# PHYS-UA 210 Computational Physics
# Problem Set 03

Sandhya Sharma

September 23, 2023

## Question 1

### Exercise 4.2: Calculating Derivatives

The following code calculates the derivative of the function $f(x) = x(x-1)$ with the value of $\delta$ approaching 0. The values $\delta$ used are: $10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, 10^{-14}$.

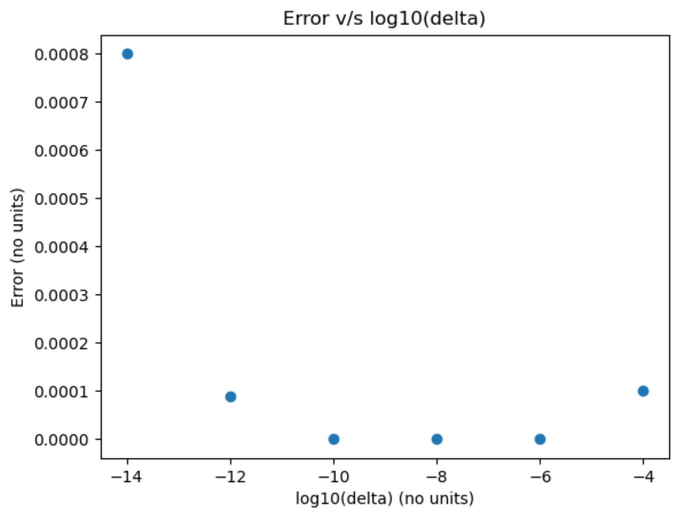
The error from the true value of the derivative calculated analytically is plotted against the value of $\delta$.

```
#importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import math

#defining the function
def func(x):
    return x*(x-1)

#function to calculate derivative computationally
def derivative(x,delta):
    f_x = (func(x+delta) - func(x))/delta
    return f_x

#setting the values for delta and x
delta = 10**(-2)
x = 1

#calculating the derivative computationally and analytically
func_x = derivative(x, delta)
func_x_true = (2*x) - 1
```

```
print()
print("Function: f(x) = x(x-1)")
print("At x = " + str(x) + "," + " Delta = " + str(delta))
print("df/dx = ", func_x)
print("True value of derivative (analytically) = ", func_x_true)
print("Fractional difference = ", (func_x - func_x_true)/func_x_true)
print()

#caluclating derivative with different values for delta
delta_array = np.array([10**-4, 10**-6, 10**-8, 10**-10, 10**-12, 10**-14])
func_x_array = np.empty(shape = (0,))

for i in range(delta_array.size):
    func_x = derivative(x, delta_array[i])
    func_x_array = np.append(func_x_array, func_x)
    print("Delta = " + str(delta_array[i]) + ", df/dx = " + str(func_x_array[i]))

error = func_x_array - func_x_true
error = np.abs(error)

print()

#plotting error vs log10(delta)
plt.scatter(np.log10(delta_array), error)
plt.xlabel('log10(delta) (no units)')
plt.ylabel('Error (no units)')
plt.title('Error v/s log10(delta)')
plt.savefig('cp_ps3_q1.png')
plt.show()
```

**Output:**

```
Function: f(x) = x(x-1)
At x = 1, Delta = 0.01
df/dx =  1.010000000000001
True value of derivative (analytically) =  1
Fractional difference =  0.010000000000000897

Delta = 0.0001, df/dx = 1.0000999999998899
Delta = 1e-06, df/dx = 1.0000009999177333
Delta = 1e-08, df/dx = 1.0000000039225287
Delta = 1e-10, df/dx = 1.000000082840371
Delta = 1e-12, df/dx = 1.0000889005833413
Delta = 1e-14, df/dx = 0.9992007221626509
```




Here, the error gets better as $\delta$ gets closer to zero however gets worse again as it goes below $10^{-8}$ since dividing with such extremely small values yield to truncation of digits beyond the memory capacity of a the variable type it is being stored in.

# Question 2

## Example 4.3: Matrix Multiplication

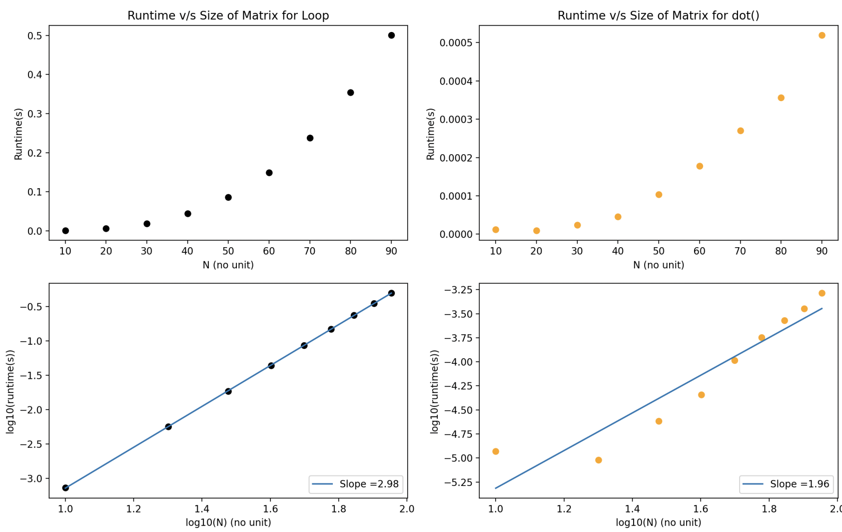
The following code performs $NxN$ matrix multiplication two ways: using nested loops and using $np.dot()$ and comparing their runtimes as N increases.

```
#importing necessary libraries
import numpy as np
import timeit
import matplotlib.pyplot as plt
import math

#function for matrix multiplication
def multiply(N, A,B):
    result = np.zeros([N,N] ,int)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                result[i,j] += A[i,k]*B[k,j]
    return result
```

```
N = np.arange(10, 100, 10)

loop_time = np.empty(shape = (0,))
dot_time = np.empty(shape = (0,))

for i in range(N.size):
    #creating matrices A and B of random integers < 20 of size N x N
    A = np.random.randint(20, size = (N[i], N[i]))
    B = np.random.randint(20, size = (N[i], N[i]))

    #using nested loops for multiplication and measuring runtime
    start1 = timeit.default_timer()
    C = multiply(N[i],A,B)
    stop1 = timeit.default_timer()
    loop_time = np.append(loop_time, stop1-start1)

    #using np.dot() for multiplication and measuring runtime
    start2 = timeit.default_timer()
    C_without_loop = np.dot(A,B)
    stop2 = timeit.default_timer()
    dot_time = np.append(dot_time, stop2-start2)

#plotting runtimes against number of operations
fig, ((ax1, ax2) , (ax3, ax4)) = plt.subplots(2,2)
ax1.scatter(N, loop_time, color = 'black')
ax1.set(xlabel = 'N (no unit)', ylabel = ("Runtime(s)"))
ax1.set_title("Runtime v/s Size of Matrix for Loop")

ax2.scatter(N, dot_time, color = 'orange')
ax2.set(xlabel = 'N (no unit)', ylabel = ("Runtime(s)"))
ax2.set_title("Runtime v/s Size of Matrix for dot()")

#plotting log10 of runtimes against lo10 of number of operations to
#see if the complexity increases with N**3
ax3.scatter(np.log10(N), np.log10(loop_time), color = 'black')
#finding the slope of the linear fit
m1, b1 = np.polyfit(np.log10(N), np.log10(loop_time), deg=1)
ax3.set(xlabel = 'log10(N) (no unit)', ylabel = ("log10(runtime(s))"))
ax3.plot(np.log10(N), m1*np.log10(N) + b1, label = 'Slope =' + str(float(f'{m1:.2f}')))
ax3.legend(loc = "lower right")

ax4.scatter(np.log10(N), np.log10(dot_time), color = 'orange')
#finding the slope of the linear fit
m2, b2 = np.polyfit(np.log10(N), np.log10(dot_time), deg=1)
ax4.set(xlabel = 'log10(N) (no unit)', ylabel = ("log10(runtime(s))"))
```

```
ax4.plot(np.log10(N), m2*np.log10(N) + b2, label = 'Slope =' + str(float(f'{m2:.2f}')))
ax4.legend(loc = "lower right")
plt.show()
```

**Output:**




In the first row of the graph, the run-times of each of the algorithms have been plotted against $N$ and in the second row, their respective logarithms are plotted to check if the run-time increases with $N^3$. Indeed, the slope for the nested loop algorithm is very lose to 3(2.98) proving the proposed time complexity to be true. However, this is not same for the *np.dot*() method with the slope of 1.96. Clearly they differ in their run-time and therefore the algorithms by which each of the method work. To be specific, *np.dot*() calculates the dot product of two arrays and for 2-D arrays, this is the matrix multiplication.

# Question 3

## Exercise 10.2: Radioactive Decay of Bi-213

The following code performs the radioactive decay of 10,000 Bi-213 atoms into Pb-209 and Tl-209 and subsequently Bi-209 for 20,000 seconds.

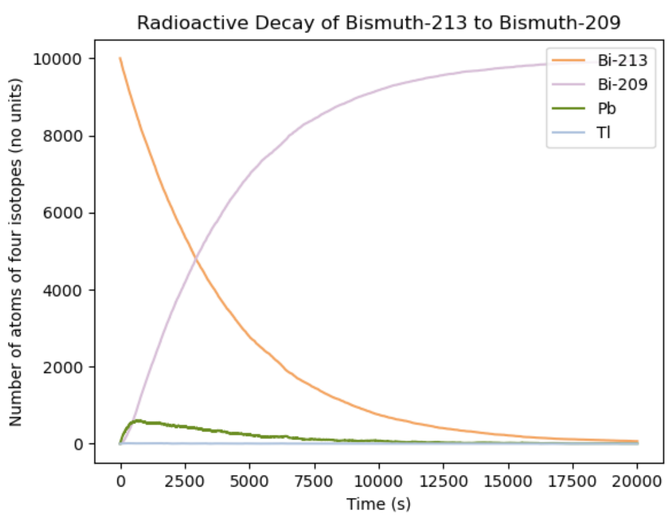
```
#importing necessary libraries
import math
```

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

#initializing variables required to simulate the radiaoactive decay
n_bi213 = 10000; n_pb = 0; n_tl = 0; n_bi209 = 0 #initial number of four isotopes
total_time = 20000 #total time of simulation
time_step = 1 #length of time for each step
tau_bi213 = 46*60 #half-life of Bi-213
tau_pb = 3.3*60 #half-life of Pb-209
tau_tl = 2.2*60  #half-life of Tl-209
prob_bi213_to_pb = 0.9791 #probability that one atom of Bi-213 decays
#into Pb-209 at any given time
prob_bi213_to_pl = 0.209 #probability that one atom of Bi-213 decays
#into Tl-209 at any given time

#initializing arrays to record the number of each isotope over time
time_array = np.arange(0,total_time, time_step)
n_bi213_array = np.empty(shape = (0,))
n_pb_array = np.empty(shape = (0,))
n_tl_array = np.empty(shape = (0,))
n_bi209_array = np.empty(shape = (0,))

#function to return the probability of decay of any one atom (determined by the value
#of tau) in the span of one second
def probability_of_decay(tau):
    p = 1 - 2**(-1/tau)
    return p

#performing every step of radioactive decay for four isotopes for each unit of tine
for i in range(total_time):
    n_bi213_array = np.append(n_bi213_array, n_bi213)
    n_pb_array = np.append(n_pb_array, n_pb)
    n_tl_array = np.append(n_tl_array, n_tl)
    n_bi209_array = np.append(n_bi209_array, n_bi209)

    for j in range(n_bi213):
        if random.random() < probability_of_decay(tau_bi213):
            n_bi213 -= 1
            if random.random() < prob_bi213_to_pb:
                n_pb += 1
            else:
                n_tl += 1

    for j in range(n_tl):
        if random.random() < probability_of_decay(tau_tl):
```

```
            n_tl -= 1
            n_pb += 1

    for j in range(n_pb):
        if random.random() < probability_of_decay(tau_pb):
            n_pb -= 1
            n_bi209 += 1

#recording the initial and final number of atoms for each isotope
print('Initial Number of Atoms (at t = 0 s): ')
print('Bi-213 = ', n_bi213_array[0])
print('Pb-209 = ', n_pb_array[0])
print('Tl-209 = ', n_tl_array[0])
print('Bi-209 = ',  n_bi209_array[0])
print()

print('Final number of atoms (at t = 20000 s): ')
print('Bi-213 = ', n_bi213)
print('Pb-209 = ', n_pb)
print('Tl-209 = ', n_tl)
print('Bi-209 = ', n_bi209)
print()

#plotting the number of atoms of each isotope against time
plt.plot(time_array, n_bi213_array, color = 'sandybrown', label = 'Bi-213')
plt.plot(time_array, n_bi209_array, color = 'thistle', label = 'Bi-209')
plt.plot(time_array, n_pb_array, color = 'olivedrab', label = 'Pb')
plt.plot(time_array, n_tl_array, color = 'lightsteelblue', label = 'Tl')
plt.legend(loc = 'upper right')
plt.title('Radioactive Decay of Bismuth-213 to Bismuth-209')
plt.xlabel('Time (s)')
plt.ylabel('Number of atoms of Bi-213(no units)')
plt.savefig('cp_ps3_q3.png')
plt.show()
```

**Output:**

```
Initial Number of Atoms (at t = 0 s):
Bi-213 =  10000.0
Pb-209 =  0.0
Tl-209 =  0.0
Bi-209 =  0.0

Final number of atoms (at t = 20000 s):
Bi-213 =  66
Pb-209 =  6
Tl-209 =  0
Bi-209 =  9928
```




# Question 4

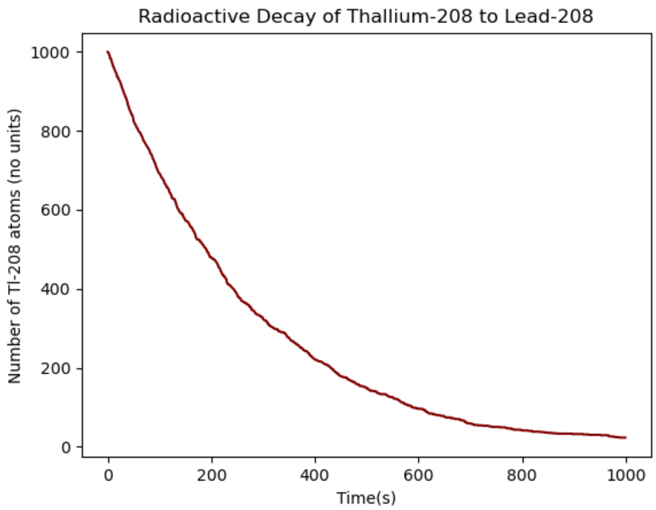
## Exercise 10.4: Radioactive Decay of Tl-208 to Pb-208

The following code performs the radioactive decay of 1000 atoms of Tl-208 to Pb-208 using non-uniform distribution.

```python
#importing necessary libraries
import math
import numpy as np
import random
import matplotlib.pyplot as plt

#initializing values required for radioactive decay
```

```
n_tl208 = 1000
tau_tl209 = 3.053*60
mu = math.log(2)/tau_tl209
total_time = 1000
time_step = 1
time_array = np.arange(0,total_time, time_step)

#initializing an array to store the random numbers generated in the next loop
t_array = np.empty(shape = (0,))

#generating random numbers from a non-uniform distribution that represents the time
#at which a given atom will decay
for i in range(n_tl208):
    t = -(1/mu)*math.log(1-random.random())
    t_array = np.append(t_array, t)

#sorting the array and finding the number of atoms that decay before a given time
t_array_sorted = np.sort(t_array)
decayed_atoms = 0
n_tl208_array = np.empty(shape = (0,))

for i in range(total_time):
    decayed_atoms = np.argmax(t_array_sorted > i)
    remaining_n_tl208 = n_tl208 - decayed_atoms
    n_tl208_array = np.append(n_tl208_array, remaining_n_tl208)

#plotting the number of atoms of Tl-208 against time
plt.plot(time_array, n_tl208_array, color = 'maroon')
plt.title('Radioactive Decay of Thallium-208 to Lead-208')
plt.xlabel('Time(s)')
plt.ylabel('Number of Tl-208 atoms (no units)')
plt.savefig('cp_ps3_q4')
plt.show()
```

**Output**:




Please find my GitHub repository through this link.