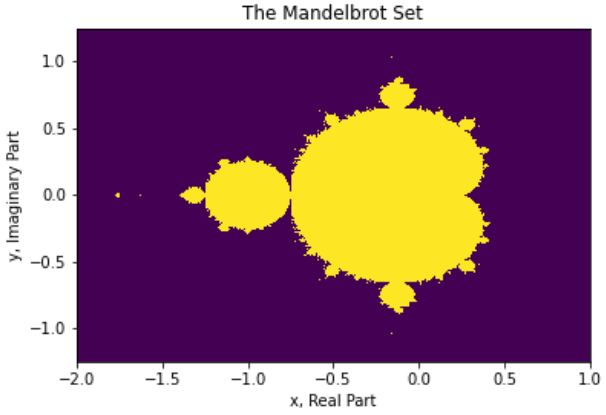
3.3 Exercise 3

Here is a zoomed out plot of the Mandelbrot set made from pyplot's plot function.



Mandelbrot Set zoomed out. The details of the set's periphery are not visible

Here is a zoomed in plot of the Mandelbrot set made from pyplot's pcolormesh function.



Yellow (Purple) points correspond to complex numbers that are in (not in) the set. The details of the set's periphery are more visible

3.4 Exercise 4

First we note that the standard quadratic formula can be rewritten in the equivalent version by rationalizing the denominator:

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a} \times \frac{-b \mp \sqrt{b^2-4ac}}{-b \mp \sqrt{b^2-4ac}} = \frac{b^2 - b^2 + 4ac}{-2ab \mp 2a\sqrt{b^2-4ac}} = \frac{2c}{-b \mp \sqrt{b^2-4ac}}$$

We can estimate the exact value of the roots for the case where $a = c = 0.001$ and $b = 1000$ by Taylor expanding the standard formula as follows:

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a} = -\frac{b}{2a} \pm \frac{b}{2a}\sqrt{1-\frac{4ac}{b^2}} = -\frac{b}{2a}\left(1 \mp \sqrt{1-\frac{4ac}{b^2}}\right)$$

$$\approx -\frac{b}{2a}\left[1 \mp \left(1 - \frac{4ac/b^2}{2} - \frac{(4ac)^2/b^4}{8} - \frac{(4ac)^3/b^6}{16}\right)\right]$$

$$= -\frac{b}{2a}\left[1 \mp \left(1 - \frac{2ac}{b^2} - \frac{2a^2c^2}{b^4} - \frac{4a^3c^3}{b^6}\right)\right] = -\frac{b}{2a}\left[1 \mp 1 \pm \frac{2ac}{b^2} \pm \frac{2a^2c^2}{b^4} \pm \frac{4a^3c^3}{b^6}\right)\right]$$

$$= -\frac{10^6}{2}\left[1 \mp 1 \pm 2 \times 10^{-12} \pm 2 \times 10^{-24} \pm 4 \times 10^{-36}\right]$$

When $\pm$ is set to $+$, the root is -1×10^{-6}. When $\pm$ is set to $-$, the root is -1×10^{6}.

The image below illustrates the roots calculated by Python by the two equivalent versions of the quadratic formula and the percent error between Python's values and the expected value we got through Taylor expansion.

```
In [1]: runfile('C:/Users/mooke/aGradComp/Prob4ab.py', wd
The roots obtained by the standard formula are:
-9.999894245993346e-07 -999999.999999
and the corresponding percent errors are:
0.001057540066535716 % and 1.00000761449337e-10 %

The roots obtained by the equivalent formula are:
-1.000000000001e-06 -1000010.5755125057
and the corresponding percent errors are:
1.0001341524705151e-10 % and 0.0010575512505718507 %
```

The standard formula has a smaller percent error for the root -10^6 and the equivalent formula had a smaller percent error for the root -10^{-6}

Absolute errors tell the same story as shown below.

```
The roots obtained by the standard formula are:
-9.999894245993346e-07 -999999.999999
and the corresponding absolute errors are:
1.057540066535716e-11 and 1.00000761449337e-06

The roots obtained by the equivalent formula are:
-1.000000000001e-06 -1000010.5755125057
and the corresponding absolute errors are:
1.0001341524705151e-18 and 10.575512505718507
```

Absolute errors behave the same way. Notice the large absolute error of 10.5755

The standard quadratic formula method passed when we used pytest but the equivalent version did not pass as shown in the images below. This makes sense since the unit test file test_quadratic.py tests whether the absolute errors are below a certain threshold and we see a very large absolute error when the equivalent method calculates the root -10^6.

```
C:\Users\mooke\aGradComp>pytest test_quadratic.py
============================================= test session starts =============================================
platform win32 -- Python 3.8.2, pytest-7.4.2, pluggy-1.3.0
rootdir: C:\Users\mooke\aGradComp
collected 1 item

test_quadratic.py .                                                                                  [100%]

============================================== 1 passed in 0.58s ==============================================
```

Unit test on standard method passes.

```
C:\Users\mooke\aGradComp>pytest test_quadratic.py
============================================= test session starts =============================================
platform win32 -- Python 3.8.2, pytest-7.4.2, pluggy-1.3.0
rootdir: C:\Users\mooke\aGradComp
collected 1 item

test_quadratic.py F                                                                                  [100%]

================================================== FAILURES ===================================================
_____________________________________________ test_quadratic _____________________________________________

    def test_quadratic():
        # Check the case from the problem
        arr = quadratic.quadratic(a=0.001, b=1000., c=0.001)
        assert (np.abs(arr[0] - (- 1.e-6)) < 1.e-10)
>       assert (np.abs(arr[1] - (- 0.999999999999e+6)) < 1.e-10)
E       AssertionError: assert 10.575513505726121 < 1e-10
E        +  where 10.575513505726121 = <ufunc 'absolute'>((-1000010.5755125057 - -999999.999999))
E        +    where <ufunc 'absolute'> = np.abs

test_quadratic.py:10: AssertionError
=========================================== short test summary info ===========================================
FAILED test_quadratic.py::test_quadratic - AssertionError: assert 10.575513505726121 < 1e-10
============================================== 1 failed in 0.22s ==============================================
```

Unit test on equivalent method fails. Notice the comment on large absolute error of 10.5755

4 Discussion

As noted in the introduction, the computation of a square root almost guarantees that there will be numerical errors due to truncation; however, there is another reason why we observe such a large absolute error when the computer uses the equivalent formula to compute the root -10^6. This root occurs when $\pm$ is set to $-$, or equivalently when $\mp$ is set to $+$. Consider the following formula, where we note that $b >> a = c$:

$$\frac{2c}{-b \mp \sqrt{b^2 - 4ac}} = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \approx \frac{2c}{-b + b}$$

The computer is adding two numbers with similar magnitudes and opposite signs and so errors will be amplified, as explored in section 2 of the first chapter of *Computational Physics: Simulation of Classical and Quantum Systems*. The absolute error will be quite large because this particular root has a large order of magnitude. For the same root, the standard formula does not have numerical errors due to adding numbers with similar magnitudes and different signs:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \approx \frac{-b - b}{2a}$$

In the other case, the root is -10^{-6}. This root occurs when $\pm$ is set to $+$, or equivalently when $\mp$ is set to $-$. Consider the following formula, where we note that $b >> a = c$:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \approx \frac{-b + b}{2a}$$

The standard formula has numerical errors due to adding numbers with similar magnitudes and different signs. The equivalent formula does not have this problem here:

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

This explains why the first method has larger errors, both absolute and percent, when computing this root.