
Problem 1. *Exercise 4.3 of Newman: Calculating derivatives*

Solution.

A) For the first part of the problem, I defined a function `f` that takes a value x and returns $x * (x - 1)$. This relation is given by:

$$f(x) = x(x-1)$$

This is the function whose derivative will be calculated. I then defined a function `diff_f` that accepts the parameters: `f, x, and dx` where `f` is a function, `x` is the point at which the derivative of the function will be calculated and `dx` is the δ.

This function then returns the value of the derivative of f at x using the definition of the derivative:

$$\frac{df}{dx} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}$$

I calculated the derivative of our function `f` setting $x = 1$ and $\delta = 10^{-2}$ which gave us a value of 1.010000000000001.

Finding the same derivative analytically, we get:

$$\frac{df(x)}{dx} = 2x - 1$$
$$\frac{df(1)}{dx} = (2 * 1) - 1 = 1$$

The true value of the derivative differs from our answer by 0.010000000000000897 i.e. an error in the factor of 10^{-2}. The two do not agree completely because the numerical approximation is just an approximation that gets better as we approach smaller numbers for δ. This numerical method should approach the exact derivative as $\delta \to 0$. However, as evident in part B, we encounter problems when δ is too small.

B) For the second part of the problem, I created a list with δ values ranging from 10^{-4} to 10^{-14} and repeated the calculation for each value. The results from the calculations are given in Table 1.

We can see that the result gets closer to the true derivative as δ gets smaller, but only up to a certain point ($\delta = 10^{-8}$) and then it starts deviating from the true value again. This is because as δ starts getting too small, numerical errors rise and start affecting our result. The loss of precision from subtractive cancellation errors arise since we are subtracting two

δ	difference from true $f^{'}$
10^{-04}	$9.999999988985486 * 10^{-5}$
10^{-06}	$9.99917733279787 * 10^{-7}$
10^{-08}	$3.922528746258536 * 10^{-9}$
10^{-10}	$8.284037100736441 * 10^{-8}$
10^{-12}	$8.890058334132256 * 10^{-5}$
10^{-14}	$-7.992778373491216 * 10^{-4}$

Table 1: Table of differences in numerically calculated derivatives

almost equal values in $f(x+\delta)-f(x)$ as δ gets smaller. This combined with round-off errors and limited precision leads to bigger deviations from the true derivative.

```
(prithivirana@dynamic-oit-visitornet101-10-24-111-28 ps-3 % python3 problem1.py
dx = 0.01 ---> df/dx= 1.010000000000001
               Difference from true df/dx =  0.010000000000000897

dx = 0.0001 ---> df/dx= 1.0000999999998899
               Difference from true df/dx =  9.999999988985486e-05

dx = 1e-06 ---> df/dx= 1.0000009999177333
               Difference from true df/dx =  9.99917733279787e-07

dx = 1e-08 ---> df/dx= 1.0000000039225287
               Difference from true df/dx =  3.922528746258536e-09

dx = 1e-10 ---> df/dx= 1.000000082840371
               Difference from true df/dx =  8.284037100736441e-08

dx = 1e-12 ---> df/dx= 1.0000889005833413
               Difference from true df/dx =  8.890058334132256e-05

dx = 1e-14 ---> df/dx= 0.9992007221626509
               Difference from true df/dx =  -0.0007992778373491216
```

Figure 1: Output of Problem 1

Problem 2. *Read Example 4.3 in Newman. Using successively larger matrices (10×10, 30×30, etc.) find empirically and plot how the matrix multiplication computation rises with matrix size. Does it rise as N^3 as predicted? Use both an explicit function (i.e. the one in the example) and use the dot() method. How do they differ?*

Solution.

Problem 2 tasks us with finding the complexities of matrix multiplications using a naive method and NumPy's `.dot()` method. I first made a function using the code in Example 4.3 in Newman's *Computational Physics*. My function also has a `count` variable that keeps track of the number of operations performed. The line: $C[i,j] += A[i,k] * B[k,j]$ has multiplication, addition, and assignment operations. So, `count` gets updated by 3 every time the innermost loop gets executed.

There is no direct way of counting the number of operations performed by NumPy's `.dot()` under the hood, and I couldn't find documentation detailing the operations online. So, I used the `time()` function to record the time taken for `.dot()` when performed on 1x1 matrices. For each bigger matrix, I divided the computation time for the bigger `np.dot()` matrix multiplication by the time taken for the 1x1 matrix to get an estimate of the number

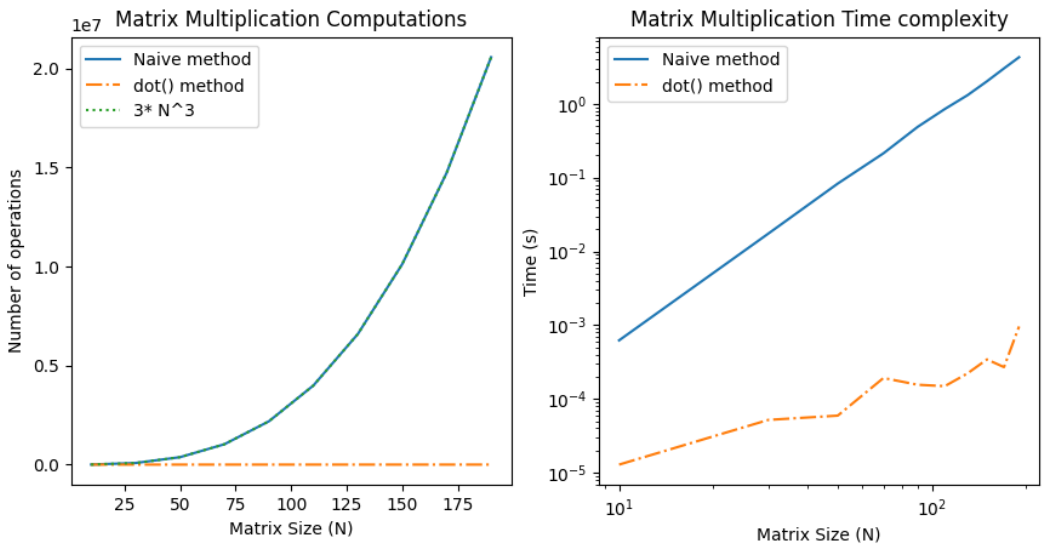

Figure 2: Matrix Multiplication Complexities

of operations. However, this is not an accurate method since the computation time varies due to computer hardware and CPU load.

The measured number of operations (and estimated operations for `.dot()`) and the measured computation times for the two methods are plotted in Figure 2. The left side of the figure compares the number of operations for the naive and `.dot()` methods. It can be seen that the number of operations for the naive method rises by N^3 which confirms our predictions. The three nested loops run $C[i,j] += A[i,k] * B[k,j]$ a total of N^3 times, and since there are 3 operations, a total number of $3N^3$ operations are performed. The curve for the `.dot()` method is almost a horizontally straight line when compared to the N^3 curve in the broad scale. The figure on the right is plotted in log scale to get a better comparison between the two methods. Information available on NumPy's `.dot()` method suggests that it is highly optimized and uses libraries implemented in C to perform its calculation.

The graph on the right compares the time taken for each matrix size using both methods. The naive method shows a linear behavior in the log scale which suggests that its time complexity is polynomial, more specifically a complexity of $\texttt{O}(N^3)$. The `dot()` method shows an almost linear behavior but its slope is smaller than the slope for the naive method, so we know that it is a polynomial of a smaller degree.

Problem 3. *Exercise 10.2 in Newman: Radioactive Decay Chain*

Solution.

This problem looks at a simulation of a simple radioactive decay chain. It tasks us with tracking the decay and change in the number of Bismuth, Thallium, and Lead isotopes in the decay chain over time.

My program achieves this by first calculating the probability of decay for each isotope using

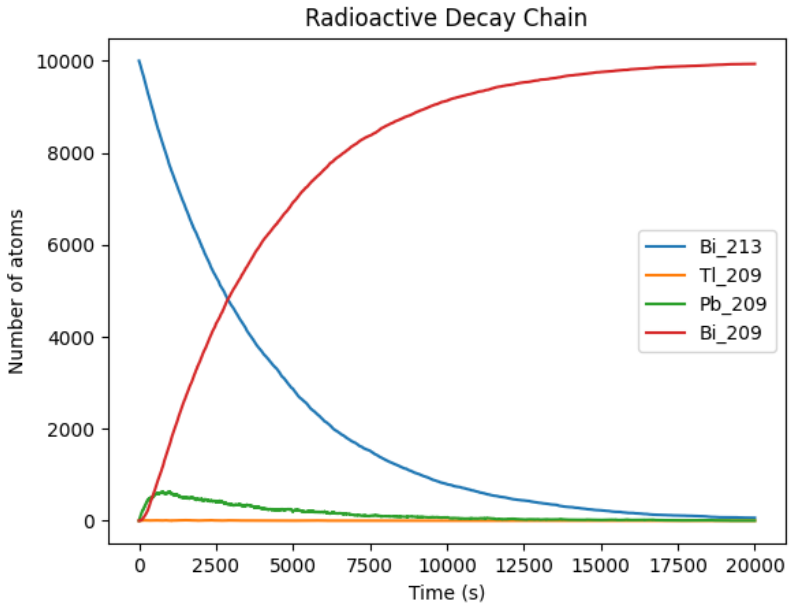

Figure 3: Number of atoms of Isotopes as a function of time

the formula

$$P(t) = 1 - 2^{-t/\tau}$$

where τ is the half-life of the isotope and t is the time. Setting t = 1 gives us the probability of decay for each second. I declared counters to keep track of the number of atoms of each isotope. Since we start off with 10,000 ^{213}Bi atoms, `NBiI` is initialized to 10,000 and all other counters start off at 0. I then have a main loop that iterates 20,000 times (simulating 20,000 seconds since we are looking at decay probabilities for 1-second time intervals) and counts the number of atoms that have decayed for each isotope.

For counting the number of atoms decayed in each iteration, we start off from the bottom of the chain with ^{209}Pb. There is a loop inside the main loop that invokes the `random()` function and checks if it is less than the probability of decay of the ^{209}Pb isotope. This checks whether one atom of ^{209}Pb has decayed or not. If it has decayed, 1 is subtracted from the ^{209}Pb counter and added to the counter for the stable Bi. This loop iterates until all ^{209}Pb atoms currently in the system are checked.

There is a similar loop for ^{209}Tl that counts all the ^{209}Tl atoms that decayed into ^{209}Pb. We then reach the starting isotope of the decay chain, ^{213}Bi. This loop is a little different because ^{213}Bi can either decay into ^{209}Tl or ^{209}Pb (with different probabilities of decaying into each). The loop first checks if the atom decays in a similar way to the other loops. Then it invokes the `random()` function again and checks which isotope it decays into and updates the counters accordingly. Working from the bottom up in the chain helps avoid making the same atom decay twice in a single step.

Problem 4. *Exercise 10.4 in Newman: Radioactive Decay again*

Solution. This problem tasks us with simulating the radioactive decay with a faster method. For this, we first need to generate non-uniform random numbers from the transformation of the equation

$$P(t)dt = 2^{-t/\tau}\frac{ln2}{\tau}dt$$

where:

`P(t)` is the probability of decay of an isotope at time t

τ is the half-life of the isotope.

This helps us generate non-uniform random numbers representing decay times for atoms of an isotope using the equation:

$$x = -\frac{1}{\mu}ln(1-z)$$

where:

`x` is the randomly generated decay time

$\mu = \frac{ln2}{\tau}$

`z` is the uniform random distribution.

For `z`, we generate an array of 1000 uniformly distributed random numbers (since we start off with 1000 ^{209}Tl atoms) using NumPy's `random.rand()` function. The equation above transforms this into decay times for 1000 Tl atoms. Similar to the last problem, a loop runs 1000 times to check the state of the system each second. The loop counts and updates an array with how many atoms are left to decay at that particular time. this data is plotted in Figure 4.

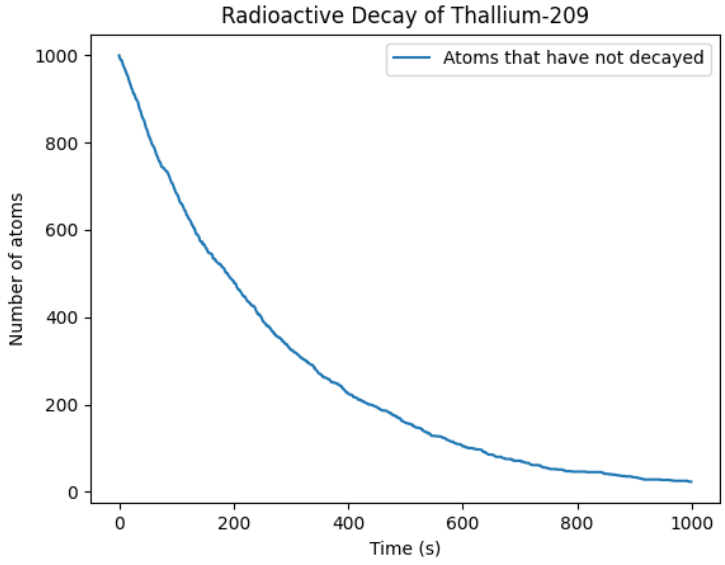

Figure 4: Number of atoms of ^{209}Tl that have not decayed at time t