Problem Set 3

Debendro Mookerjee
GitHub Account Name: Debendro-dm5790

September 25, 2023

Abstract

This problem set continues the exploration of numerical errors that were first considered in the previous problem set, but also considers approximation errors which are errors due to the use of approximated formulas. In addition to the topic of error control, some of the exercises discuss efficiency and program speed. The first problem falls into the former category of errors and the remaining ones fall into the latter topic of efficiency.

1 Introduction

When writing and testing a program, it is important to find a compromise between error minimization and computational speed. The Python timeit module provides a suitable way to determine how long the computer takes to execute a block of code and to compare the speeds of multiple equivalent programs. Knowing the various execution times will allow us to determine the algorithm that runs for the appropriate time and gives tolerable errors. For instance, one can use this module to realize that vectorization, which can be implemented by the numpy package's meshgrid, is faster than using nested for loops to evaluate a finite sum to the same degree of accuracy. This was explored in the previous problem set when we computed the Madelung constant of NaCl. The timeit module is used in this problem set to determine whether matrix multiplication is efficiently calculated with nested for loops or the dot() function.

We will first describe the methods we used to answer the four exercises and then we will provide the suitable outputs.

2 Methods

2.1 Exercise 1

In this exercise, we consider the following approximation formula for the first derivative of a single variable function $f(x)$:

$$\frac{df}{dx}|_{x=x_0} \approx \frac{f(x_0+\delta)-f(x_0)}{\delta}$$

Our function is the quadratic polynomial $f(x) = x(x-1)$ and $x_0 = 1$. We first implement a python function called derivativeApprox() that approximates the derivative of $f(x)$ at $x = 1$.

The function takes two parameters, value, which is the x value at which the derivative is approximated, and delta, which is the small deviation from the input x value. The function returns an approximate value of the derivative at the input x value as a single precision floating point number. We note that this code inherently has approximation error.

We also implement a variable called deltaVal and an numpy array called deltaList. The first variable stores the deviation used in the first part of the problem as a single precision floating point number. This variable has a value of 0.01. The array stores the deviations used in the second part of the problem as single precision floating point numbers. These deviations are $10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$, and 10^{-14}.

For the first part of the exercise **the actual value of the derivative is** 1 **since** $\frac{df}{dx} = 2x - 1$, but the computer gives 1.01. **There are three main sources of this error**:

1. Addition of a large number (ex: $x_0 = 1$) and a small number (ex: $\delta_0 = 0.01$) will lead to a loss of precision. This will become more prominent as delta values become smaller and smaller.
2. The addition of two numbers, $x_0 + \delta$ and -1, with similar magnitudes but differing signs will amplify numerical errors.
3. Approximation error due to the approximate derivative formula.

We implement a for loop to determine the computer's estimate of the derivative for the various deviations found in the array deltaList. **Since delta is getting smaller, we should expect some improvement in the derivative calculations; however, as delta gets smaller precision should decrease due to reasons** (1) **and** (2).

2.2 Exercise 2

We implement a function that performs matrix multiplication with a nested for loop. The function takes the matrix dimension N as an input and creates three N x N matrices called A, B, and C. Matrices A and B are matrices filled with ones while C is a matrix filled with zeros. The goal is to calculate C = AB. The for loops fill up the elements of the matrix C. We use the python module timeit to determine the time it takes the computer to determine the matrix C. This time is stored in the variable called timeElapsed. The function returns timeElapsed.

We implement a second function that performs matrix multplication with numpy's method dot(). The function takes the matrix dimension N as an input and creates two N x N matrices of all ones called A and B. The goal is to calculate C = AB. We again use the python module timeit to determine the time it takes the computer to determine the matrix C. This time is stored in the variable called timeElapsed. The function returns timeElapsed.

We create an array NArray of 32-bit integers whose elements are the various matrix dimensions we want to test. We consider the following dimensions: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, and 600. We use a for loop to iterate through all of these dimensions. For each iteration, we call the first and second functions and get the times it took the computer to calculate the matrix product with for loops and numpy.dot(). These times are stored in the arrays timeArrayForLoop and timeArrayDot, respectively.

We make two log-log plots. In the first (second) plot we plot the matrix dimensions and the

execution times for the for loop (numpy.dot()). For each plot, we perform a power law fit to determine the appropriate function of the form time = $a*$(matrix dimension)^p that fits the data. We then determine the fitted value of the powers. This allows us to numerically compare the empirical power with the expected power. **We expect the power to be three in each case and we get a value close to three**.

2.3 Exercise 3

We implement the initial number of Bismuth-213, Thallium, Lead and Bismuth-209 atoms. These are stored in the variables NumBi213, NumTl, NumPb, and NumBi209, respectively. The values stored in these variables are updated as the code simulates the radioactive decay of Bismuth-213.

We also implement the half lives of Bismuth-213, Thallium, and Lead in seconds. These constants are stored in the variables tauBi213, tauTl, and tauPb. The time step of 1 second is stored in the variable h. The probabality of decays are stored in the variables probBi213Decays, probTlDecays, and probPbDecays. The probabilities that Bismuth-213 decays to Thallium and Lead are stored as constants in the variables BiToPbProb and BiToTlProb.

We use a for loop to iterate through the various times and stimulate the decay of 20000 seconds with a time step of one second. At each time step we determine whether lead, thallium, and Bismuth-213 decay in that order and we update the values of NumBi213, NumTl, NumPb, and NumBi209 and store it into arrays called Bi213Points, TlPoints, PbPoints, and Bi209Points. These arrays store the populations of these nuclei across all timesteps. When the loop is finished, we plot the nuclei amount over time.

2.4 Exercise 4

We simulate the radioactive decay of a sample of 1000 Thallium atoms. The sample size is stored in a variable called NumOfAtoms and the half life of Thallium, in seconds, is stored as a single precision floating point number in a variable called tau.

Instead of randomly determining whether a decay occurs at each time step, we generate a non-uniform distribution of times during which a decay occurs. This is much faster than calling random.random() during each iteration of a for loop. To do this, we do the following. Let $P(t)$ be the probability distribution of decay times that we want to generate from a uniform distribution. Let t denote the random variable corresponding to the decay times and let z correspond to the uniformly distributed random variable. We demand that

$$\int_0^{t(z)} P(t^{'})dt^{'} = \int_0^z dz^{'}$$

where $P(t) = \frac{\ln 2}{\tau} 2^{-t/\tau}$. From this we see the following:

$$z = \frac{\ln 2}{\tau}\int_0^t 2^{-t^{'}/\tau}dt^{'} = \ln 2 \int_{-t/\tau}^0 2^x dx = 1-2^{-t/\tau} \rightarrow -\frac{t}{\tau} = \log_2(1-z) = \frac{\ln(1-z)}{\ln 2} \rightarrow t = -\frac{\tau}{\ln 2}\ln(1-z)$$

The final equation is how we generate the non-uniform and random decay times from the uniformly distributes z's.

Using the above transformation method, we generate 1000 random numbers from a non-uniform distribution. These numbers corresponding to the times a Thallium nucleus decays. We use numpy.sort() to sort the times in increasing order. The i-th (i = 1,2,3,...) time value in the sorted list corresponds to the time when i nuclei have decayed into Lead. The array named ThalliumPoints will contain a list of Thallium nuclei present in the sample over time. Thallium-Points[0] is the number of Thallium nuclei when there have been no decays. ThalliumPoints[1] is the number of Thallium nuclei after the first decay, which is 999. Thallium[i] is the number of Thallium nuclei after the i-th decay and is equal to Thallium[i-1] - 1. As a result, we initialize ThalliumPoints to contain elements of value 1000 and we update the values stored in it accordingly through a for loop. Finally, we generate a plot showing the population of Lead and Thallium nuclei in our sample.

3 Results

3.1 Exercise 1

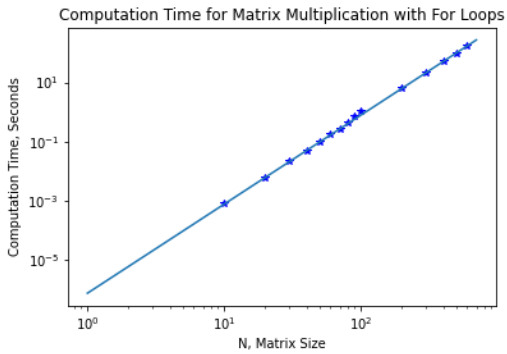
Here are the estimated values of the derivatives as the deviations vary. The methods subsection devoted to the first exercise explains these observations.

```
The approximate derivative of f(x) at x = 1 is 1.01
When delta is 1e-04, the approximated derivative at x = 1 is 1.0001
When delta is 1e-06, the approximated derivative at x = 1 is 1.000001
When delta is 1e-08, the approximated derivative at x = 1 is 1.0
When delta is 1e-10, the approximated derivative at x = 1 is 1.0000001
When delta is 1e-12, the approximated derivative at x = 1 is 1.0000889
When delta is 1e-14, the approximated derivative at x = 1 is 0.99920076
```

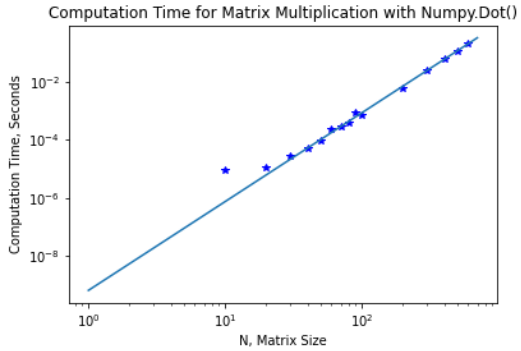
How derivatives vary with different deviation values.

3.2 Exercise 2

Below is a log-log plot of the execution times versus the size of the matrices along with a power law fit when a nested for loop is used, followed by a log-log plot when the numpy.dot() function was used. **The for loop formulation of the algorithm had longer execution times.**

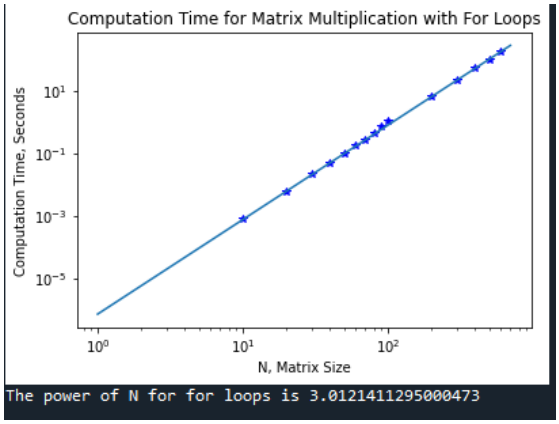


Log-Log plot of execution times for nested for loops

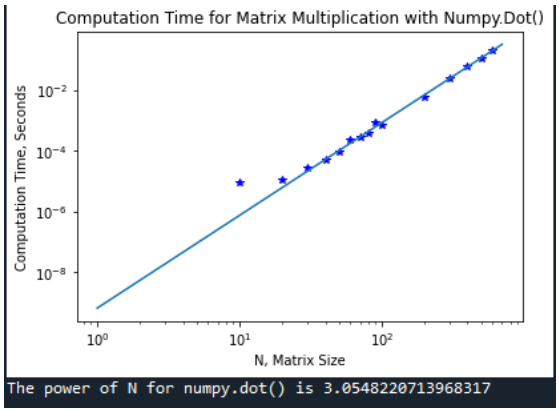


Log-Log plot of execution times for numpy.dot()

Below are the powers of N determined from the fitted models. **In both cases, the computation time scales as N^3.**



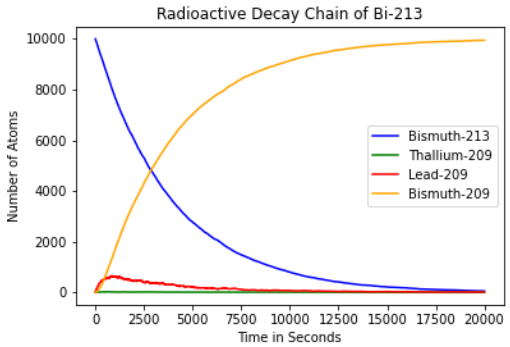
The power of N for the for loop formulation



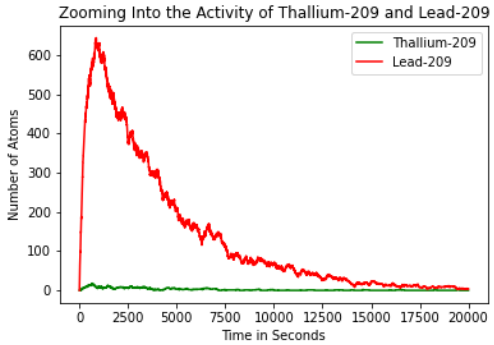
The power of N for the numpy.dot() formulation

3.3 Exercise 3

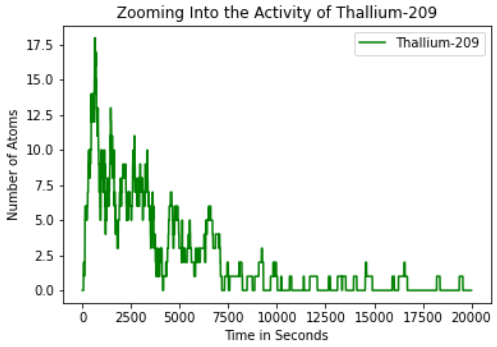
Below are several plots that depict how the nuclei population changes in the radioactive decay chain of Bismuth-213.



Plot for all four nuclei species.



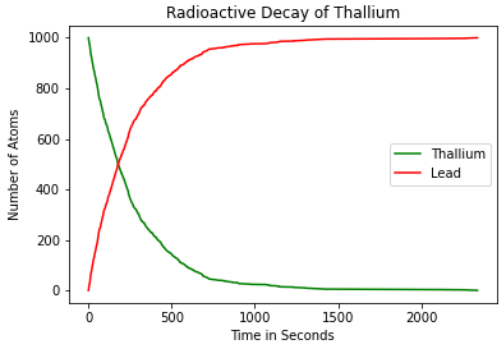
Plot that zooms into the thallium and lead species.



Plot that focuses on the thallium population.

3.4 Exercise 4

Here is the plot of the two nuclei species over time.



Population of thallium and lead. I included the lead population, which was not mandatory in the problem statement, in order to see whether the two curves' intersection matched the intersection shown in figure 10.2. The intersections match.