Problem Set 3

By Hayden Orth; GitHub: haydenorth

September 26, 2023

Abstract

This article contains my solutions to Problem Set 3 for the graduate Computational Physics course. Problem Set 3 investigates topics such as numerical errors, differentiation, array arithmetic, computational complexity, random numbers, probability distributions, and radioactive decay.

1 Problem 1: Calculating Derivatives and Numerical Error

Problem 1 investigates using a computer to approximate a function's derivative using the definition of the derivative. I took the approach that the textbook outlined: I wrote a function for the function of x in question, f(x) = x(x-1), then I wrote a second function that evaluates the derivative of the previous function using the definition of the derivative. I approximate taking the limit of delta to zero by using small values of delta on the order of 10^{-2} to 10^{-14}. I used a for loop to repeat the derivative calculation for several decreasing values of delta. The value of the derivative calculated by the computer is a little different than the value calculated analytically. This is due to the limit on delta being approximated by just using a small value (close to zero) for delta. Also, as the value of delta is decreased, the accuracy of the calculation gets better but then gets less accurate again. This is because if delta gets too small, it will lose its precision in python's floating point arithmetic. Therefore, there is an optimal value of delta that is small enough to provide a good approximation of zero, but not too small that the computer throws its precision away as negligible. This makes sense since, as often with computers, the method works well but does not work perfectly due to numerical approximation errors. Ah, the realities of computers. We still love them. Perfection is an impossibility anyway.

2 Problem 2: Matrix Multiplication and Computational Complexity

In problem 2, I study computational complexity and computation time through the lens of matrix multiplication. I use two different methods of matrix multiplication: an explicit method using three for loops to iterate through the matrices and NumPy's build-in dot() method. In order to gather data on the computational complexity, I repeat a calculation of the computation time using Python's time module. I store the time before the calculation, run the matrix multiplication of two matrices with arbitrary (nonzero) elements, store the time after, and then subtract the two times to find the time elapsed. The calculation is repeated using a for loop iterating through increasing matrix sizes. I then plot the computation time for the for loop method and the dot() method as a function of the NxN matrix size. The computation time for the explicit for loop method rises proportionally to N^3 as expected. The dot() method runs much faster. So much faster, in fact, it is difficult to make out an increase in computation time when plotted on the same axes as the explicit function. I found this to be surprising, but figured that my matrices weren't large enough to see a big difference in the dot() computation time. However, the for loop method was already taking times on the order of a minute to perform multiplication on these matrices. The dot() method is doing something much more efficient than for loops. I'm not sure what it does but I'm sure that whoever came up with what it's doing was smart and clever.

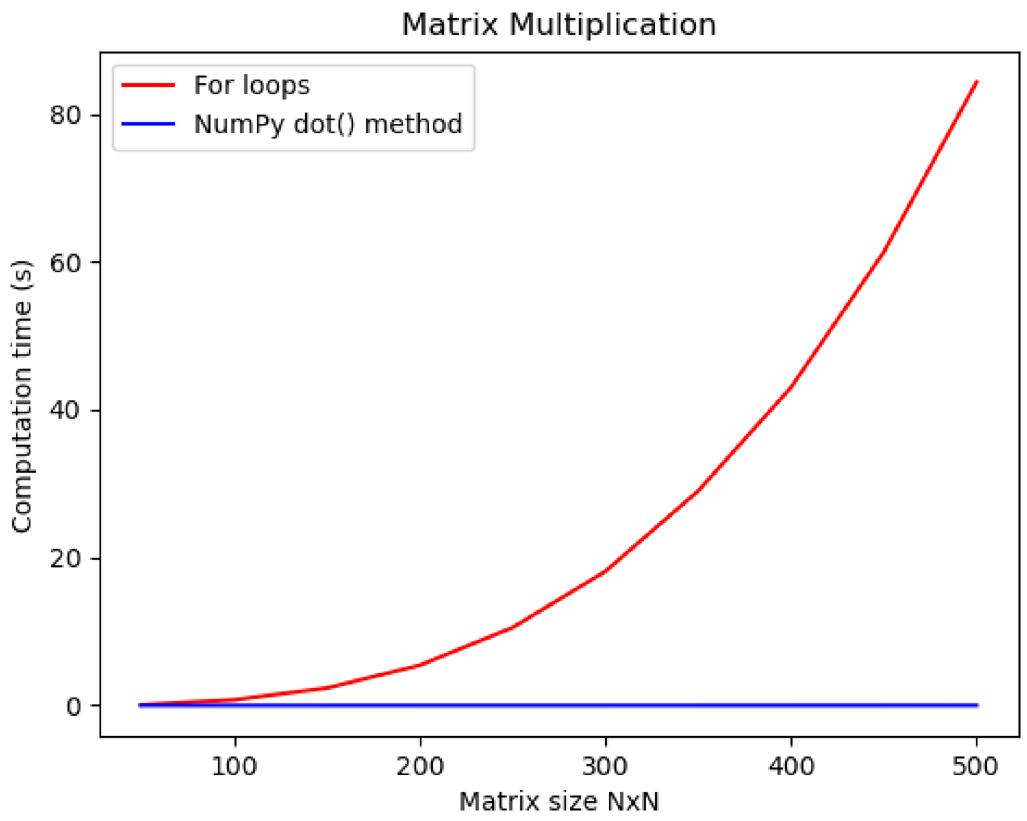

Figure 1: Plot of computation times as a function of matrix size for the explicit for loop method and the NumPy dot() method.

3 Problem 3: Radioactive Decay Chain

Problem 3 is an investigation into a chain of radioactive decay using random numbers. I used the method described in Example 10.1 in the textbook applied to a chain of isotope decays. Using the half-life of an isotope, one can find the probability of an atom of that isotope to decay in a time step of one second. Once I calculated these probabilities for all of the isotopes involved, I use a for loop to iterate time steps. At each time step, I decide whether each atom of each type will decay using a random number from a uniform probability distribution and the probability that the isotope will decay. For each atom, I compare the random number generated between 0 and 1 to the probability of decay. If the random number is less than the probability of a decay, then I say that that atom will decay. Once I calculate how many of each atom will decay for each time step, I adjust the total number of each atom accordingly and proceed to the next time step. I needed to work from the bottom up through the chain (i.e., PB decay first, then Tl, then Bi) in order to prevent an atom from decaying twice within one time step of my simulation. An additional added complication is that Bi atoms can either decay into Pb or Tl. Thus for each Bi atom decayed, I calculated a second random number to be compared to the

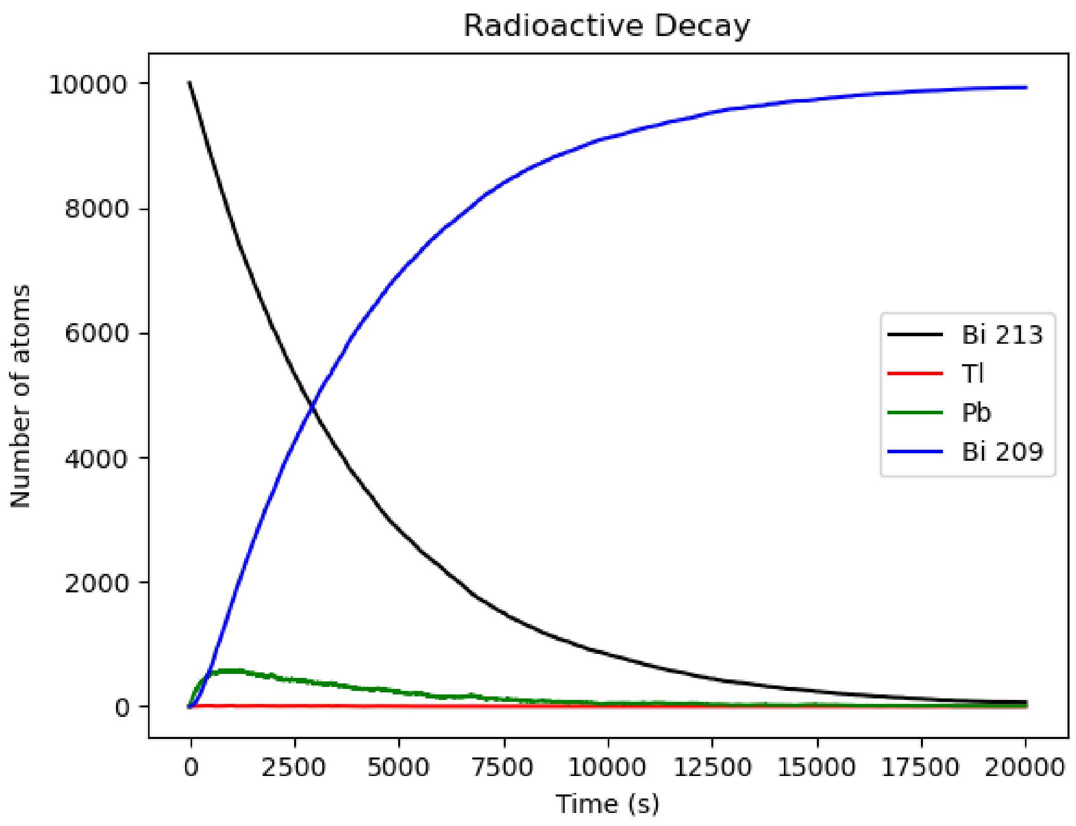

Figure 2: Plot of the number of isotopes involved in a decay chain of 10000 Bi 213 atoms.

probability of a Pb decay. If the random number did not provide a Pb decay, then the atom was labeled as decaying to Tl. As I proceeded through time steps in the for loop, I appended the total number of each atom to a respective list of number values, then plotted these values as a function of the time steps. This method uses a for loop to iterate through time which may be computationally taxing. I'm not sure of a more computationally efficient way to complete this simulation. Nevertheless, this method provided data that matches the expectation for this decay chain.

4 Problem 4: Radioactive Decay with Non-uniform Probability Density

In Problem 4, I re-investigate the radioactive decay simulation with a new method. This time, instead of drawing random numbers from a uniform probability distribution, I pull numbers from a non-uniform distribution acquired through the transformation method. The transformation method transforms a uniform distribution to a desired non-uniform distribution, depending on the desired distribution. In this case, using the half-life of Tl 208 to calculate the probability of a Tl 208 atom decaying in a single-second time step, I transformed the uniform probability distribution (providing a probability that an atom decays) to a function that provides the times at which an atom will decay (randomly, according to the newly transformed non-uniform distribution). I make an array of these 'time of decay' values, one for each of 1000 atoms in my simulation, and sort them from least to greatest. Then, for each time step, I count the number of atoms with decay times greater than this time step and append it to a list of values corresponding to the number of Tl atoms remaining. I iterate through enough time steps to cover the whole decay using a for loop. Finally, I plot the data for the number of remaining Tl atoms as a function of the time step. This is a method that is computationally much less strenuous and takes much fewer lines of code to run than the method investigated in problem 3. It also provided data that has the proper shape of a decreasing exponential. A plot of the data is included in Figure 3 on the next page. However, I don't believe this method would work for a decay chain like the one in problem 3.

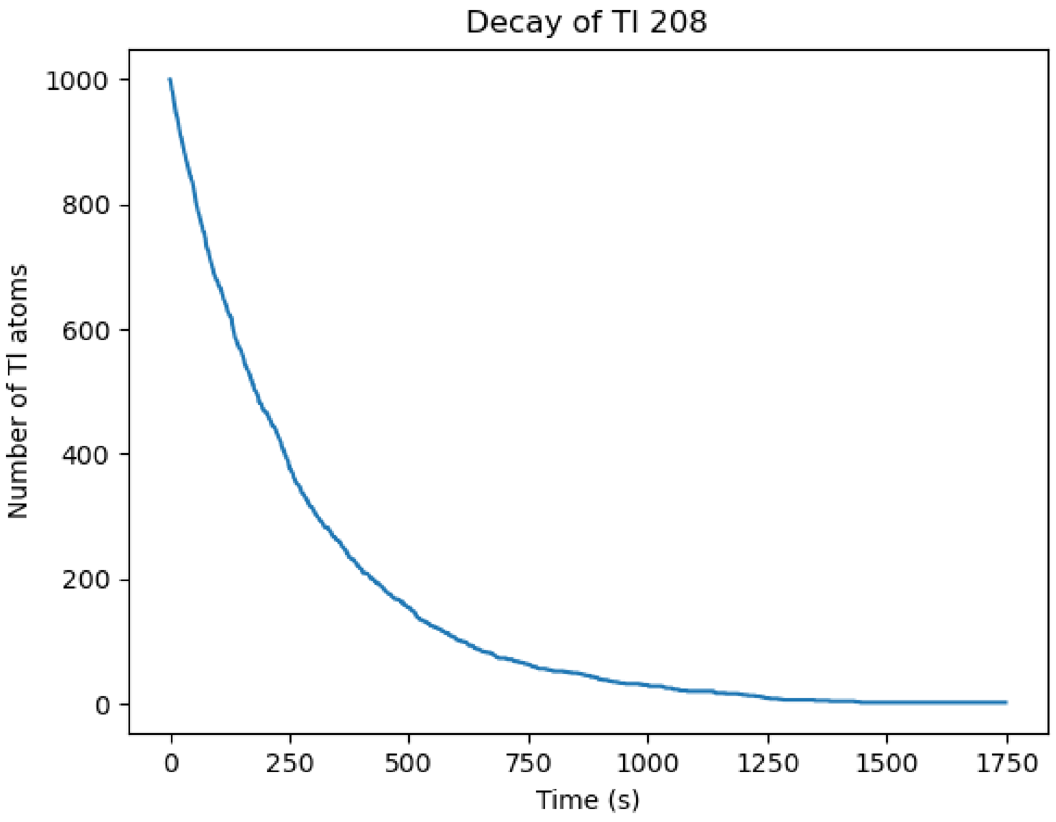

Figure 3: Plot of the number of Tl atoms remaining as the isotope decays.