ps-2

Greta Goldberg

September 19, 2023

1 Github Link

https://github.com/gretagoldberg/phys-ua210/tree/main

2 Question 1

For this question I simply divided 121 sequentially by 2 and then used the remainders as the binary representation.

3 Question 2

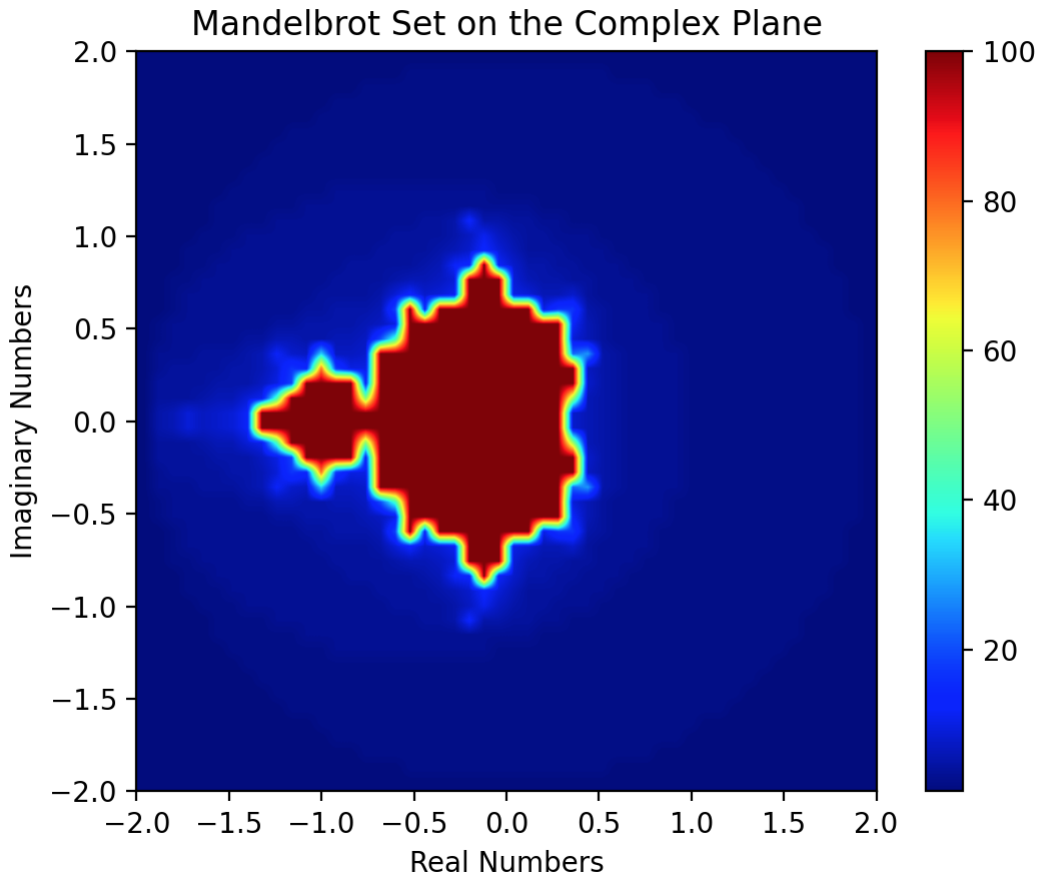
This code times the two ways of computing the sums. The first sum uses a for loop, and the second sum uses a meshgrid. The sign of the factor is determined by the even/oddness of the sum of coordinates.

```
import numpy as np
import timeit

def with_for_loop():
    sum = 0

    for i in range(-10,10):
        for j in range(-10,10):
            for k in range(-10,10):

            #check if the sum = 0
                if i == 0 and j == 0 and k == 0:
                    continue

                #if the conditions are met, add the value of M
                else:
                    sign = (-1)**((i + j + k )%2)
                    M = 1/np.sqrt(i**2+j**2+k**2)
                    total_factor = sign*M
```

```
                    sum += (-1)*total_factor

    print("first sum = ", sum)


def without_for_loop():
    #arrays for i,j,k
    i = np.arange(-10, 10, 1)
    j = np.arange(-10, 10, 1)
    k = np.arange(-10, 10, 1)

    #creating meshgrid
    I, J, K = np.meshgrid(i, j, k)

    #calculating the sum, excluding i=j=k=0 and making negative/positive depending
    on even/oddness
    mask = (I != 0) | (J != 0) | (K != 0)

    #calculating M for elements where the mask is True
    M = np.zeros_like(I, dtype=float)
    M[mask] = 1 / np.sqrt(I[mask]**2 + J[mask]**2 + K[mask]**2)

    #choosing sign based on even/odd condition
    total_factor = (-1)**((I + J + K) % 2) * M

    sum = np.sum(total_factor)

    print("new sum = ", (-1)*sum)

first_half_time = timeit.timeit(with_for_loop, number=1)
second_half_time = timeit.timeit(without_for_loop, number=1)

print("Execution time for the for loop = ", first_half_time, "seconds")
print("Execution time for the meshgrid version = ", second_half_time, "seconds")
```

4 Question 3

This code plots the mandelbrot set from -2 to 2 in both x and y. I again used a meshgrid to create the matrix of x and y values. I then masked any values that are greater than 2.




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

xmin, xmax = -2, 2
ymin, ymax = -2, 2

#creating values for the complex plane
x = np.linspace(xmin, xmax)
y = np.linspace(ymin, ymax)
X, Y = np.meshgrid(x, y)
c = X + 1j * Y

#initializing arrays to store mandelbrot set
z = np.zeros_like(c)
mandelbrot = np.zeros_like(c, dtype=np.uint8)
```

```
#defining N/number of iterations
N = 100

#looping through values and checking that they are less then 2
and then calculating output number
for i in range(N):
    mask = np.abs(z) < 2.0
    z[mask] = z[mask] ** 2 + c[mask]
    mandelbrot += mask

#plotting set with heatmap that tells how many iterations it took for c to be too large
plt.imshow(mandelbrot, extent=(xmin, xmax, ymin, ymax), cmap='jet', interpolation='bilinear'
plt.colorbar()
plt.title("Mandelbrot Set on the Complex Plane")
plt.xlabel("Real Numbers")
plt.ylabel("Imaginary Numbers")
plt.show()
```

5 Question 4

In this section of code I defined a function for the quadratic formula. I then rewrote the function as the question suggested and chose the most correct outputs from function 1 and function 2.

```
import numpy as np
import cmath

def quadratic_formula(a, b, c):
    if a == 0:
        return "Your a value is zero. You cannot divide by zero"
    else:
        discriminant = b**2 - 4*a*c
        if discriminant >= 0:
            x_positive = ((-1)*b+np.sqrt(discriminant))/(2*a)
            x_negative = ((-1)*b-np.sqrt(discriminant))/(2*a)
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        else:
            x_positive = ((-1)*b+cmath.sqrt(discriminant))/(2*a)
            x_negative = ((-1)*b-cmath.sqrt(discriminant))/(2*a)
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        return solution1, solution2
```

```
#first test solution
print(quadratic_formula(0.001,1000,0.001))

#quadratic formula in other form
def quadratic_formula_form2(a, b, c):
    if a == 0:
        return "Your a value is zero. You cannot divide by zero"
    else:
        discriminant = b**2 - 4*a*c
        if discriminant >= 0:
            x_positive = 2*c/((-1)*b+np.sqrt(discriminant))
            x_negative = 2*c/((-1)*b-np.sqrt(discriminant))
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        else:
            x_positive = 2*c/((-1)*b+cmath.sqrt(discriminant))
            x_negative = 2*c/((-1)*b-cmath.sqrt(discriminant))
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        return solution1, solution2

#second test solution
print(quadratic_formula_form2(0.001,1000,0.001))


#these give different solutions because of the way floats are stored
#this causes precision issues

#rewritten selecting the positive solution from quadratic_forumula and negative solution fr
def quadratic_formula_form3(a, b, c):
    if a == 0:
        return "Your a value is zero. You cannot divide by zero"
    else:
        discriminant = b**2 - 4*a*c
        if discriminant >= 0:
            x_positive = ((-1)*b+np.sqrt(discriminant))/(2*a)
            x_negative = 2*c/((-1)*b-np.sqrt(discriminant))
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        else:
            x_positive = ((-1)*b+cmath.sqrt(discriminant))/(2*a)
            x_negative = 2*c/((-1)*b-cmath.sqrt(discriminant))
            solution1 = "the positive solution to the quadratic is " + str(x_positive)
            solution2 = "the negative solution to the quadratic is " + str(x_negative)
        return solution1, solution2
```

```
#third test solution
print(quadratic_formula_form3(0.001,1000,0.001))
```