Ps-2

Jaewoo Lee

September 19, 2023

Abstract

This document contains results of ps_2. The code written to solve the problems could be found in my github: patrick612. Codes for solutions to all probelms could be found in ps-2.py except for 4), which has its own pyhton file named quadratic.py. The module test of quadratic.py also has been done

1

The IEEE 754 standard floating point number in 32 bits use 1 sign bit, 8 bits for the exponent and 23 bits as mantissa. Therefore, Numpy's 32-bit floating point representation of the number 100.98763 is 01000010110010011111100110101011. The floating point representation differs from the actual value by -2.7514648479609605e-06.

2

The function madelung_for uses for loop iteration from -l to l over all three dimensions while the function madelung_vect uses numpy.meshgrid to create arrays containing lattice indices along each dimensions then applies operations onto numpy arrays to calculate the potential. To test the time difference, I ran each function with l = 10 thousand times then averaged the time for the function to run. Function using for loop took about $30.6 * 10^{-3}$ seconds while the function using the array took $0.127 * 10^{-3}$ seconds per function call. The array method was about 241 times faster than the for loop method.

3

The Mandelbrot plot was generated using two iteration loops as shown in Figure 1. Mandelbrot function given as $z' = z + c$ where c is a constant and starting z is taken as 0, is repeated over some number of iterations by feeding z from the previous iteration. Mandelbrot plot was drawn over [-2, 2] for Re(c) and Im(c). Within a 100 iterations of Mandelbrot function for some c, if the modulus of the resulting complex number did not exceed 2, c is marked True for belonging to the Mandelbrot set.

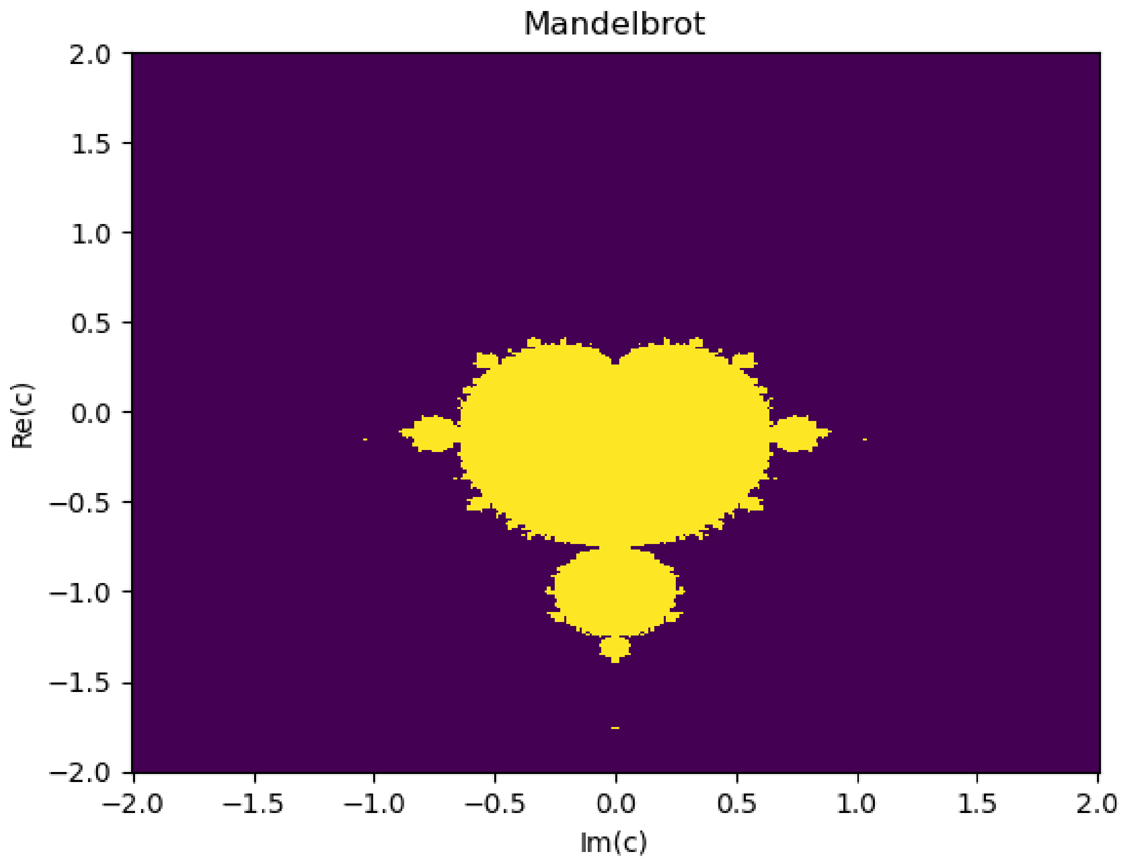
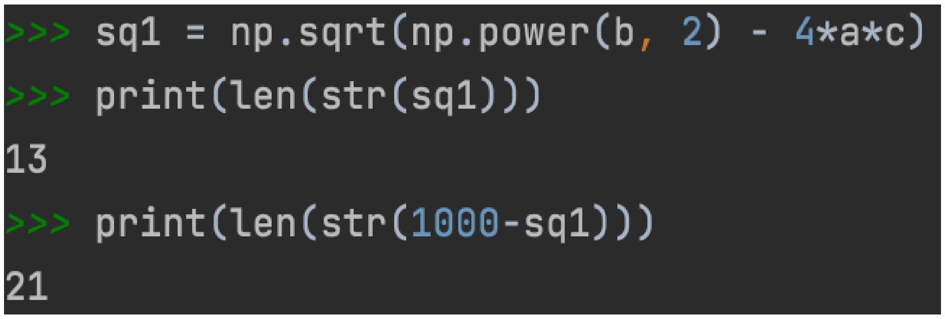


Figure 1: Yellow area corresponds to region that yielded True for the Mandelbrot function while purple corresponds to False



```
>>> sq1 = np.sqrt(np.power(b, 2) - 4*a*c)
>>> print(len(str(sq1)))
13
>>> print(len(str(1000-sq1)))
21
```

Figure 2: The numerator of the quadratic root equation for a = 0.001, b = 1000, c = 0.001 is represented by 21 digits

4

Two roots found for $0.001x^2$+$1000x0.001$ by solving for the equation in a) is (-9.999894245993346e-07, -999999.999999) while the roots using equations in b) is (-1.000000000001e-06, -1000010.5755125057). The difference in number comes from the fact that certain number of significant digits is used to represent a floating number. Specifically, for the first root with method a) term $b \pm \sqrt{b^2 - 4ac}$ is truncated to its significant figures then divided by 0.002 where as with method b) 0.002 is divided by $b \mp \sqrt{b^2 - 4ac}$ then the significant figure is taken. The reason why the resulting roots are similar but not exact is due to the difference in order of operations in division and taking the significant digits.

For part c), I have written a function, quadratic_corrected(), that compares a total number of digits on the denominator. For example we see that $b - \sqrt{b^2 - 4ac}$ is represented by a number of 21 digits as shown in Figure 2 above. If we put this in the denominator, we will be diving by a truncated number up to 16 significant digits. On the other hand, if this term were in the numerator, we will be dividing this number by 0.001 then take the significant digit truncation, propagating less error. The code I've written compares the length of digit of $b \pm \sqrt{b^2 - 4ac}$ to the length of digits of $2a$ and puts the number with less digits on the denominator to propagate less error.

Finally we have also confirmed that the module passed the module test.