Computational physics
Homework 2
14 Sep 2023

Name: *P. Demidov*
GitHub: PalPaulD

Problem 1

Let's introduce some useful notation (not sure whether it is widely used in the US, but for our purposes it is a must). If we have a number in base x, we will write this base as an index of the number. If there is no index, assume that it is 10 by default. For example, $5_{10} = 5 * 10^0 = 2^2 + 1 * 2^0 = 101_2$.

So far so good, but, unfortunately, problems are not solved by notation agreements. Let's try to massage our precious 100.98763. How is it stored in PC's memory?

1. The sign bit is 0, $(-1)^{sign} = +1$.

2. To get the exponent, we need to rearrange the number: $100.98763 = 2^6 * (1 + 0.57793171875)$. Keep in mind that this equality is exact. Then the exponent is $E - 127 = 6$, even I can solve this equation: $E = 133_{10} = 10000101_2$. I assume that converting from binary to decimal and vice versa is a well-known procedure.

3. Lastly, we want to convert 0.57793171875 to get the mantissa. Well, maybe this is the reader's favourite number, but not mine: it can't be represented as a finite binary number. It can be seen if we try to convert this fraction to binary.

   The algorithm of converting fractions that I know suggests to divide this number by 2 *and if the integer part becomes equal to* 1*, it takes* 1 *as the first digit the binary fraction, subtracts* $1.(0)$ *from the number itself and repeats the procedure. If the integer part stays* 0*, then we must set the first number in the binary fraction is* 0 *and repeat the procedure again. The procedure stops when the number becomes integer.*

   If we do the procedure several times:

$$
\begin{gathered}
0.57793171875 * 2 = 1.1558634375, \\
0.1558634375 * 2 = 0.311726875, \\
0.311726875 * 2 = 0.62345375, \\
0.62345375 * 2 = 1.2469075, \\
0.2469075 * 2 = 0.493815, \\
0.493815 * 2 = 0.98763, \\
0.98763 * 2 = 1.97526, \\
0.97526 * 2 = 1.95052, \\
0.95052 * 2 = 1.90104, \\
0.90104 * 2 = 1.80208, \\
0.80208 * 2 = 1.60416, \\
0.60416 * 2 = 1.20832, \\
\dots
\end{gathered}
$$

(ok, maybe a bit more than several times) we will see that after some point the last digit is stuck in a loop: $2 \to 4 \to 8 \to 6 \to 2 \to \ldots$ (last digits of powers of 2, right? $2, 4, 8, 16, 32, \ldots$) . And there is no way of getting out of this loop with multiplications by 2 and if we want to get rid of the fractional part, we need to get 0. Hence, the procedure never stops and the number has infinitely many digits in the binary system.

This means that the initial number 100.98763 can't be stored exactly in a machine, sad news. But with 32-bit precision we know that the mantissa is $m = 10010011111100110101011_2$. Red digits are the digits that we've explicitly seen in the conversion above.

I hope you enjoyed long numbers in this part, but some typos could spoil your experience. They are not meant to be there, but my eyes were dazzled when I was writing this, so missprints are possible. I have used some code to make this conversion sustainable to my typos, check it out in hw2, prob1, supplementary code.py.

So we know that our *approximated* number is encrypted with s, E, m by the formula

$$100.98763_{10} \approx (-1)^s \times 2^{E-127} \times (1+m) = \left(2^6\right)_{10} (1_2 + 0.10010011111100110101011_2)$$

And we can convert the approximation back to the decimal:

$$2^6 \left(1 + 2^{-1} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} + 2^{-18} + 2^{-20} + 2^{-22} + 2^{-23}\right)$$

Each decimal fraction in this sum is finite, so the summation can be done exactly with mathematica, for example. Even wolfram alpha is capable of doing that. The result is

$$\frac{13236651}{131072}$$

Then we can subtract 100.98763 from this and get the answer: $100.98763 - \text{float32}(100.98763) = 2.75146 * 10^{-6}$. Note that this is the exact value of error.

Problem 2

We need to compute the sum:

$$M = \sum_{i,j,k=-L}^{L} \pm \frac{1}{\sqrt{i^2 + j^2 + k^2}},$$

where $\pm$ turns in a "+" if $i + j + k$ is odd and in a "$-$" if $i + j + k$ is even. Before we even touch the code, we'd better rewrite the sum to avoid this word fuss with signs:

$$M = \sum_{i,j,k=-L}^{L} \frac{(-1)^{i+j+k}}{\sqrt{i^2 + j^2 + k^2}}$$

So much better now.

Two versions of the code developed to calculate M. In the first version (hw2, problem2.py) the function $V(i,j,k) = \frac{(-1)^{i+j+k}}{i^2+j^2+k^2}$ is tabulated on a grid (i,j,k) and numpy's .sum() method is used to calculate the Madelung constant. To avoid troubles with $(0,0,0)$, we set $V\left[0,0,0\right]=0$ in the code after tabulation, so no infinities are sumed. The result of this computation for $L=100$ is

$$M \approx -1.74494.$$

Module "time" is used to determine execution time, %timeit seems to be a jupyter notebook thing, it didn't work for me. This result was obtained in $t_1 = 2.7$ seconds.

The second version of the code (hw2, problem2 (slow).py) relies on three nested for-loops. I wonder, if our space had 26 dimensions (strings say hello), then would we have to write 26 nested loops in this problem? Anyway, this time the problem of the origin is more severe: we do not tabulate the function any more (instead of that we just sum it term by term), so we can't set $V(0,0,0)=0$. My way around this was to use try-except to catch division by zero and continue the loop without adding an infinity. It works, but it works in a weird way.

If one tries to use np.sqrt(), they will not catch anything. Numpy makes python treat division by zero with a warning, but not as an error, so things do not work. This can be fixed by using "np.seterr(all = 'raise')".

However, I had no idea about this fix for some time, so after some thought I decided to use "**(1/2)" instead of np.sqrt(). This modification made my calculation 3 times faster (4.51s vs 13.72s)! And I am not talking about the fact that it throws an error when dividing by zero. I am not sure about the precision, but from this we learn that over-hyped numpy is not always better than its competitors.

This way of computing the sum gives

$$M \approx -1.741819,$$

in 4.51s, it is close to what we had before but slower (sometimes numpy does a good job).

Problem 3

Setup

A couple of notes before we start discussing my code and results.

To start with, let us introduce proper notation. We call a Mandelbrot sequence the sequence defined as

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

for any $c \in \mathbb{C}$ and $z_1 = 0$. There is a theorem that if $|c| > 2$, then a sequence of absolute values $|z_n|$ always diverges, that's why we are not asked to take care of any other numbers. However, there is another tricky possibility. Assume that for some $|c| <= 2$ we have managed to find a number $N \in \mathbb{N}$, such that $|z_N| > 2$. Will the sequence $|z_n|$ always stay above the value of 2? Or we can possibly find another $N' > N$, such that $|z_{N'}| < 2$? If it can happen, we might face problems with picking up extra numbers that are not elements of the set if we finish calculations too early. If it can happen, I am supposed to check $|z| > 2$ every time and exclude numbers before they go back under the limit. Frankly, it sounds painful in terms of coding and I don't want to get a headache from that. So there is a way around this if we do some mathematics.

Yes, if the sequence $|z_n|$ grows slowly enough, then we might fail to find some numbers that should be in the set, but this is a problem that I can't deal with. And if you can't solve the problem, why bother?

However, it turns out that for $|c| <= 2$ it never happens, so if $|z_n|$ goes above 2 once, it never gets back. In other words,

Claim. Assume that for some $|c| <= 2$ there is $N \in \mathbb{N}$, such that $|z_N| > 2$. Then for any $N' > N$ we will have $|z_{N'}| > 2$.

Proof. Consider

$$\frac{|z_{N+1}|}{|z_N|} = \frac{|z_N^2 + c|}{|z_N|} \geq |z_N| - \frac{|c|}{|z_N|}.$$

We know that $|z_N| > 2$ and $\frac{|c|}{|z_N|} <= 1$, so $\frac{|z_{N+1}|}{|z_N|} \geq 1$. Hence,

$$\frac{|z_{N+1}|}{|z_N|} \geq 1,$$

so $|z_{N+1}| > 2$. Then apply the same procedure to z_{n+1} and get that $|z_{N+2}| > 2$ and so on. □

So there is no chance that I pick up numbers that I am not supposed to see in the set. Maybe I will lack some, but... Nevermind. Anyway, instead of checking that $|z| > 2$ at each step, I can check this only after I finish with my iterations. That's a nice thing to know.

Solution

Speaking about the code, there is a fancy (not really, see later) way to solve the problem without using any loops at all. In the code (hw2, problem3.py) there is a function called **no_loops_calculation** that iterates the Mandelbrot sequence M times with recursion (so loops are not needed here at all), where M is defined at the beginning of the code. To make this function work, I had to import sys-module, so I can overlock the recursion limit with sys.setrecursionlimit(N*M). Not sure whether it is a feature of my system or everybody has to do that.

This function can be applied to the grid of complex numbers, where the total amount of steps is equal to N+1, where N is defined by hand. The return of the function is compared to 2, thanks to numpy we can compare the whole (N+1)×(N+1) array just with "<2". Then the result is plotted via imshow(...). Note that we need to set extent = $[-2, 2, -2, 2]$, otherwise we will have a rescaled Mandelbrot set. The outcome of execution is the following:

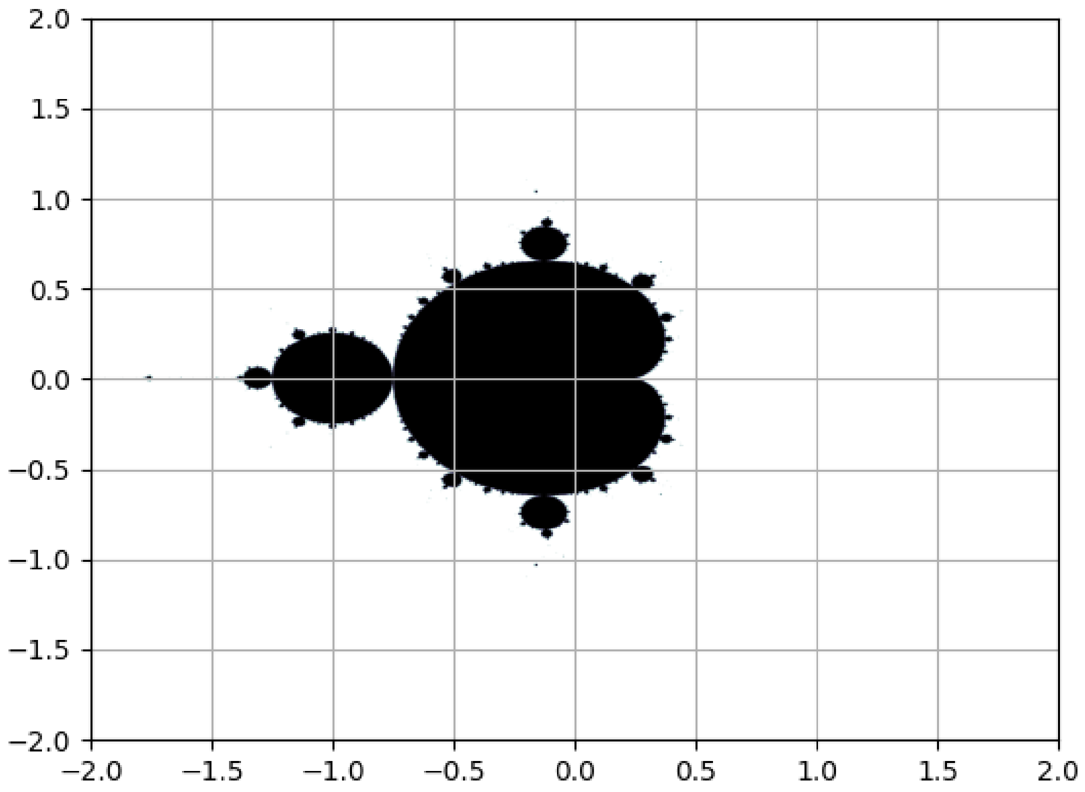

Figure 1: Calculated Mandelbrot set for N=1000, M=500

To loop, or not to loop, that is the question

One might think that getting rid of the only loop brings joy and speed to computations, but it doesn't turn out to be the case. There is another function in the code called **one_loop_computation**, it works just in the same way, but it uses a for-loop for iterations. To see whether it is faster or not, the module "time" was used to measure computational time. The code was executed several times with different values of M (I could vary N, but conceptually it will not give any new insights), the table of execution times (t_{ex}) with N=500 is below:

M	for-loop t_{ex}, seconds	recursion t_{ex}, seconds
10	0.04	0.04
100	0.39	0.51
250	0.9	1.01
500	1.74	2.28
1000	3.6	6.53
5000	20.14	48.17

For-loop wins! The lesson from this should sound like **doing a loop N times is always faster than doing if-else N times**. Instructive.

Problem 4

(a)

Let's try to solve this equation by ourselves. No calculators, only brute force. Consider

$$\sqrt{b^2-4ac} = b\sqrt{1-\frac{4ac}{b^2}} \approx b - \frac{2ac}{b^2} + O\left(\frac{ac}{b^2}\right) \approx 10^3 - 2*10^{-9} + ...,$$

where we've used the limit $\frac{4ac}{b^2} \ll 1$ for $a = 10^{-3}, b = 10^3, c = 10^{-3}$. So we see that the discriminant deviates from 10^6 in the negative direction if we hold only first-order corrections (second order is $\propto 10^{-18}$, so this is a good approximation). We will keep that in mind.

The roots take the form

$$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

(who would have thought?), so we encounter a tricky arithmetic operation:

$$-10^3 \pm \sqrt{10^6 - 4*10^{-3}*10^{-3}} \approx \left[\begin{array}{l} \dfrac{-10^3 + (10^3 - 2*10^{-9})}{2*10^{-3}}, \\ \dfrac{-10^3 - (10^3 + 2*10^{-9})}{2*10^{-3}}, \end{array}\right.$$

What could go wrong? Notice that we in the first case we have to calculate a difference of two very close numbers ($-b$ and $\sqrt{b^2-4ac}$). This is a problem, because this subtraction will lead to a big loss in precision (the desired result is too small to be obtained in this operation with large numbers). This effect is known as **catastrophic cancellation**: the relative error is huge in this case.

Subtracting numbers does not always cause big rounding errors, sometimes you can make it even exact, thanks to a Sterbenz lemma:

Theorem. (Sterbenz lemma) In a floating-point number system with subnormal numbers (this is exactly our case with IEEE), if two floating-point numbers satisfy

$$\frac{y}{2} \le x \le 2y,$$

then $x - y$ is a floating-point number, i.e. can be computed exactly. But it's not our case, because our numbers cannot be represented exactly in IEEE, so rounding errors still occur when computer converts float numbers.

In the second case our numbers are not close to each other, so our relative rounding errors turn out to be much smaller. We see this happening in results of the code (checkout quadratic.py file, solve() function) that uses this formula to compute roots. The first root (with a "$+$" sign) deviates (relatively!) from the true result more than his colleague with a "$-$" sign).

(b)

However, it is not the end of the world, we can get rid of this plus sign:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a} \times \frac{-b \mp \sqrt{b^2-4ac}}{-b \mp \sqrt{b^2-4ac}} = \frac{2c}{-b \mp \sqrt{b^2-4ac}}$$

Now for the first root we don't need to subtract close numbers anymore, so precision grows. But the second root falls in the trap that first root was in: we subtract close numbers and get rounding issues.

In the same program there is also a calculation that uses this formula and one can see that x_1 is much closer to the true answer, but x_2's business is not doing exactly good.

(c)

To come up with a solver, one needs to be careful with signs of b. We are working in $\mathbf{R}$, so square root is always non-negative. This means that if $b < 0$, we should change our choice of formulas for roots. In this case we will have the best precision if we use

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

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

so then $-b$ and $\sqrt{b^2 - 4ac}$ have the same sign and no catastrophic cancellation happens.

Then another script called test_quadratic.py imports this code and runs tests successfully (it even sends a message for you to know about that).