PHYS-UA 210 Computational Physics
Problem Set 02

Sandhya Sharma

September 19, 2023

Question 1: Binary Representation of 121

The binary representation of 121 is 1111001.

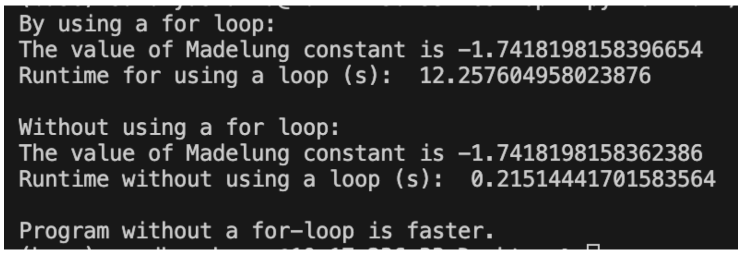
Question 2: Exercise 2.7 of Newman

The following code computes the Madelung constant for a sodium atom at origin in a sodium chloride lattice with 100 atoms in all x, y, z directions:

```python
import math
import numpy as np
import timeit

start1 = timeit.default_timer()
#function to calculate potential due to single atom at distance d
def potential(i, j, k):
    d = float(math.pow((i**2) + (j**2) + (k**2), 0.5))
    potential = (math.pow(-1, (i+j+k)%2))/d
    return potential

#fixing the number of atoms in x, y and z direction
L = 100

#using a for-loop
potential_sum = 0;

for i in range (-1*L, L+1):
    for j in range(-1*L, L+1):
        for k in range(-1*L, L+1):
            if i == 0 and j == 0 and k == 0:
                continue
            potential_sum += potential(i,j,k)

print("By using a for loop:")
print("The value of Madelung constant is " + str(potential_sum))

#finding runtime for using a loop
stop1 = timeit.default_timer()
print("Runtime for using a loop (s): ", stop1 - start1)
print()

#without using a for-loop
start2 = timeit.default_timer()

i = np.arange(-L, L + 1)
j = np.arange(-L, L + 1)
k = np.arange(-L, L + 1)

#calculate the sum array
sum_array = i[:, np.newaxis, np.newaxis] + j[np.newaxis, :, np.newaxis] + k[np.newaxis, np.n
sum_array = sum_array.flatten()

#calculate the distance array
i_squared = np.power(i, 2)
j_squared = np.power(j, 2)
k_squared = np.power(k, 2)

distance_array = i_squared[:, np.newaxis, np.newaxis] + j_squared[np.newaxis, :, np.newaxis
distance_array = distance_array.flatten()
distance_array = np.sqrt(distance_array)

#finding the index where distance = 0 and removing it from distance_array and sum_array
index_to_be_removed = np.argwhere(distance_array == 0)
index_to_be_removed = index_to_be_removed[0][0]

distance_array = np.delete(distance_array, index_to_be_removed)
sum_array = np.delete(sum_array, index_to_be_removed)

#calculating the individiual potential due to particular atoms and summing them
potential = np.power(-1, sum_array%2)/distance_array
result = np.sum(potential)

print("Without using a for loop:")
print("The value of Madelung constant is " + str(result))

#finding runtime without using a loop
```

```
stop2 = timeit.default_timer()
print("Runtime without using a loop (s): ", stop2 - start2)
print()

#determing which method is faster
if (stop2 - start2) < (stop1 - start1):
    print("Program without a for-loop is faster.")
else:
    print("Program with for-loop is faster.")
    print()
```

Output:



```
By using a for loop:
The value of Madelung constant is -1.7418198158396654
Runtime for using a loop (s):  12.257604958023876

Without using a for loop:
The value of Madelung constant is -1.7418198158362386
Runtime without using a loop (s):  0.21514441701583564

Program without a for-loop is faster.
```

Values for Madelung constant using two different algorithms and their runtimes

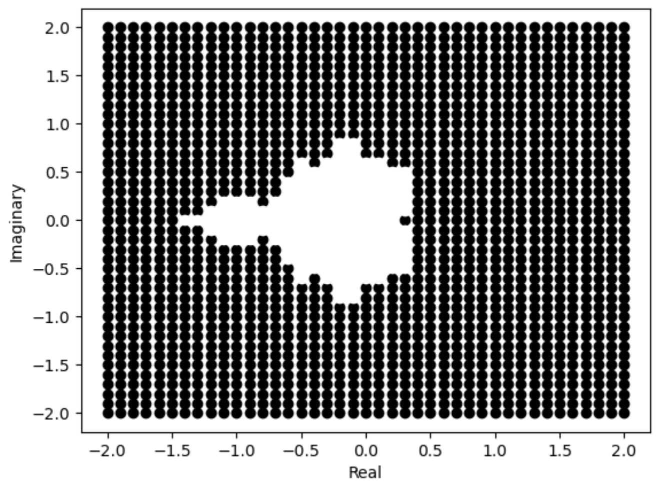
Question 3: Exercise 3.7 of Newman

The following code produces a plot of the Mandelbrot set for values of c for $-2 < x < 2$ and $-2 < y < 2$:

```
import math
import numpy as np
import matplotlib.pyplot as plt

#setting the range for real and complex values
x = np.arange(-2,2.1, 0.1)
y = np.arange(-2,2.1, 0.1)

#creating empty numpy arrays for z and c
z_array = np.empty(shape = (0,), dtype=complex)
c_array = np.empty(shape = (0,), dtype=complex)

#iterating over the Mandelbrot equation for every value of c
#and storing the values in z_array
iterations = 1000
```

```
for i in range(x.size):
    for j in range(y.size):
        c = complex(x[i], y[j])
        c_array = np.append(c_array, c)
        z = 0 + 1j*0
        for n in range(iterations):
            if np.abs(z) > 2:
                break
            z = z*z + c
        z_array = np.append(z_array, z)

#finding the magnitude of z for every value of c
z_magnitudes = np.abs(z_array)

#storing z values in two different arrays for points in and out of Mandelbrot set
c_in = np.empty(shape = (0,), dtype=complex)
c_out = np.empty(shape = (0,), dtype=complex)

for i in range(z_magnitudes.size):
    if z_magnitudes[i] > 2:
        c_in = np.append(c_in, c_array[i])
    else:
        c_out = np.append(c_out, c_array[i])

#plotting the Mandelbrot set
plt.scatter(np.real(c_in), np.imag(c_in), color = 'black')
plt.scatter(np.real(c_out), np.imag(c_out), color = 'white')
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.show()
```

Output:



Mandelbrot set

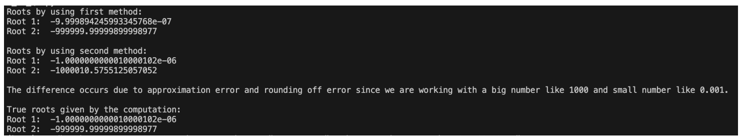
Question 4: Exercise 4.2 of Newman

The following code solves a given quadratic equation using two different methods and chooses the roots that are closer to the true values:

```
import math
import numpy as np
import random

a = np.float128(0.001); b = np.float128(1000); c = np.float128(0.001)

#finding roots using first method
x1_1 = np.float128((-b + math.sqrt((b**2) - (4*a*c)))/(2*a))
x2_1 = np.float128((-b - math.sqrt((b*b) - (4*a*c)))/(2*a))

print("Roots by using first method: ")
print("Root 1: ", x1_1)
print("Root 2: ", x2_1)
print()

#finding roots using second method
x1_2 = np.float128((2*c)/(-b - math.sqrt((b*b) - (4*a*c))))
x2_2 = np.float128((2*c)/(-b + math.sqrt((b*b) - (4*a*c))))

print("Roots by using second method: ")
```

```python
print("Root 1: ", x1_2)
print("Root 2: ", x2_2)
print()

print("The difference occurs due to approximation error and rounding off error")
print("since we are working with a big number like 1000 and small number like 0.001.")
print()

#comparing the roots given by both methods to the true roots given by Wolframalpha
x1_true = -1*(10**6)
x2_true = -1*(10**-6)

if abs(x1_1 - x1_true) < abs(x1_2 - x1_true):
    x1 = x1_1
else:
    x1 = x1_2

if abs(x2_1 - x2_true) < abs(x2_2 - x2_true):
    x2 = x2_1
else:
    x2 = x2_2

print("True roots given by the computation: ")
print("Root 1: ", x1)
print("Root 2: ", x2)
```

Output:



```
Roots by using first method:
Root 1:  -9.999894245993345768e-07
Root 2:  -999999.99999899998977

Roots by using second method:
Root 1:  -1.00000000000010000102e-06
Root 2:  -1000010.5755125057052

The difference occurs due to approximation error and rounding off error since we are working with a big number like 1000 and small number like 0.001.

True roots given by the computation:
Root 1:  -1.00000000000010000102e-06
Root 2:  -999999.99999899998977
```

Roots for the quadratic equation using different methods.

Please find my GitHub repository through this link.