Problem Set 2

Dylan Lane
Github: dl4729

September 20, 2023

1 Introduction

The binary implementation of numbers (principally the various forms of integers and floats) in computer systems is restricted by memory to be finite in precision. Floating point numbers, of interest in this project, are typically stored in 32 bits, the first being a sign bit (0 for positive, one for negative), 8 bits which determine the exponent, and the remaining 23 bits storing the mantissa and forming a base-2 scientific notation. To perform typical algebraic operations on two floats the computer must shift the exponent and correspondingly adjust the mantissa, however this can produce a loss of information as the least-order bits are eliminated in the adjustment (1) (2). Consequently, addition over large ranges and subtraction of comparable floats can be limited and these must be worked around to ensure accurate results.

In addition to numeric troubles, of concern to programmers is the temporal efficiency of algorithms. For large data sets, nested for loops can be inefficient because as the data size increases, the computation time increases by a power law depending on the number of loops. A remedy is the notion of vectorization, as a vector has elements which are easier to work with than typical data structures. In Python, because a NumPy array has a known datatype and interacts with other arrays in a more structured way than arbitrary lists, while the latter are limited to loops the former has much faster vector operations to leverage when doing computations. This project will employ vector operations for faster computations. (2)

2 Methods

Problem 1

Formulation of the problem

This problem is an analysis of the 32-bit representation of floating-point number 100.98763. No creative computational techniques are required; using the code provided in class, the float can be decomposed into its 32 bits and the output can be compared to the exact value. (2)

Problem 2

Formulation of the Problem

As mathematically described by the textbook, the goal is to assess the Madelung constant of a sodium chloride lattice structure at its center. The potential at the center of the lattice is simply the Coulomb potential

$$V(\vec{r}) = \frac{q}{4\pi\epsilon_0 a|r|} \tag{1}$$

. In a lattice, the potential position vectors are integer linear combinations of the basis vectors $\hat{i}, \hat{j}, \hat{k}$. Importantly, the sodium and chlorine atoms have opposite charges as they interact ionically; the sodiums impart a positive term to the potential as they are positively charged, while the chlorine atoms impart a negative. If the origin is assumed to be sodium, the sign of q is simply $(-1)^{i+j+k}$, and q's magnitude is always $e/$. By superposition, the total potential is:

$$\sum_{i,j,k} \frac{(-1)^{i+j+k} e}{4\pi\epsilon_0 a\sqrt{i^2+j^2+k^2}} = \frac{Me}{4\pi\epsilon_0 a} \tag{2}$$

where the case $i = j = k = 0$ is excluded; from this the constants cancel and it becomes apparent that M is simply a coordinate dependent sum. (1)

Computational methods

For an exact result, the lattice points should extend infinitely. Obviously this is not computationally possible so we can bound the box size to have side lengtth $2L$ and pick some large L such that M can be reasonably accepted. For the sake of computational time, $L = 100$ is used.

There are two ways to perform the summation; a triple for loop to iterate over all positions of lattice points and accumulate the potentials from each individual point. Alternatively, the NumPy meshgrid method can be used to automatically get all integer linear combinations of basis vectors, and then that can be summed over simply with array methods (3). The hypothesis is that the latter will be faster. In the for loop implementation, having direct access to i, j, k allows an immediate rejection of the $i = j = k = 0$ case, however with meshgrid I omit this handling and simply convert the infinity that NumPy outputs in the initial pass to a zero before summing.

Problem 3

Formulation of the Problem

The Mandelbrot Set in the complex numbers is the set of numbers c such that, starting from $z = 0$, repeated iterations of the map$z' = z^2 + c$ do not exceed modulus 2. In theory this should be an infinite sequence, but for the purpose of computability we will restrict ourselves to finite iterations (1).

Computational methods

There are not terrible restraints on the problem except for limiting overflows and using vectorization. In particular, those values of c such that z eventually explodes need to be handled to curb strange overflow behavior. To solve this, I simply applied the condition that if a particular c had an iteration which exceed modulus 2, I set it to infinity (this created some additional difficulties which will be noted in the discussion section). Beyond this, creating a 2D array of zeros, adding the 2D array of c complex numbers, then squaring on this copied array is a fairly concise way of performing the iterations and the Mandelbrot set can be extracted simply by array comprehensions.

Problem 4

Formulation of the Problem

On its face, this is a fairly trivial problem; given $a, b, c \in R$, compute the roots of the quadratic $ax^2 + bx + c = 0$. This of course has the solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

. Multiplying by the conjugate of the numerator divided by itself,

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

which, after simplifying, comes to

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \tag{5}$$

(1). The key point of interest is that subtraction of similar-order numbers can have substantial roundoff error; in the case of $|b| >> |ac|$, in either of these methods one of the roots (the one which involves addition of two positive numbers or subtraction of two negative numbers respectively) is fine, but the other (with opposite signs) will have roundoff (1). This will be demonstrated in the results section.

Computational methods

As noted, the major issue in this problem is selecting which quadratic formula to use for a given situation. I assumed that $|b| >> |ac|$ so the size of the square root term and the b term can be compared directly. Following my thought that the signs of the two terms should be the same, I separated by solution into a $b > 0$ and $b < 0$ case; in the positive case, we have

$$x_1 = \frac{-b - \sqrt{b^4 - 4ac}}{2a} \tag{6}$$

$$x_2 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{7}$$

. In the negative case, we instead have

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{8}$$

$$x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{9}$$

3 Results

Problem 1

In 32-bit representation, it can be found that 100.98763 = 01000010100100111111100110101011; broken down into components this is sign= 0 (positive), exponent 10000101 = 133 in decimal, and mantissa = 00100111111100110101011. This can be compared to the "true" value to good approximation by finding the difference between the 32-bit value and the 64-bit value; this turns out to be $-2.7514648479609605e-06$, the sign indicating that the 32-bit slightly overestimates the true value.

Problem 2

For L=10, the for loop method outputs $M = -1.6925789282594415$ in 0.07919692993164062 seconds. and for L=100, $M = -1.7418198158396654$ outputted in 69.56296110153198 seconds.

The meshgrid method with L=10 outputs $M = -1.692578928259451$ in 0.0006151199340820312 seconds; with L=100, $M = -1.7418198158362388$ outputted in 0.47038888931274414 seconds.

Problem 3

I chose to use a 1000x1000 grid to cover the square grid whose sides run from -2 to 2 in the complex plane. After running for 1000 iterations and allowing the colors to correlate to the number of iterations before the modulus of z exceeded 2 (darker means fewer iterations, the Mandelbrot set has an iteration number of infinity), figure (1) was generated:

Problem 4

The combined code outlined in the Methods section passed the test code provided by Professor Blanton; proof of the pass is shown in figure (2).

4 Discussion

Problem 1

This demonstrates the limited precision of floating points in Python and how the particular bit representation can impact the value of a number by a nontrivial amount.

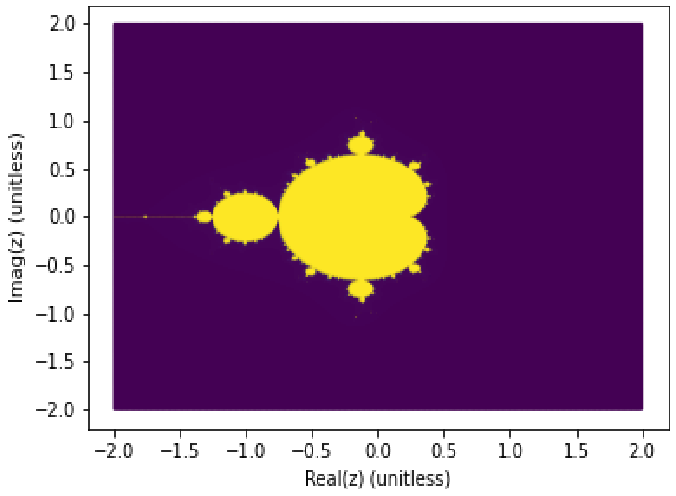
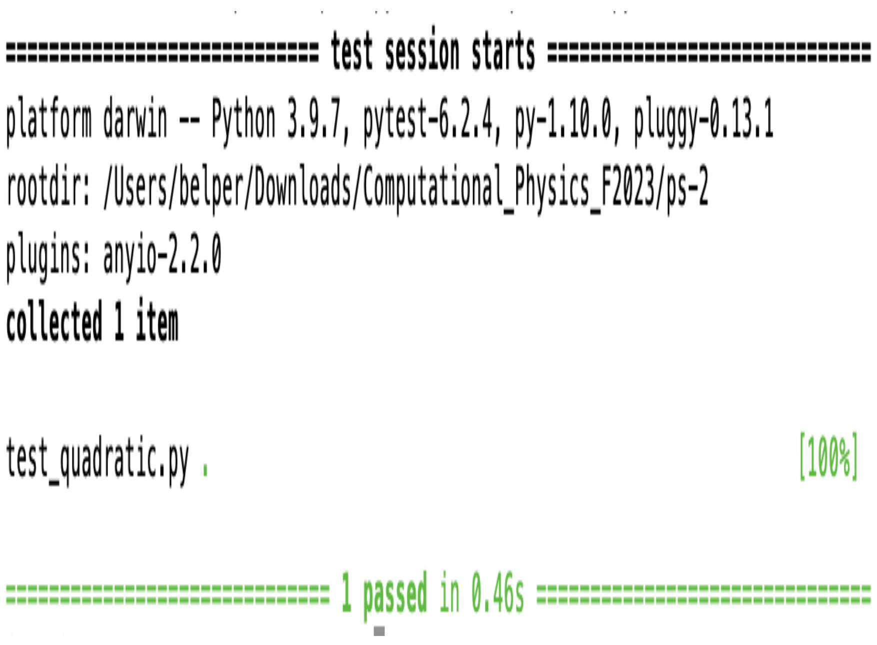


Figure 1: Mandelbrot set; N=1000, 1000 iterations, coloring according to iterations required before leaving the set. Axes are complex plane axes (unitless).



```
============================ test session starts =============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /Users/belper/Downloads/Computational_Physics_F2023/ps-2
plugins: anyio-2.2.0
collected 1 item

test_quadratic.py .                                                      [100%]

============================ 1 passed in 0.46s ===============================
```

Figure 2: Proof of quadratic code passing unit test

Problem 2

The data confirm fairly conclusively that vectorizing Python processes is a huge help in keeping computations manageable. On the one hand, there is no accuracy cost when employing array operations over looping; the values given by iteration and by vector operations are in agreement to 11 decimal places. In the time domain, the array implementation far exceeds the nested for loop approach; for L=100, the arrays were 150 times faster than iterating over each point and making an individual calculation.

Problem 3

There is nothing particularly novel about this solution but it affirms the value of array operations in NumPy. A double for loop over a very large subset of the complex plane would take a long time, comparable to the time for problem 2 except repeated for 1000 iterations. This is untenable. NumpyArrays can be directly squared and have access to element-wise addition, making the code for finding the Mandelbrot set very straightforward.

Problem 4

As stated, the code which combines the two forms of the quadratic formula works on the provided test cases. However, it must be noted that the typical quadratic formula with no accounting for subtraction roundoff error also passes. Looking at the raw values, the corrected one is closer to the true values (of course the shared value is the same, but the normal quadratic formula varies only in the fifth digit of the Mantissa (about the 12th actual digit accounting for order of magnitude of -7) and is therefore sufficiently close to pass. It is likely that if one created a more extreme example, the roundoff error could be forced into a regime where the correct code still holds but the naive implementation fails simply because the combined code is more accurate; in this sense the more elaborate piece of code is still an improvement.

References

[1] Newman, M. 2012, Computational Physics (Createspace Independent Pub)

[2] `https://blanton144.github.io/computational-grad/`

[3] `https://github.com/mcmorre/computational_TA/blob/main/PS2_hints.ipynb`