Week 7 - Accumulation Function

November 21, 2024

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import math
```

[2]:
```
#program TABLE

#This program takes in a function (fnf) that defines the rate of change of the␣
↪function at any point,
# initial and final times (t), and a set number of steps and accumulates total␣
↪change
# of that function over the time period. This can be added to any initial␣
↪condition.

def fnf(x):
    return (math.cos(x**2)) #define dy/dt

t_initial = 0
t_final = 4

numberofsteps = 2**3

deltat = (t_final-t_initial)/numberofsteps #calculate delta t for use in␣
↪calculating total change.

t = t_initial #initialize t

accumulation = 0 #accumulation always starts at zero


#print header
print("{:<15} {:<10} {:<20} {:<10}".format("starting t","DeltaY", "Accumulated␣
↪DeltaY", "ending t"))

for k in range (numberofsteps):
    deltay = fnf(t) * deltat #calculate the each change in y
    accumulation = accumulation + deltay #accumulate the total change in y
    t_end = t + deltat #ending t for this step
```

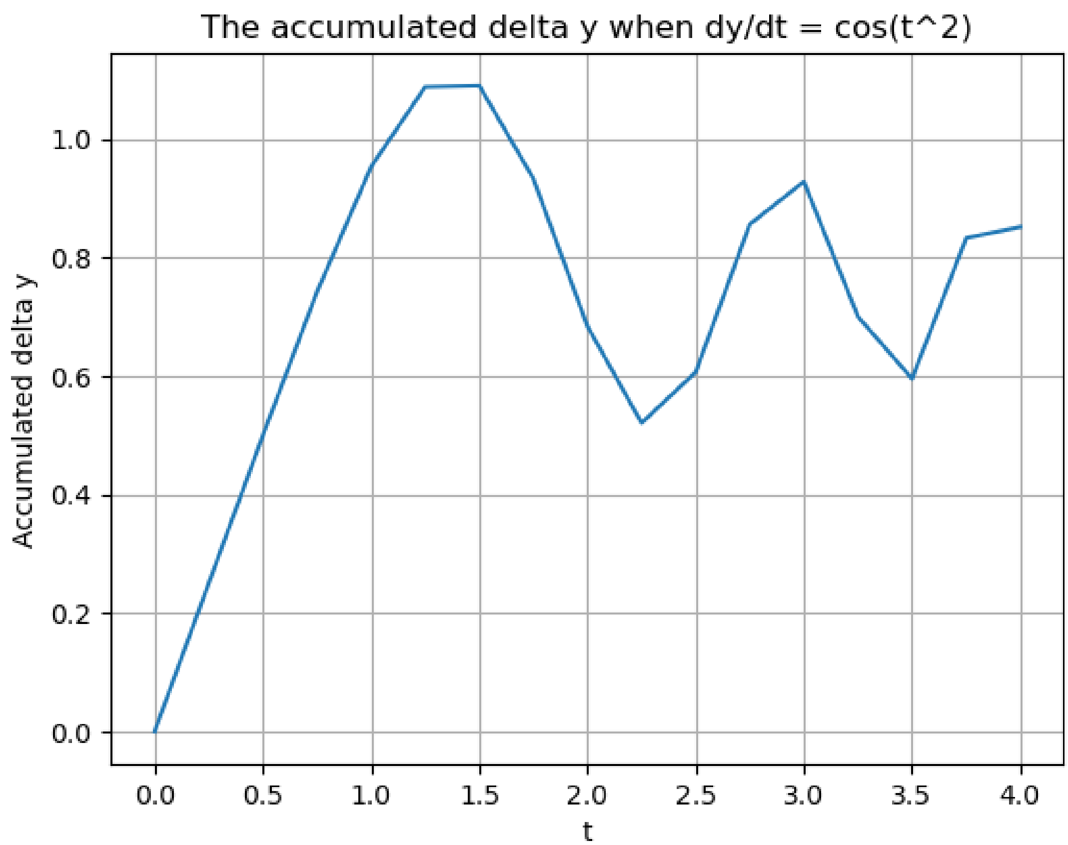
```
    print("{:<15.4f} {:<10.4f} {:<20.4f} {:<10.4f}".format(t, deltay,
↪accumulation, t_end)) #print output with format
    t = t_end # set new start t
```

```
starting t      DeltaY     Accumulated DeltaY   ending t
0.0000          0.5000     0.5000               0.5000
0.5000          0.4845     0.9845               1.0000
1.0000          0.2702     1.2546               1.5000
1.5000          -0.3141    0.9405               2.0000
2.0000          -0.3268    0.6137               2.5000
2.5000          0.4997     1.1134               3.0000
3.0000          -0.4556    0.6579               3.5000
3.5000          0.4752     1.1330               4.0000
```

I coded plot below and deliberately kept the same function, number of steps, and time to see how this plotted with fewer steps.

[3]:
```
#Program: Plot

#Plot is similar to TABLE in that it takes in a function for dy/dt, number of
↪steps, and time and calculates the accumulated change.
#Unlike TABLE, Plot outputs the results as a graph.
t_initial = 0
t_final = 4

numberofsteps = 2**4

deltat = (t_final-t_initial)/numberofsteps

t = t_initial #initialize t

accumulation = 0 #accumulation always starts at zero

t_graph = [t]
accum_graph = [accumulation]




for k in range (numberofsteps):
    deltay = fnf(t) * deltat #calculate the each change in y
    accumulation = accumulation + deltay #accumulate the total change in y
    t = t + deltat #accumulate the total t

    #append values to return
    t_graph.append(t)
    accum_graph.append(accumulation)
```

```
plt.plot(t_graph, accum_graph)


plt.xlabel('t')
plt.ylabel('Accumulated delta y')
plt.title('The accumulated delta y when dy/dt = cos(t^2)')

plt.grid(True)
plt.show()
```



Thanks to Christopher for the idea of hard coding each of the functions from the Ebola activity to make it easier to use later.

[4]:
```
def A(x):
    return  827 / ((10.5)**2 + (x - 55)**2)



def B(x):
```

```
    return 1400 / ((10)**2 + (x - 45)**2)


def C(x):
    return 1200 / ((10)**2 + (x - 45)**2)


def D(x):
    return 827 / ((10.5)**2 + (x - 44.4)**2)

def E(x):
    return 1000 / ((10.5)**2 + (x - 48)**2)


def F(x):
    return 827 / ((10.5)**2 + (x - 48)**2)
```

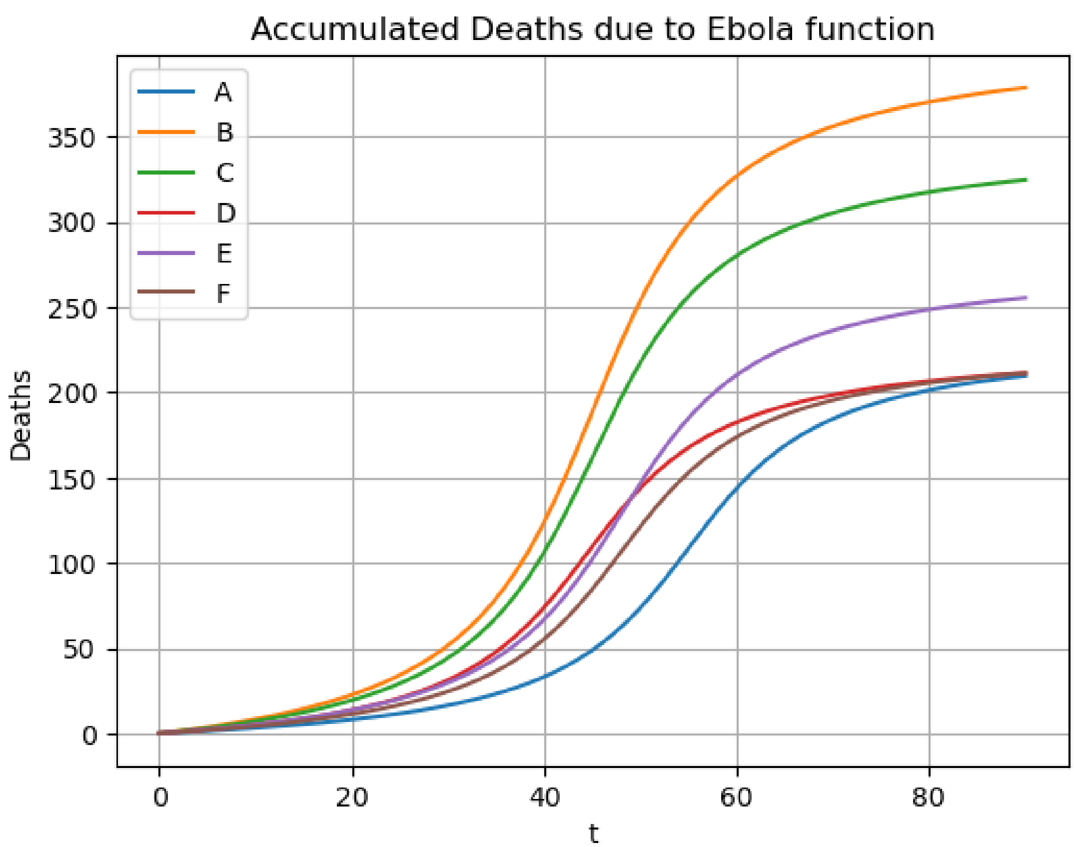
I modified the original plot to take the initial time, final time, steps, and function as inputs and to return a plot labeled with which function was used.

[5]:
```
def plot(t_initial, t_final, steps, func):

    numberofsteps = steps

    deltat = (t_final-t_initial)/numberofsteps

    t = t_initial #initialize t

    accumulation = 0 #accumulation always starts at zero

    t_graph = [t]
    accum_graph = [accumulation]




    for k in range (numberofsteps):
        deltay = func(t) * deltat #calculate the each change in y
        accumulation = accumulation + deltay #accumulate the total change in y
        t = t + deltat #accumulate the total t

        #append values to return
        t_graph.append(t)
        accum_graph.append(accumulation)

    plt.plot(t_graph, accum_graph, label= func.__name__)
```

```
    plt.xlabel('t')
    plt.ylabel('Deaths')
    plt.title('Accumulated Deaths due to Ebola function')

    plt.grid(True)
```

I then ran all of the functions from the Ebola exercise through the plot function and graphed them against each other. Originally I had printed them as individual graphs but found this a much better way to compare the different function's outputs.

[6]:
```
funcs = [A,B,C,D,E,F]
for fun in funcs:
    plot(0,90,1000,fun)
plt.legend()
plt.show()
```



[ ]:
```
```