ps-2

Greta Goldberg

September 23, 2023

1 Github Link

https://github.com/gretagoldberg/phys-ua210/tree/main

2 Question 1

For this question I defined a function to be evaluated and then also defined a function that uses the limit definition of the derivative that takes delta in as an argument. I then calculated the true derivative by hand and defined a function to take x in as an input. Upon decreasing delta, it seems as though the limit derivative approaches the correct value. However, it eventually diverges from the true solution.

```
import numpy as np

#defining function to evaluate
def function(x):
    return x*(x-1)

#calculating the derivative using the limit definition
def derivative_of_function(delta):
    return (function(1+delta)-function(1))/delta

#derivative for delta=10^-2
print(derivative_of_function(10**(-2)))

#actual derivative
def true_value(x):
    return 2*x-1

#actual value
print(true_value(1))

#the two answers do not agree perfectly
```

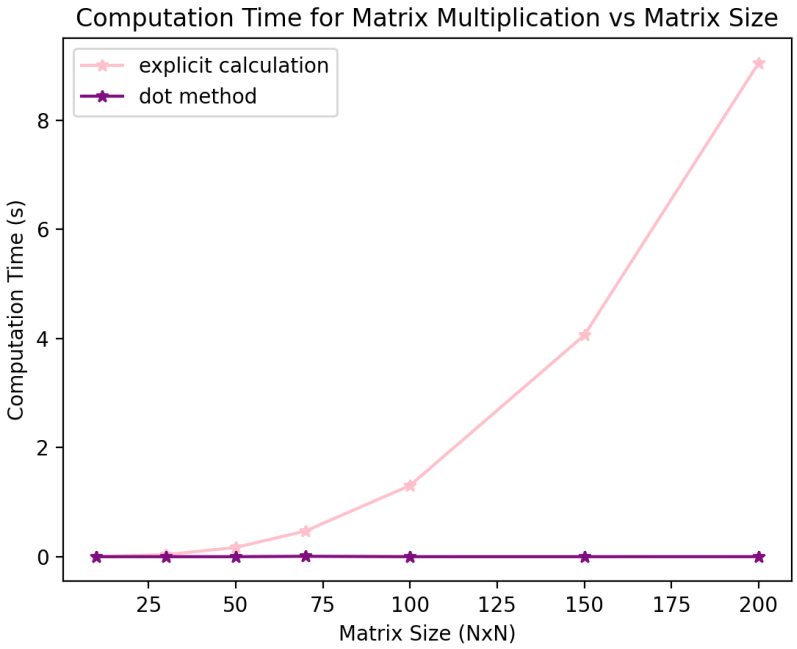
```
#because delta is quite large in comparison to zero

deltas = [10**(-4),10**(-6),10**(-8),10**(-12),10**(-14)]

for i in deltas:
    print(derivative_of_function(i))

#the value seems to approach the correct value
#and then gets worse because of how computers store and deal with floats
```

3 Question 2

For this question I first defined a function to compute matrix multiplication as the problem shows. I then initialized a list of matrix dimensions as well as two empty list that I would use to store the computation times. I then iterate through the different matrix dimensions, creating matrices of the correct size with random numbers. I begin timing each method and then stop timing afterwards. In this way I can compare the two methods by plotting the matrix dimensions in relation to the computational time elapsed.

```
import numpy as np
import time
import matplotlib.pyplot as plt

#defining function to do matrix multiplication
def matrix_multiplication(A, B):
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape
    C = np.zeros((rows_A, cols_B))

    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                C[i][j] += A[i][k] * B[k][j]

    return C

#creating list to store dimensions of matrices
N = [10, 30, 50, 70, 100, 150, 200]

# Initialize lists to store computation times for both methods
times_explicit = []
times_dot = []

for i in N:
```

```
    A = np.random.rand(i, i)
    B = np.random.rand(i, i)

    #explicit method
    start_time = time.time()
    result_explicit = matrix_multiplication(A, B)
    end_time = time.time()
    elapsed_time_explicit = end_time - start_time
    times_explicit.append(elapsed_time_explicit)

    #dot method
    start_time = time.time()
    result_dot = np.dot(A, B)
    end_time = time.time()
    elapsed_time_dot = end_time - start_time
    times_dot.append(elapsed_time_dot)

#plots
plt.plot(N, times_explicit, marker='*', label='explicit calculation',c="pink")
plt.plot(N, times_dot, marker='*', label='dot method',c="purple")
plt.xlabel('Matrix Size (NxN)')
plt.ylabel('Computation Time (s)')
plt.title('Computation Time for Matrix Multiplication vs Matrix Size')
plt.legend()
plt.show()

#this does approximately exhibit N^3 behavior as predicted
```

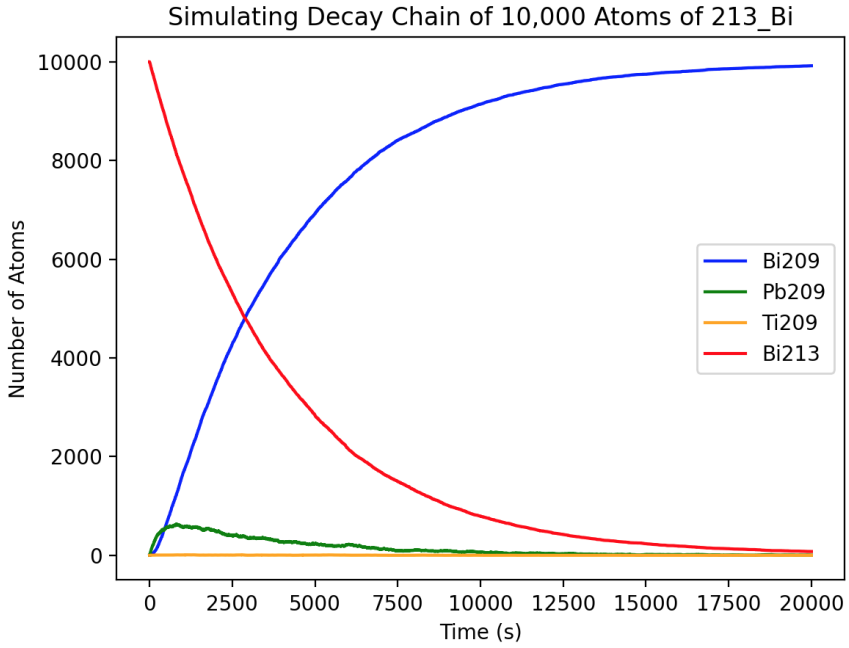


4 Question 3

This code begins by initializing many variables including number of molecules of each isotope, the time step, the probability of decay for three isotopes and lists to store the values for each isotope over time. We then iterate through time going up the chain as the problem suggests. This way we do not double count any isotopes. The logic for Bi 213 is slightly different because instead of using the equation with half life, I used the probabilities for decay. I then plot the number of atoms over time.

```
from random import random
from numpy import arange
import matplotlib.pyplot as plt

#assigning variables
h = 1
NBi209 = 0
NPb209 = 0
NTi209 = 0
NBi213 = 10000

#assigning probabilities
```

```
pPb = 1 - 2**(-h/3.3/60)
pTi = 1 - 2**(-h/2.2/60)
pBi = 1 - 2**(-h/46/60)

#initializing lists to store data over time
Bi209_list = []
Pb209_list = []
Ti209_lsit = []
Bi213_list = []

t = arange(0,20000,h)
for ti in t:
        Bi209_list.append(NBi209)
        Pb209_list.append(NPb209)
        Ti209_lsit.append(NTi209)
        Bi213_list.append(NBi213)

        for i in range(NPb209):
                if random()<pPb:
                        NPb209-=1
                        NBi209+=1

        for i in range(NTi209):
                if random()<pTi:
                        NTi209-=1
                        NPb209+=1

        for i in range(NBi213):
                if random()<pBi:
                        NBi213 -=1
                        if random()<0.9791:
                                NPb209+=1
                        else:
                                NTi209+=1

plt.plot(t,Bi209_list,label='Bi209',c="blue")
plt.plot(t,Pb209_list,label='Pb209',c="green")
plt.plot(t,Ti209_lsit,label='Ti209',c = "orange")
plt.plot(t,Bi213_list,label='Bi213',c="red")
plt.legend()
plt.title("Simulating Decay Chain of 10,000 Atoms of 213_Bi")
plt.xlabel('Time (s)')
plt.ylabel('Number of Atoms')
plt.show()
```



5 Question 4

For this question I used 1000 randomly generated integers from the distribution $x = -\frac{1}{\mu}\ln(1-z)$. Using the sort method, I can order the times from low to high. I can then plot the number of atoms left over time.

```
from numpy.random import random
import numpy as np
import matplotlib.pyplot as plt

#initializing variables
N = 1000
tau = 3.053*60
mu = np.log(2)/tau

z = random(N)

time_dec = -1/mu*np.log(1-z)
time_dec = np.sort(time_dec)
decayed = np.arange(1,N+1)

#updating number of atoms
```

```
atoms_left = N -decayed

#plot statements
plt.plot(time_dec,atoms_left)
plt.xlabel("Time (s)")
plt.ylabel("Number of Ti Atoms")
plt.title("Simulation of Decay of Ti Atoms over Time")
plt.show()
```