Computational Physics / PHYS-GA 2000 / Problem Set #2

Zak Saeed
(GitHub: Zakariah-S)

September 2024

Abstract

I present the attempted solutions to various problems meant to explore the strengths and limitations of the NumPy library. The first two problems deal with the limitations of 32-bit and 64-bit floats in NumPy. The third and fourth problems demonstrate how NumPy array functionalities can be exploited to optimise calculations. The fifth problem explores the limitations imposed by subtraction errors.

Introduction

Today's computers are extraordinarily powerful, and thus an excellent tool for physicists to perform numerical calculations. However, even the best computers have limitations that must be understood in order to use them effectively. The following work details my solutions to the problems of Computational Physics Problem Set #2, which explore the aforementioned limitations as well as the uses of arrays in NumPy.

There are five problems in the problem set. For each problem, a brief description of the setup and the approach taken to solve the problem will be provided in Section **??**. The code used to solve each problem is present in the GitHub repository phys-ga2000.

1 Conversion Errors in 32-bit

The first problem explores how the number 100.98763 is represented in the IEEE-standard 32-bit format. As it turns out, the number's representation in standard 32-bit is not exactly the same value as its decimal form. The same is true for its 64-bit and 128-bit representations, though these standards are able to more closely approximate the number.

1.1 Methods

As a first attempt, I used NumPy's `frexp` function to decompose the 32-bit float into the mantissa m and an exponent E such that the original number equaled $m \times 2^E$. Decomposing the number in this way and then reconstructing it yielded the decimal value that was actually represented by the original number's 32-bit representation.

In the pursuit of a more exact represented value, I used Michael Blanton's code for finding the bits reflected by a NumPy float. The value 100.98763 was stored as a `numpy.float32` object that was then decomposed by Blanton's code into its sets of sign, exponent, and mantissa bits. Denoting the value of the sign and exponent bit-collections as s and E, and letting f_i (for $0 \leq i \leq 22$) denote the bits of the mantissa from most to least significant, the decimal form of the represented number can be calculated using

$$(-1)^s \times \left(1 + \sum_{i=0}^{22} f_i 2^{(i-23)}\right) \times (2^{E-127}). \tag{1}$$

Decomposing the 32-bit float into its component bits and reconstructing its decimal value allowed me to calculate the actual represented number as an exact fraction. The code used to solve this problem can be found in `ps-2-1.py`.

1.2 Results

The first method, using `numpy.frexp()`, indicated that the 32-bit representation of 100.98763 corresponds to the number 100.98763275146484. This "true" value has an absolute difference of $2.7514648479609605 \times 10^{-6}$ from the original number. The fractional difference $\frac{r-o}{o}$, where r is the represented value and o the original, is equal to $2.724556312452288 \times 10^{-8}$.

The second method, using Blanton's code, returned the "true" value as the fraction $\frac{13236651}{131072}$. When computed, this fraction becomes 100.98763275146484– the same value as from the first method. The absolute and fractional errors are also the same.

2 Single vs. Double Precision

The second problem asks us to estimate the smallest number that can be added to 1., with the sum being a number different than 1., in both 32-bit and 64-bit precision. It also asks us to estimate the minimum and maximum positive numbers that can be represented with these two levels of precision.

2.1 Methods

For the first part of this problem, I reasoned that the target number, in both representations, was likely a power of 2. To understand why, suppose this is not the case, i.e. that the binary-form mantissa of our "smallest add-able number" n looks like .000..0001...1 (with the ellipses on the left being populated by an arbitrary number of zeroes, and the ellipses on the right having arbitrary values within). Removing the rightmost figure yields a number that is both smaller than n and guaranteed to also be "add-able" to 1., because the most significant bit is unchanged. In fact, removing every figure to the left of the most significant one makes the number smaller and retains the number's property of being "addable" to 1.

Hence, the function I write simply initialises a value `addend` at $2^0 = 1.$ (in whatever float format we're working in) and adds this value to 1. If the result is different from 1., then `addend` is halved and the process is repeated. This occurs until the sum is the same as 1., in which case the

last value of `addend` is returned. The function repeats this procedure for subtraction, finding the smallest power of 2 that you can subtract from 1. while getting a different number.

For the second part of this problem, I reason that the largest number in each representation will be the largest possible power of 2 plus a series of the next largest powers of 2. Multiplying out Equation 1 and assuming $s = 0$, we get

$$2^{E-127} + \sum_{i=0}^{22} f_i 2^{(E+i-150)}. \tag{2}$$

This expression can be maximised by first maximising E and then maximising each f_i in order from the highest- to lowest-significant bit. However, if each of these bits is set to 1 then the number will be assigned to infinity by NumPy. Thus, bits should only be flipped until NumPy converts the number to this value. My function uses this reasoning to find the maximum 32-bit and 64-bit numbers, i.e. it first finds the highest possible value of 2 and then keeps adding successively lower powers of 2 until the value `numpy.inf` is reached.

To find the lowest positive number, I reason that this number should be subnormal, i.e. its 32-bit decimal representation would be calculable as

$$\left(\sum_{i=0}^{22} f_i 2^{(i-23)} \right) \times (2^{-126}) \tag{3}$$

(again assuming that $s = 0$). Then the lowest possible number must be a power of 2, and in fact should be 2^{-149} for 32-bit floats. I use the same reasoning for 64-bit floats. Thus, my function for finding the lowest positive numbers simply initialises a value at 1. and then divides by 2 until the smallest possible value is reached. At this point, further division by 2 prompts NumPy to set the result to 0, and the function's loop breaks.

As it turns out, NumPy also provides a built-in class for finding all of the above information for each data type. I compare the results of this method with my own in Section 2.2. The code used to solve this problem can be found in `ps-2-2.py`.

2.2 Results

When considering only the addition of positive values to 1., the first method finds that the smallest number that makes a difference is $1.1920929 \times 10^{-7} = 2^{-23}$ for 32-bit floats and $2.220446049250313 \times 10^{-16} = 2^{-52}$ for 64-bit floats. However, when we also think about subtraction, we find that the smallest number we can subtract from 1. is a factor of 2 smaller than the result we get from just considering addition. For 32-bit floats, the smallest number we can subtract from 1. is $5.9604645 \times 10^{-8} = 2^{-24}$. For 64-bit floats, the smallest number is $1.1102230246251565 \times 10^{-16} = 2^{-53}$.

My program computed the smallest positive 32-bit float to be $1 \times 10^{-45} \approx 2^{-149}$, and the smallest positive 64-bit float to be $5 \times 10^{-324} \approx 2^{-1074}$. The largest floats were found to be $3.402823669209385 \times 10^{38}$ in 32-bit and $1.7976931348623157 \times 10^{308}$.

As previously stated, `numpy.finfo` can be used to get each of the values calculated above. For the most part, the values given by `numpy.finfo` match my results up to six significant figures. For

Value	Computed	NumPy-given
Smallest addable (32-bit)	1.1920929×10^{-7}	$1.1920928955078125 \times 10^{-7}$
Smallest addable (64-bit)	$2.220446049250313 \times 10^{-16}$	$2.220446049250313 \times 10^{-16}$
Smallest subtractable (32-bit)	5.9604645×10^{-8}	$5.960464477539063 \times 10^{-8}$
Smallest subtractable (64-bit)	$1.1102230246251565 \times 10^{-16}$	$1.1102230246251565 \times 10^{-16}$
Smallest number (32-bit)	1×10^{-45}	$1.401298464324817 \times 10^{-45}$
Smallest number (64-bit)	5×10^{-324}	5×10^{-324}
Largest number (32-bit)	$3.402823669209385 \times 10^{38}$	$3.4028234663852886 \times 10^{38}$
Largest number (64-bit)	$1.7976931348623157 \times 10^{308}$	$1.7976931348623157 \times 10^{308}$

Table 1: Calculated values representing the range of 32-bit and 64-bit floats in NumPy.

each of the 64-bit calculations, the match is exact (possibly because the round-off errors are too small to be captured in the digits printed by NumPy).

The main discrepancies occur with the 32-bit calculations: here, round-off errors may be preventing my code from achieving the same precision as the output of `numpy.finfo`. In the case of the smallest 32-bit number, almost all precision was lost as a result of the solving function repeatedly dividing the result value by 2, successively removing significant non-zero bits. I verify in my code that taking `numpy.log2(`1×10^{-45}`)` still returns the value -149, suggesting that my code leads to the same number contained in `numpy.finfo`, only with much less precision. The calculated values– both mine and those given in NumPy, are summarised in Table 1.

3 Newman 2.9: The Madelung Constant for NaCl

The Madelung constant M quantifies the potential felt by an atom in a crystal lattice– for instance, a sodium atom in sodium chloride. Assuming a perfect, infinite grid-like lattice, with distances a between each atom, we can pick out a sodium atom and sum up the electric potentials due to all of the atoms around it. This results in a converging sum of potentials with form

$$V(i,j,k) = \pm \frac{e}{4\pi\epsilon_0 a\sqrt{i^2+j^2+k^2}}, \tag{4}$$

where i, j, and k denote the distance along the x, y, and z axes from the target atom in units of the distance a. Because e, ϵ_0, and a remain the same for each term of the sum, they factor out and the result can be simplified to

$$V_{total} = \sum_{i,j,k=-\infty}^{\infty} \frac{e \times even(i+j+k)}{4\pi\epsilon_0 a\sqrt{i^2+j^2+k^2}} = M \times \frac{e}{4\pi\epsilon_0 a}, \tag{5}$$

where the function $even(x)$ returns 1 if x is even and -1 if x is odd. It should be noted that the term corresponding to $i = j = k = 0$ is not included in this sum. The sum can be approximated by

Method	1	2
Calculated M	-1.74181981583961481	-1.7418198158361038
Absolute Error	0.005780184160385282	0.0057801841638962514
Fractional Error	0.0033074983751346316	0.0033074983771436547
Run time (s)	19.478233301	0.6514724299999983

Table 2: Estimated values of Madelung's constant. The time taken to find the Madelung constant is also shown for each method.

having i, j, and k each vary from -L to L, for some sufficiently large integer L. Thus, we write that

$$M \approx \sum_{i,j,k=-L}^{L} \frac{even(i+j+k)}{\sqrt{i^2+j^2+k^2}} \equiv \sum_{i,j,k=-L}^{L} m_{ijk}. \tag{6}$$

3.1 Methods

A simple method of approximating the Madelung constant is to simply nest three for loops in which i, j, and k respectively iterate from -L to L. For each iteration, m_{ijk} is added to an accruing sum that eventually converges to M. However, the procedure can be done much more quickly by utilising the built-in functionality of NumPy arrays. For a given integer L, the 3D space can be modelled using 3 $(2L+1)\times(2L+1)\times(2L+1)$ arrays, which we can name `I`, `J`, and `K`. The corresponding elements of each array form a set of $(2L+1)^3$ ordered pairs (i, j, k), each corresponding to the position of one of the atoms in the lattice. The array `signs`, populated with the values $even(i+j+k)$, should also be constructed. Then, the array of all m_{ijk} can be built simply by computing `signs * np.power(i**2 + j**2 + k**2, -1/2)`. Finally, the Madelung constant is returned by performing a sum over this array. The code written for this problem can be found in `ps-2-3.py`.

3.2 Results

Both of the methods detailed in Section 3.1 were run for $L = 100$, i.e. for a cubic lattice with 201 atoms along each side. Method 1, the method that used nested for loops, returned the result $M_1 = -1.74181981583961481$. Method 2, the method written using no explicit Python loops, returned $M_2 = -1.7418198158361038$. These results, along with their errors with respect to the true NaCl Madelung constant $M \approx -1.7476$, are recorded in Table 2 [Glasser, 2016].

Python's timeit module was used to check which of the two methods was faster. The measured run times recorded in Table 2 are also for calculations with $L = 100$. According to these run times, the second method runs almost 30 times faster than the first, for an error that is practically the same!

4 Newman 3.7: The Mandelbrot Set

The Mandelbrot set is a famous fractal that occurs in the complex plane of numbers. It arises when we consider the recurrence relation

$$z_n = z_{n-1}^2 + c, \tag{7}$$

where $z_0 = 0$ and c is a constant, assigned value. The Mandelbrot set is the set of all c such that $|z_n(c)| \leq 2$ for all z_n [Newman, 2013]. We can approximate this set by repeating this iteration for a large set of numbers in the complex plane. As the set is mostly confined to the region $[-2, 2] \times [-2, 2]$ (and because the problem asks me to), I iterate 100 times over a grid of 1000×1000 points spanning this range.

4.1 Methods

For this problem, I use the `numpy.complex128` float type. I initialise two 1000-element-long arrays using `numpy.linspace(2, 2, 1000`. I then use `numpy.meshgrid()` with these to construct 1000×1000 arrays `x` and `y`, where the corresponding elements of each array can be composed into an ordered pair (x, y) within the region $[-2, 2] \times [-2, 2]$. I then initialise an array `c` to represent the grid of complex numbers in this region, constructing it simply by taking `x + y * 1j`. Finally, I apply the recurrence relation 100 times, using a marker to "block out" points whose $|z_n|$ values eclipse 2. The surviving set of points is plotted using Matplotlib. In a second, more colorful attempt at this problem, I use the same marker method to instead mark down the number of the iteration at which each point dropped out of the set. Plotting these gave a much more interesting picture. The code written for this problem can be found in `ps-2-4.py`.

4.2 Results

The results of the calculation are shown as plots in Figure 1.

5 Newman 4.2: Finding the Roots of a Quadratic

This problem asks us to write several solver functions that find the roots of a quadratic $P(x) = ax^2 + bx + c$. Specifically, we investigate the case where $a = c = 0.001$ and $b = 1000$ [Newman, 2013].

5.1 Methods

The first part of the problem asks us to solve for the roots of this quadratic using the quadratic formula

$$x_{12} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{8}$$

I implement this by dealing with the cases of real and complex numbers separately. If the value $b^2 - 4ac$ is negative, I use `np.complex128` numbers for a, b, and c. For real numbers, I use only

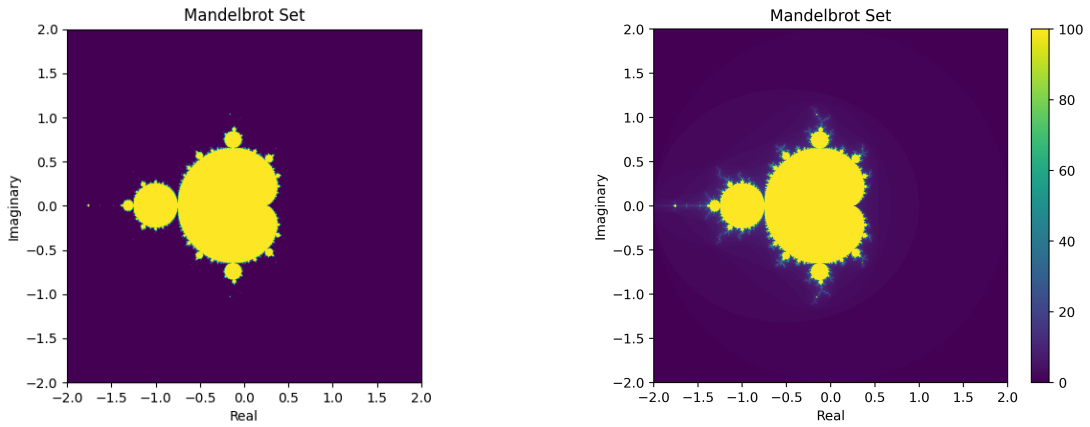

Figure 1: Left: Plot of the Mandelbrot set, as defined in the introduction to this section. Points within the set are coloured yellow, while all other points are coloured purple. Right: A plot of the same set, with each point coloured according to the iteration number at which it dropped out of the set Points that stayed in the set for all 100 iterations are simply coloured according to the color value at 100.

instances of `np.float64`. I then simply append the solutions x_1, x_2 as calculated using Equation 8 to an array which is then returned.

The second part of the problem asks us to use the alternative formula

$$x_{12} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \tag{9}$$

My solver function for this part has the exact same form, except for the fact that it uses Equation 9 instead of 8 to find the roots.

Finally, the problem asks us to identify the issue arising with both solver functions, and to devise a third solver that fixes this issue. I talk about this further in the following results section. The code for this problem can be found in `quadratic.py`. The file `test_quadratic.py` was written and used to verify that the final solver function worked correctly.

5.2 Results

The first solver function, which applies Equation 8 to a, b, and c as defined above, returns the root values $x_1 = -1.00000761 \times 10^{-6}$ and $x_2 = -1. \times 10^{6}$. These values are close to the correct ones; plugging them back into the quadratic gives $P(x_1) = -7.61449237 \times 10^{-9}$ and $P(x_2) = 7.24792480 \times 10^{-8}$ respectively. However, the solutions given by an ideal solver function would return 0. when plugged into the quadratic.

The second solver function, which uses Equation 9, returns the roots $x_1 = -1. \times 10^{-6}$ and $x_2 = -1.00001058 \times 10^{6}$. While these roots are close (though not the same) as the ones returned by the first solver, they also aren't perfect. Plugging them back into the quadratic yields $P(x_1) = 0.$

and $P(x_2) = 10575.62534721$ respectively. While the first root is therefore correct (or at the least the closest we can get), the second root is clearly off the mark.

Looking at the formulas and considering the values of a, b, and c, it becomes apparent that the problem lies with high errors caused by the subtraction $-b + \sqrt{(}b^2 - 4ac)$. With the given values, $b^2 >> 4ac$, meaning this subtraction is between b and a number very close to b. Thankfully, the two forms of the quadratic equation give us a convenient solution to this problem: for each solution, just find and use the formula where you aren't performing a subtraction between close numbers. Using this strategy, I wrote a function that, for positive b, calculates the solutions as

$$x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{10}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{11}$$

For negative b, the formulas for x_1 and x_2 are swapped, and the sign of the determinant is changed in both formulas. Solving the problem in this way eliminates the large subtraction errors that hampered the other two solvers.

The resulting function returns the roots $x_1 = -1 \times 10^{-6}$ and $x_2 = -1 \times 10^{6}$ which, when plugged into the $P(x)$, give $P(x_1) = 0.$ and $P(x_2) = 7.24792480 \times 10^{-8}$. Still not perfect, but pretty close!

Conclusion

The exercises in this problem set, while a great introduction to the utility of computers and NumPy in performing calculations efficiently, also did well to show how some very basic limitations can affect the work we try to do in unexpected ways. After finding the limits of 32-bit and 64-bit numbers, one might expect to avoid them completely by respecting this dynamic range and only working with values that lie well within its borders. However, something as simple as a subtraction of variables can destroy the precision of a calculation. Understanding this source of errors and others is therefore pivotal for obtaining meaningful numerical solutions.

References

Leslie Glasser. Solid-state energetics and electrostatics: Madelung constants and madelung energies, Feb 2016. URL `https://acs.figshare.com/collections/Solid_State_Energetics_and_Electrostatics_Madelung_Constants_and_Madelung_Energies/2513560/1`.

M.E.J. Newman. *Computational Physics*. CreateSpace Independent Publishing Platform, 2013. ISBN 9781480145511. URL `https://books.google.ae/books?id=SS6uNAEACAAJ`.