Homework 2

Maryam Manzoor Amanullah

September 21, 2024

1 Introduction

The homework provided a foundational introduction to how computers work in terms of numerical data representation. I covered key concepts related to how computers store numbers using different data types, such as integers and floating-point numbers, and how these representations can lead to various types of computational errors. For instance, floating-point arithmetic can result in rounding errors due to the limited precision with which computers store real numbers. This gave me a deeper understanding of issues like overflow and underflow, as well as how small numerical inaccuracies can propagate and affect the outcome of computations, especially when performing many operations in sequence.

The homework allowed me to explore practical tools in NumPy, a Python library for numerical computing. I learned how to use a wider range of NumPy functions for matrix operations and data manipulation.

I saw the comparison between using NumPy's vectorized operations and traditional for loops in Python. NumPy's operations are optimized for performance, often running in parallel, especially for matrix computations (such as the one done for Q3). By leveraging parallel processing, NumPy can handle operations on entire arrays or matrices simultaneously, leading to significantly lower computational time.

The directory within the github repository that contains the code for problem set 2 can be found here: https://github.com/maryam-manz/phys-ga2000/tree/main/PS2.

2 Methods

2.1 Question 1

There was a difference between the original number inputted into the computer and the floating point representation of the number. This difference arises because floating-point numbers can't precisely represent all real numbers due to the limited number of bits. In binary, numbers like '100.98763' can't be represented exactly and are represented as a fraction in the 32 bits of space because the binary representation has to be truncated after 23 bits in the mantissa. As a result, the number stored in memory was an approximation of the true value.

2.2 Question 2

In this homework question I found the smallest epsilon value by adding the epsilon to the 1 in 64 bit and 32 bit respectively and if the result was not 1, I would divide the epsilon by 2 and reiterate. I divided by 2 because in binary this is the way to get the next smallest number. Intuitively this can be seen as a bit shift to the right.

In the next part, I multiplied a large number by 2 to find the largest number before the computer reads the number as infinity. I did similar thing and divided a number by 2 until the computer read it as 0.

2.3 Question 3

Solving for the Madelung constant with the meshgrid function in NumPy took 1/10 of the time it took using for loops. This is because of how I can use vectorized computations in meshgrid. Vectorized computations refer to performing operations on entire arrays or matrices (vectors) at once, instead of using loops to perform operations element by element.

2.4 Question 4

I used similar operations for this question as I did in Question 3 when using meshgrid.

2.5 Question 5

This question was a great example of recognizing how rounding errors that occur in calculations can give different answers and need to be considered and taken into account.

In creating the function that would be able to solve any quadratic, I was sure to include cases where the determinant was negative. Additionally, to prevent rounding errors I considered two cases, one where b would be greater and equal to zero and the other when it would be less than zero.

3 Results and Discussion

3.1 Question 1

The libraries used for this question:
NumPy (for mathematical manipulation)
Struct (to convert and look at the binary representations of numbers)

Figure 1 shows the code I used to deconstruct the number to its binary form and then to reconvert it. Then to subtract the original number with the number reconverted from binary. The result is shown below in figure 2. As we can see there was a difference between the original and the reconverted number which occurred since the binary representation of fractions is not always exact.

```python
og = 100.98763

bit32 = np.float(og)



# 'f' is the format for 32-bit float                (variable) c: int      d int
binary_rep = ''.join(f'{c:08b}' for c in struct.pack('>f', bit32))

# print(binary_rep)

# Binary back to float
float_again = struct.unpack('>f', struct.pack('>I', int(binary_rep, 2)))[0]

# print(float_again)

difference = og - float_again

# print (difference)


print(f"Original number: {og}")
print(f"32-bit float number: {bit32}")
print(f"Binary representation: {binary_rep}")
print(f"32 bit float number: {float_again}")
print(f"Difference: {difference}")
```

Figure 1: The code written in python used to understand Question 1

```
Original number: 100.98763
32-bit float number: 100.98763
Binary representation: 01000010110010011111100110101011
32 bit float number after reconverting the binary: 100.98763275146484
Difference: -2.7514648479609605e-06
```

Figure 2: The output of the code

3.2 Question 2

The libraries used for this question:
NumPy (for mathematical manipulation)

In this question, I started by looking up the theoretical parameters of the data type numpy float32 and numpy float64 using the function np.finfo(dtype). The results are shown in figure 3.

For the precision test I wrote a function that iteratively decreases the value of epsilon and then adds it to 1 until the addition of epsilon doesn't cause the float 1 to change its value. That is the largest number you can add to the float 1 without it changing its value. The result for these epsillon value for both 64 and 32 bits are shown in figure 5

```
Machine parameters for float32
---------------------------------------------------------------
precision =   6   resolution = 1.0000000e-06
machep =    -23   eps =        1.1920929e-07
negep =     -24   epsneg =     5.9604645e-08
minexp =   -126   tiny =       1.1754944e-38
maxexp =    128   max =        3.4028235e+38
nexp =        8   min =        -max
smallest_normal = 1.1754944e-38   smallest_subnormal = 1.4012985e-45
---------------------------------------------------------------

Machine parameters for float64
---------------------------------------------------------------
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.2204460492503131e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
smallest_normal = 2.2250738585072014e-308   smallest_subnormal = 4.9406564584124654e-324
---------------------------------------------------------------
```

Figure 3: The output which shows the parameters of float32 and float 64

```python
# Precision Test (largest value added to 1 such that result is different from 1)
def find_epsilon(dtype):
    epsilon = dtype(1)
    while dtype(1) + epsilon != dtype(1):
        epsilon /= dtype(2) # divided by 2 since thats the binary value each digit is incremented by
    return epsilon * 2  # The last epsilon before the sum becomes equal to 1

# Finding epsilon for 32-bit and 64-bit floats
epsilon_32 = find_epsilon(np.float32)
epsilon_64 = find_epsilon(np.float64)

# Displaying results
print("Precision Test:")
print(f"Smallest epsilon for np.float32: {epsilon_32}")
print(f"Smallest epsilon for np.float64: {epsilon_64}")
```

Figure 4: Code for the precision test

```
Machine parameters for float32
---------------------------------------------------------------
precision =   6   resolution = 1.0000000e-06
machep =    -23   eps =        1.1920929e-07
negep =     -24   epsneg =     5.9604645e-08
minexp =   -126   tiny =       1.1754944e-38
maxexp =    128   max =        3.4028235e+38
nexp =        8   min =        -max
smallest_normal = 1.1754944e-38   smallest_subnormal = 1.4012985e-45
---------------------------------------------------------------

Machine parameters for float64
---------------------------------------------------------------
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.2204460492503131e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
smallest_normal = 2.2250738585072014e-308   smallest_subnormal = 4.9406564584124654e-324
---------------------------------------------------------------
```

Figure 5: The output for precision test for the float32 and float64

For the dynamic test I iterative multiplied a large 32bit and 64bit number with 2 respectively until it was defined as infinity, the value before infinity is the largest value for those number of bits. Similarly to find the smallest number i divided a small float by 2 until it was defined as 0, the number before that is the smallest value in the data type. The results are shown in 7. The values obtained from my code matched with the theoretical values found earlier.

```python
# Start with a large number and keep multiplying by 2 until overflow (approx inf)
# make sure everything is calculated in 32 bits and 64 bits respectively
print("Dynamic range test")

large_32 = np.float32(1e30)
large_64 = np.float64(1e30)


while not np.isinf(np.float64(large_64)): #the function checks if the value in the bracket is infinity
    max_64 = np.float64(large_64) # To store the value before it becomes inf
    large_64 *= 2

while not np.isinf(np.float32(large_32)): #the function checks if the value in the bracket is infinity
    max_32 = np.float32(large_32) # To store the value before it becomes inf
    large_32 *= 2


print(f"Maximum for np.float32: {max_32}")
print(f"Maximum for np.float64: {max_64}")
print(f"Minimum for np.float32: {-max_32}")
print(f"Minimum for np.float64: {-max_64}")

# Start with a small number and keep dividing by 2 until underflow (approx 0)
small_32 = np.float32(1e-30)
small_64 = np.float64(1e-30)

while np.float32(np.float32(small_32)) > 0:
    smallest_32 = np.float32(small_32)
    small_32 /= 2

while np.float64(np.float64(small_64)) > 0:
    smallest_64 = np.float64(small_64)
    small_64 /= 2

print(f"Smallest normal for np.float32: {smallest_32}")
print(f"Smallest normal for np.float64: {smallest_64}")
```

Figure 6: Code to find the largest and smallest value that can be represented in 32bit and 64bit

```
Dynamic range test
Maximum for np.float32: 2.6843546003927346e+38
Maximum for np.float64: 1.418129833677085e+308
Minimum for np.float32: -2.6843546003927346e+38
Minimum for np.float64: -1.418129833677085e+308
Smallest normal for np.float32: 1.401298464324817e-45
Smallest normal for np.float64: 5e-324
```

Figure 7: The output that shows the largest and smallest value obtainable before overflow and underflow

3.3 Question 3

The libraries used for this question:
NumPy (for mathematical manipulation)
Time (to measure the time taken by each method of calculation)

Initially to compute the Madelung I used for 3 nested loops in as seen in figure 8. The innermost loop iterates a total of $(2L)^3$ - 1 times. I also ignored the constants in my calculations.

```
# # Question 3 For Loops

# Madelung constant derrivation while keeping all other constants 1
start = time.time()

L = 100 # the number of atoms on each side

M = 0

i = 0
j = 0
k = 0



for i in range(-L, L+1):
    for j in range(-L, L+1):
        for k in range(-L,L+1):
            if i== j == k == 0:
                continue
            M += ((-1)**abs(i+j+k))/np.sqrt((i**2)+(j**2)+(k**2))


print(M)
end = time.time()

print(f'Time taken = {end-start}')
```

Figure 8: The for loops used to find Madelungs constant

```
-1.7418198158396654
Time taken = 38.67031121253967
```

Figure 9: Result using for loops

I then used the meshgrid function from numpy, to place each value in a matrix so that Numpy can carry out parallel calculations. In doing so we got the result in 1/100th of the time in the same calculations compared to for loops by comparing the results in łfig:$q_{3l}oop_outand11$.

```python
# # Question 3 without for Loops

start2 = time.time()
L = 100  # Number of atoms on each side

# Create arrays of i, j, and k values ranging from -L to L excluding 0
i_vals = np.arange(-L, L + 1)
j_vals = np.arange(-L, L + 1)
k_vals = np.arange(-L, L + 1)



# Use meshgrid to create 3D grids for i, j, k values
i_grid, j_grid, k_grid = np.meshgrid(i_vals, j_vals, k_vals, indexing='ij')

# Calculate the squared distance for each point (i^2 + j^2 + k^2)
dist_sq = np.float64(i_grid**2 + j_grid**2 + k_grid**2)

# Calculate the alternating sign (-1)^(i + j + k)
signs = (-1)**abs(i_grid + j_grid + k_grid)

# Create a mask to exclude the (0, 0, 0) point
mask = (i_grid == 0) & (j_grid == 0) & (k_grid == 0)

# Remove the zero distance points from the calculation (set dist_sq to avoid division by zero)
dist_sq[mask] = np.inf  # Set the value to inf so that value becomes 0 and isnt added to the sum

result = signs / np.sqrt(dist_sq)

# Sum all the values in the result array to get M
M = np.sum(result)

end2 = time.time()
print(M)
print(f'Time taken = {end2-start2}')
```

Figure 10: Using meshgrid to find Madelungs constant

```
-1.7418198158362388
Time taken = 0.44635581970214844
```

Figure 11: Reasult using meshgrid function

3.4 Question 4

The libraries used for this question:
NumPy (for mathematical manipulation)
Matplotlip (to visualize the Mandelbrot set)

I wrote the code to visualize Mandelbrot set by iterating over a grid of complex numbers and applying the Mandelbrot iteration function. I define a 100x100 grid (‘N = 100‘) and sets the number of iterations to 100. The arrays ‘x‘ and ‘y‘ are created using numpy.linspace, ranging from -2 to 2. Similar to last question, using meshgrid, the code generates a 2D grid of complex numbers ‘c_grid‘, where each point is constructed by adding the real and imaginary components. The Mandelbrot iteration is performed 100 times using the loop:

$$z_{n+1} = z_n^2 + c$$

After the iterations, the I check which points belong to the Mandelbrot set by verifying if the absolute value of z remains less than or equal to 2.

Finally, the results are visualized using matplotlib, where the Boolean array ‘ugh‘ is displayed in grayscale. Points inside the Mandelbrot set are represented in black, while points outside are shown in white. Figure 13

```
# # Question 4
# del j

N = 50 #NxN grid
iteration = 100

x = np.linspace(-2, 2, N)
y = np.linspace(-2, 2, N)


x_grid, y_grid = np.meshgrid(x, y, indexing='xy')

c_grid = x_grid + y_grid*1j

z = c_grid



for i in range(iteration):
    z_p = z**2 + c_grid
    z = z_p

ugh = (abs(z)<=2)


plt.imshow(ugh, cmap='gray',extent = [-2,2,-2,2])
```

Figure 12: Code to find the Mandelbrot set

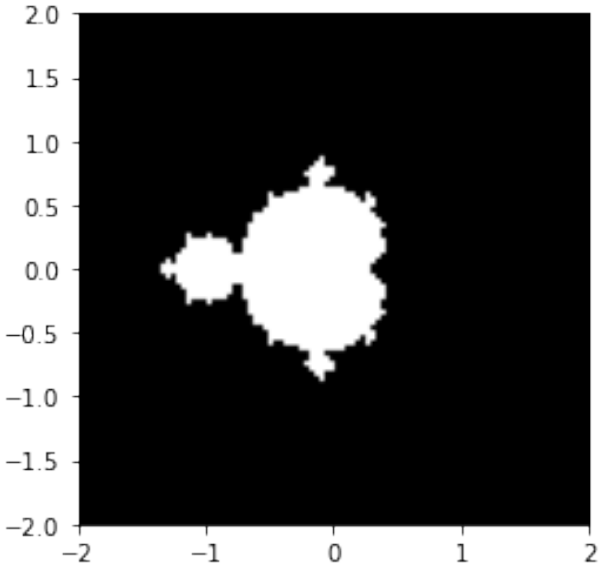

Figure 13: Image of the Mandelbrot set

3.5 Question 5

First I simply encoded the quadratic equations using python without taking any corner cases into account (Figure 14 and figure 15. There was a slight difference in the two solutions which occurred while subtracting, due to lack of precision in storing numbers as bits.

To amend this and to be able to solve teh quadratic equations in all cases, I wrote a set of functions that can give out the solutions for quadratic equations, when the determinant is negative, positive and 0. If the discriminant is negative, the function calculates the real and imaginary parts of the roots separately, and then returns the solution. If the discriminant is zero or positive, the function computes the real roots using the quadratic formula. To avoid potential issues of numerical precision loss, the code uses an alternate form of the quadratic formula depending on the sign of b. This approach prevents errors that can occur when b is large and close to the square root of the discriminant. (Figure 16)

```
print("Please enter the value for a")
a = float(input())
print("Please enter the value for b")
b = float(input())
print("Please enter the value for c")
c = float(input())


x1 = (-b+np.sqrt(b**2-4*a*c))/(2*a)
x2 = (-b-np.sqrt(b**2-4*a*c))/(2*a)
print(f'The two roots are : x1 = {x1} and x2 = {x2}')
```

Figure 14: Formula for the quadratic roots

```
x3 = 2*c/(-b-np.sqrt(b**2-4*a*c))
x4 = 2*c/(-b+np.sqrt(b**2-4*a*c))

print(f'The two roots are : x3 = {x3} and x4 = {x4}')
```

Figure 15: Alternate formula for the quadratic roots

```
x3 = 2*c/(-b-np.sqrt(b**2-4*a*c))
x4 = 2*c/(-b+np.sqrt(b**2-4*a*c))

print(f'The two roots are : x3 = {x3} and x4 = {x4}')
```

Figure 16: The module to solve quadratic equations

To test my module I wrote a test function shown in figure 17. This function tests each type of quadratic equation to ensure that my module can solve them. When run, we get the message

shown in figure 18 as the output indicating that the module passes the tests.

```python
import pytest
from quadratic import quadratic

def test_real_roots():
    # Test for real roots
    result = quadratic(1, -3, 2)  # Equation: x^2 - 3x + 2 = 0
    assert result == (2.0, 1.0) or result == (1.0, 2.0), f"Unexpected result: {result}"

def test_complex_roots():
    # Test for complex roots
    result = quadratic(1, 2, 5)
    assert result == (complex(-1, 2), complex(-1, -2)) or result == (complex(-1, -2), complex(-1, 2)), f"Unexpected result: {result}

def test_small_discriminant():
    # Test for case with a small discriminant
    result = quadratic(1, 1e8, 1)
    root1, root2 = result
    assert abs(root1 * root2 - 1) < 1e-10, f"Product of roots not close to expected value: {root1 * root2}"

def test_double_root():
    # Test for double root (discriminant = 0)
    result = quadratic(1, -2, 1)
    assert result == (1.0, 1.0), f"Unexpected result: {result}"
```

Figure 17: code for test function

```
(base) maryam_manz@10-20-22-22 PS2 % pytest test_quadratic.py

============================= test session starts ==============================
platform darwin -- Python 3.8.8, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /Users/maryam_manz/Downloads/NYU/NYU PhD/Academics/Fall 2024/Computationa
l Physics/PS2
plugins: anyio-2.2.0
collected 4 items

test_quadratic.py ....                                                   [100%]

============================== 4 passed in 0.31s ===============================
(base) maryam_manz@10-20-22-22 PS2 %
```

Figure 18: Output of testing function