# PHY407 - Lab 1

Q2,3 by Hayley Agler, Q1,3 by Nicholas Pavanel

September 18, 2020

## Question 1

### a)

No submission.

### b)

The Newtonian gravitational force defining the entirety (ie. spatial postion and spatial velocity) of a planet's orbit around a star is approximated by $F_g = -\frac{GM_sM_p}{r^3}(x\bar{x}y\bar{y})$ . Taking the $M_s >> M_p$ we get that $F_g = -\frac{GM_s}{r^3}(x\bar{x}+y\bar{y})$ . Noticing that for a first order system $\bar{F}(\bar{x}_i) = \frac{d\bar{x}}{dt} \approx \frac{\Delta x}{\Delta t} = \frac{\bar{x}_{i+1}-x_i}{\Delta t}$, and solving the classical central force problem produces the following equations of motion for numerical integration:

(1) $\bar{v}_{x_{i+1}} = \bar{v}_{x_i} - \frac{GM_s\bar{v}_{xi}}{r^3}$,

(2) $\bar{v}_{y_{i+1}} = \bar{v}_{y_i} - \frac{GM_s\bar{v}_{yi}}{r^3}$,

(3) $\bar{x}_{i+1} = \bar{x}_i + \bar{v}_{x_{i+1}}\Delta t$,

(4) $\bar{y}_{i+1} = \bar{y}_i + \bar{v}_{y_{i+1}}\Delta t$.

Such equations of motion are Euler-Cromer style (positon calculations use updated velocity) to ensure stability. Thus, equations (1), (2), (3), and (4) can be used to numerically integrate the positions and velocites of a planet governed by Newton's Law of Gravity.

```
#Pseudocode
#create blank lists to hold the position and velocity information
#populate the [0] value of the blank lists for position and velocity with the initial conditions of the
#determine the amount of timesteps(ns) needed to get 1 years worth of time from a timestep of dt
#loop over the implemented equations from part 1a ns times, effectively numerically integrating the pos
#plot the lists of time, positions and velocities
```

### c)

The graph of angular momentum displays that angular momentum is constant from the start of the integration to the end. This implies that angular momentum is conserved, as the total angular momentum at the beginning of the integration is equal to the total angular momentum at the end of the integration.

### d)

The precession of Mercury's orbit can be shown easily between the first and last orbit plotted. In the first orbit, the perihelion (approximated visually) is at (+0.4AU,0AU) this is shown in Cyan. In the last plotted orbit, the perihelion (approximated visually) is at just (-0.4AU,0.1AU) this is shown in Red. Thus, the furthest point of the orbit from (0,0) clearly changes with time. Additionally, angular momentum is still conserved with the addition of GR.

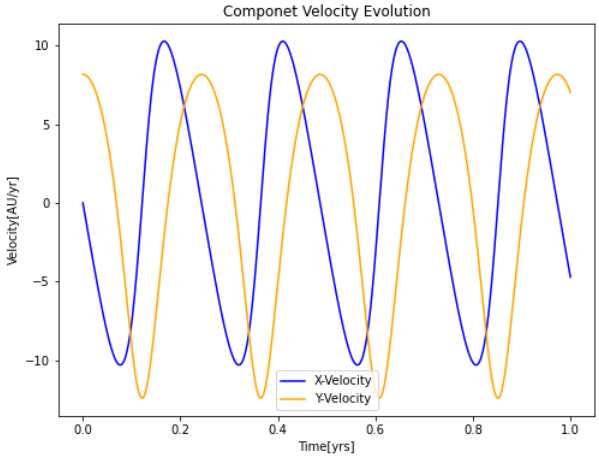


*Figure 1*

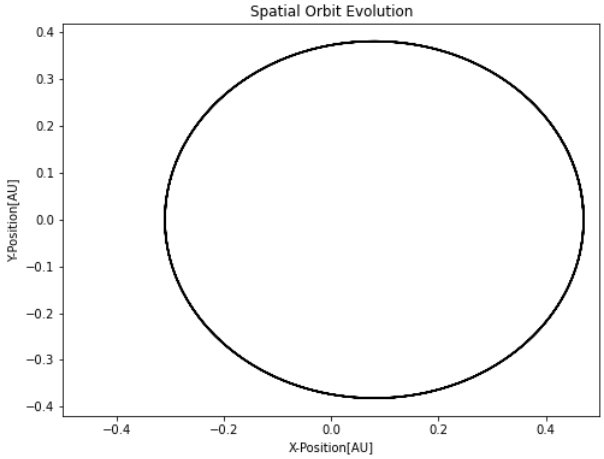


*Figure 2*

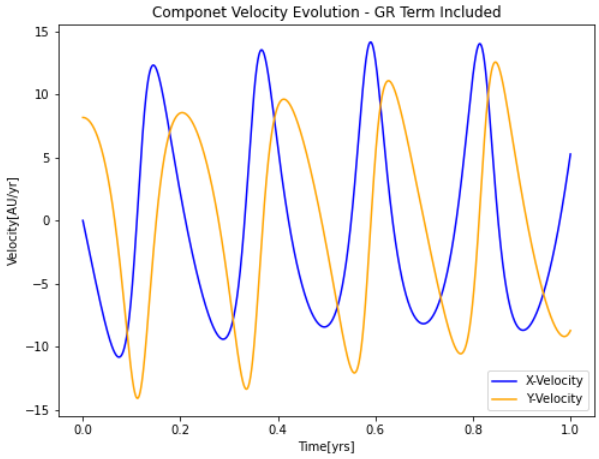


*Figure 3*

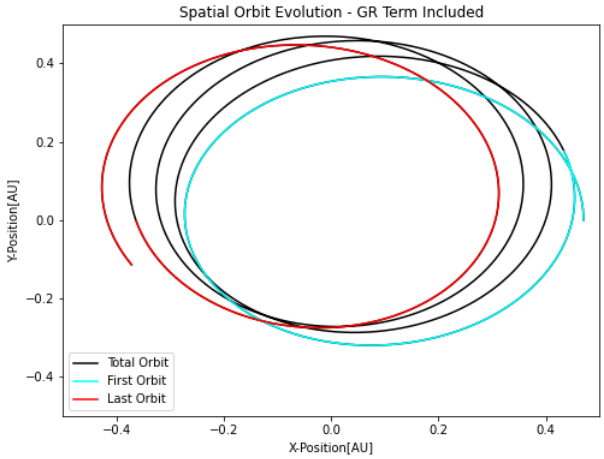


*Figure 4*

## Code for #1

```
#1c)
#import needed modules
import numpy as np
import matplotlib.pyplot as plt

#specify timestep(dt) and total time(T)
dt=0.0001
T=1

#determine the number of steps that an integration time of dt will require to reach a total time T
ns=int(T/dt) + 1 #+1 to include the zero timestep

#specify the time domain
t=np.linspace(0,T,ns)
```

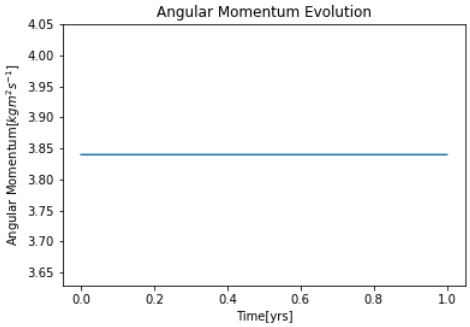


*Figure 5*

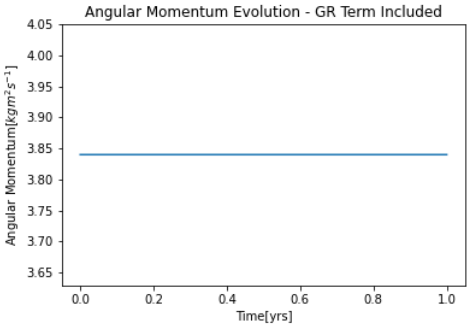


*Figure 6*

```
#specify initial conditions as the first entry in the lists that will hold all orbital information
x=np.zeros(len(t))
y=np.zeros(len(t))
vx=np.zeros(len(t))
vy=np.zeros(len(t))
x[0]=0.47
y[0]=0
vx[0]=0
vy[0]=8.17
Ms=1
G=39.5

#loop the number of timesteps dt needed to reach total time T.
for i in range(ns-1):
    vx[i+1]=(vx[i]-((G*Ms*x[i]*dt)/(np.sqrt(x[i]**2 + y[i]**2))**3))
    vy[i+1]=(vy[i]-((G*Ms*y[i]*dt)/(np.sqrt(x[i]**2 + y[i]**2))**3))
    x[i+1]=(x[i]+vx[i+1]*dt)
    y[i+1]=(y[i]+vy[i+1]*dt)

#I manually checked the angular momentum (L=rxp) is conserved over time
r=[]
v=[]

#create lists of r and v in component form. ieL r=[[x(t[0]),y(t[0])], [x(t[1]),y(t[1])], ...]
for i in range(len(x)):
    r.append([x[i],y[i]])
    v.append([vx[i],vy[i]])\

#compute L-evolution by taking the cross product of r and v for each timestep
L=[]
for i in range(len(r)):
    L.append(np.cross(r[i],v[i]))

#round L to 2 decimal places for sensible plotting
for i in range(len(L)):
    L[i]=np.around(L[i],2)

#plot a figure for velocity component evolution
plt.figure(figsize=[8,6])
```

```
plt.plot(t,vx,color='blue',label='X-Velocity')
plt.plot(t,vy,color='orange',label='Y-Velocity')
plt.xlabel('Time[yrs]')
plt.ylabel('Velocity[AU/yr]')
plt.title('Componet Velocity Evolution')
plt.legend()
plt.show()
plt.close()
#plot a figure for spatial evolution
plt.figure(figsize=[8,6])
plt.plot(x,y,color='black')
plt.xlabel('X-Position[AU]')
plt.ylabel('Y-Position[AU]')
plt.title('Spatial Orbit Evolution')
plt.xlim([-0.5,0.5])
plt.show()
plt.close()
#plot a figure that shows angular momentum evolution
plt.plot(t,L)
plt.xlabel('Time[yrs]')
plt.ylabel('Angular Momentum[$kg m^{2} s^{-1}$]')
plt.title('Angular Momentum Evolution')
plt.show()
plt.close()

#1d)
#specify timestep(dt) and total time(T)
dt=0.0001
T=1

#specify the GR constant alpha (a)
a=0.01

#determine the number of steps that an integration time of dt will require to reach a total time T
ns=int(T/dt) + 1 #+1 to include the zero timestep

#specify the time domain
t=np.linspace(0,T,ns)

#specify initial conditions as the first entry in the lists that will hold all orbital information
x_gr=np.zeros(len(t))
y_gr=np.zeros(len(t))
vx_gr=np.zeros(len(t))
vy_gr=np.zeros(len(t))
x_gr[0]=0.47
y_gr[0]=0
vx_gr[0]=0
vy_gr[0]=8.17
Ms=1
G=39.5

#loop the number of timesteps dt needed to reach total time T. INCLUDE a factor of (1+a/r) to accound f
for i in range(ns-1):
    vx_gr[i+1]=(vx_gr[i]-((G*Ms*x_gr[i]*dt)*(1+(a/(np.sqrt((x_gr[i]**2 + y_gr[i]**2)))**2))/(np.sqrt(x_g
```

```
    #print((a/((x[i]**2 + y[i]**2))**2))
    vy_gr[i+1]=(vy_gr[i]-((G*Ms*y_gr[i]*dt)*(1+(a/(np.sqrt(x_gr[i]**2 + y_gr[i]**2))**2))/(np.sqrt(x_gr
    x_gr[i+1]=(x_gr[i]+vx_gr[i+1]*dt)
    y_gr[i+1]=(y_gr[i]+vy_gr[i+1]*dt)

#check angular momentum again in the same way as 1c)
r_gr=[]
v_gr=[]
for i in range(len(x_gr)):
    r_gr.append([x_gr[i],y_gr[i]])
    v_gr.append([vx_gr[i],vy_gr[i]])
L_gr=[]
for i in range(len(r_gr)):
    L_gr.append(np.cross(r_gr[i],v_gr[i]))
for i in range(len(L_gr)):
    L_gr[i]=np.around(L_gr[i],2)

#plot a figure for velocity component evolution
plt.figure(figsize=[8,6])
plt.plot(t,vx_gr,color='blue',label='X-Velocity')
plt.plot(t,vy_gr,color='orange',label='Y-Velocity')
plt.xlabel('Time[yrs]')
plt.ylabel('Velocity[AU/yr]')
plt.title('Componet Velocity Evolution - GR Term Included')
plt.legend()
plt.show()
plt.close()
#plot a figure for spatial evolution
plt.figure(figsize=[8,6])
plt.plot(x_gr,y_gr,color='black',label='Total Orbit')
plt.plot(x_gr[:2250],y_gr[:2250],color='cyan',label='First Orbit')
plt.plot(x_gr[len(x_gr)-2250:],y_gr[len(y_gr)-2250:],color='red',label='Last Orbit')
plt.xlabel('X-Position[AU]')
plt.ylabel('Y-Position[AU]')
plt.title('Spatial Orbit Evolution - GR Term Included')
plt.xlim([-0.5,0.5])
plt.ylim([-0.5,0.5])
plt.legend()
plt.show()
plt.close()
#plot a figure that shows angular momentum evolution
plt.plot(t,L_gr)
plt.xlabel('Time[yrs]')
plt.ylabel('Angular Momentum[$kg m^{2} s^{-1}$]')
plt.title('Angular Momentum Evolution - GR Term Included')
plt.show()
plt.close()
```

# Question 2

## a)

```
#Pseudocode:
#Define initial normalized population x0 between [0,1] and the reproduction rate r
```

```
#Define an array for the years p starting at 0 until some pmax (say pmax=50 years)
#Define an array of zeros the length of pmax for xp
#Set the first element of the array xp to x0
#For pmax # of iterations:
#   Calculate the next xp value using the previous value according to equation (12)
#Plot xp vs. p (p on x-axis, xp on y-axis)
#Label axes and plot
```

## c)

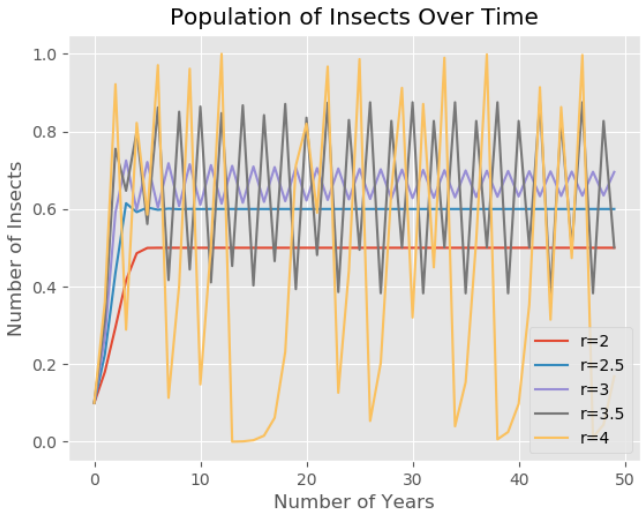


*Figure 7*

For low values of r, specifically r=2 and 2=2.5, the evolution of the population increases steadily until it is saturated. Once saturated, the population remains steady. As r increases, the maximum population also increases as can be seen from Fig. (1) - the normalized population goes from a maximum of approximately 0.7 with r=3 to 1.0 with r=4. Furthermore, when r≥3, the normalized population oscillates, dipping down to lower numbers and back up to higher numbers repeatedly. The higher the r value, the more extreme the oscillations are.

## d)

Both Figures 1 and 2 show that higher values of r lead to what appears to be random fluctuations in the insect population. A period doubling bifurcation occurs at approximately r=3 in Fig. 2 where the data splits and begins oscillating between values of xp. This corresponds to the oscillating that is seen for r=3 in Fig. 1 with a period of about 2 years. Another period doubling bifurcation occurs at approximately r=3.45, where the data splits again, now oscillating between four values of xp. In fig.1 this can be seen for r=3.5, where the data oscillates between four values of xp, with the period now doubled at approximately 4 years. At approx. r=3.55 the periodicity of the oscillations turn into chaos, with no distinct period or patterns. This is seen in Figure 1 for r=4 as there is no identifiable period and the system seems to rise and fall at random.

## e)

Figure 3 shows that there is a stable region within the chaotic region around $3.738 \leq r \leq 3.745$. The system stabilizes and begins oscillating between 5 values of xp, and bifurcations occur around 3.741. See 2g) for code.

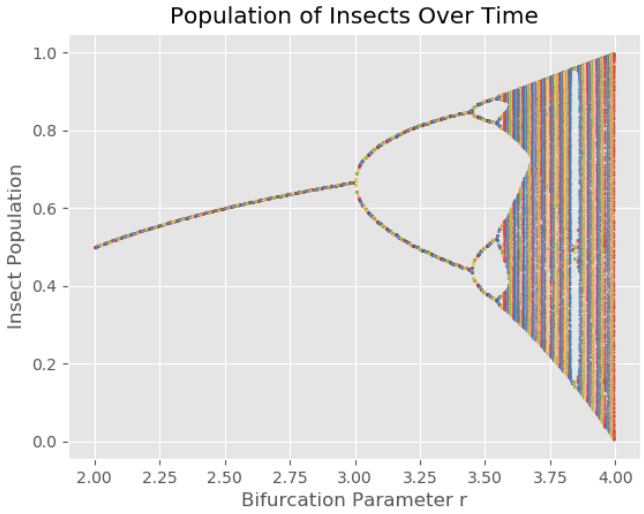


*Figure 8*

## f)

The parameter x0 was chosen to be x0=0.1 since it proved a good initial population in the other exercises, and $\epsilon$ was generated using the random number generator code from the beginning of the lab. The number of iterations pmax was chosen as 100 to give the populations a chance to evolve differently. In 2f) we see that the saturation time occurs at about p=30 years, but that small amount of time does not allow us to see how the addition of $\epsilon$ affects the evolution of the population. See Figure 4 for the plot.

## g)

An estimate for the numerical value of $\lambda$ is $\lambda = 0.45$, and $a = 5.0e-6$, found through testing various numbers until arriving at a good eyeball agreement. See Figure 5 for the plot.

## Code for #2

```
import numpy as np
import matplotlib.pyplot as plt
#define constants
n0=100 #initial population
r=2 #max reproduction rate
pmax=50 #total number of years
x0=100/1000 #initial normalized population
xp=np.zeros(50) #empty array for the xp values to go in later
nmax=1000 #max pop of insects sustainable

#set first value of xp as the initial x0
xp[0]=x0

#%%
#2c
#compute equation 12 for 50 years/iterations
for i in range(49):
    xp[i+1]=r*(1-xp[i])*xp[i]
```

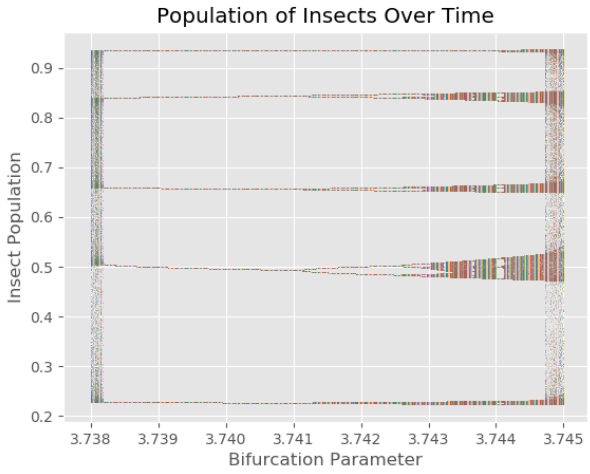


*Figure 9*

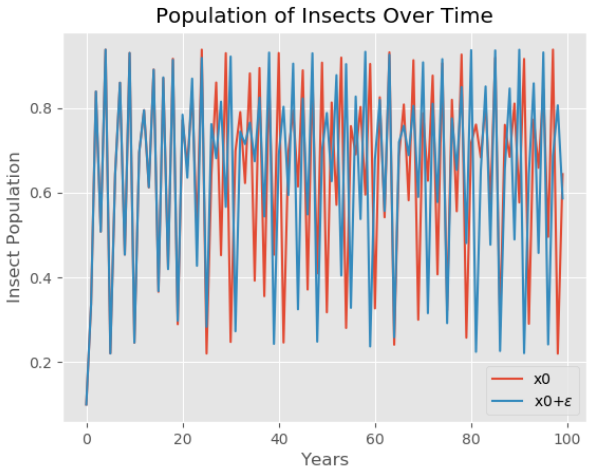


*Figure 10*

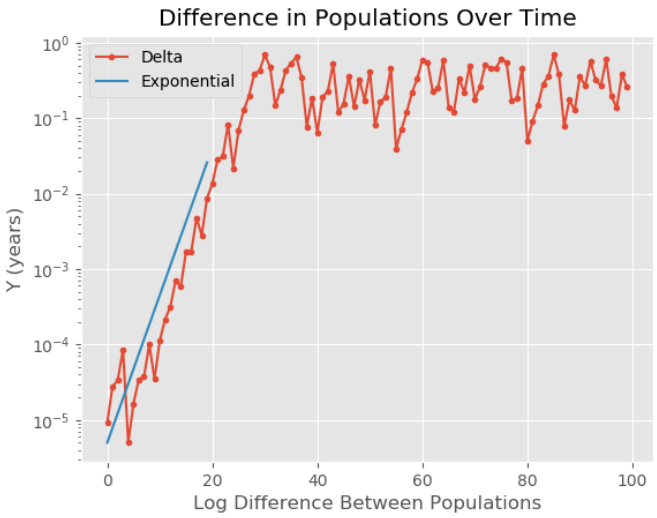


*Figure 11*

```
#repeat for different r values
r2=3
xp2=np.zeros(50)
xp2[0]=x0
for i in range(49):
    xp2[i+1]=r2*(1-xp2[i])*xp2[i]

r3=4
xp3=np.zeros(50)
xp3[0]=x0
for i in range(49):
    xp3[i+1]=r3*(1-xp3[i])*xp3[i]

r4=2.5
```

```
xp4=np.zeros(50)
xp4[0]=x0
for i in range(49):
    xp4[i+1]=r4*(1-xp4[i])*xp4[i]
r5=3.5
xp5=np.zeros(50)
xp5[0]=x0
for i in range(49):
    xp5[i+1]=r5*(1-xp5[i])*xp5[i]

#plot figure with years on x-axis and xp on yaxis
plt.figure(1)
plt.style.use('ggplot')
plt.plot(np.arange(0,50,1), xp, label="r=2")
plt.plot(np.arange(0,50,1), xp4, label="r=2.5")
plt.plot(np.arange(0,50,1), xp2, label="r=3")
plt.plot(np.arange(0,50,1), xp5, label="r=3.5")
plt.plot(np.arange(0,50,1), xp3, label="r=4")
plt.title("Population of Insects Over Time")
plt.xlabel("Number of Years")
plt.ylabel("Number of Insects")
plt.legend()

#%%
#2d
#define new values of constants according to the question
r_new=np.arange(2, 4, 0.005)
pmax_new=2000
xp_new=np.zeros([400, 2000]) #2d array of zeros

#set the first column to be x0, each row is xp for a different value of r
for i in range(400):
    xp_new[i,0]=x0

#calculate equation 12 for 400 different xp's over 2000 years
for i in range(1999):
    for j in range(400):
        xp_new[j, i+1]=r_new[j]*(1-xp_new[j,i])*xp_new[j,i]

#bifurcation plot
plt.figure(2)
for i in range(200):
    plt.scatter(np.ones([100,1], float)*r_new[i], xp_new[i][1900:2000], s=0.3) #plots last 100 xps
for i in range(200, 400, 1):
    plt.scatter(np.ones([1000,1], float)*r_new[i], xp_new[i][1000:2000], s=0.3)#plots last 1000 xps
plt.title("Population of Insects Over Time")
plt.ylabel("Insect Population")
plt.xlabel("Bifurcation Parameter r")

#%%
#2e
#define a new r array between the values stated in the question
r_e=np.arange(3.738, 3.745, 1e-5)
```

```
#2d array of xps, each row is xp for a different value of r, with 2000 columns for the years
xp_e=np.zeros([len(r_e), 2000])

#set first column to x0
for i in range(len(r_e)):
    xp_e[i,0]=x0

#calculate equation 12 for len(r_e) different xp's over 2000 years
for i in range(1999):
    for j in range(len(r_e)):
        xp_e[j, i+1]=r_e[j]*(1-xp_e[j,i])*xp_e[j,i]

#plot the r values vs last 1000 xp values
plt.figure(3)
plt.title("Population of Insects Over Time")
plt.ylabel("Insect Population")
plt.xlabel("Bifurcation Parameter")
for i in range(701):
    plt.scatter(np.ones([1000,1], float)*r_e[i], xp_e[i][1000:2000], s=0.01)

#%%
#2f
# import random function from random module
from random import random
randomNum = random()

#define constants
r_f=3.75
x0_f1=0.1
x0_f2=0.1 + random()/1e5 #add a random number much smaller than x0
pmax_f=100

#make empty arrays for values to be stored in later
xp_f1=np.zeros(pmax_f)
xp_f2=np.zeros(pmax_f)

#set first element in each array to the corresponding x0
xp_f1[0]=x0_f1
xp_f2[0]=x0_f2


#calculate eq 12 for both xps
for i in range(pmax_f-1):
        xp_f1[i+1]=r_f*(1-xp_f1[i])*xp_f1[i]
        xp_f2[i+1]=r_f*(1-xp_f2[i])*xp_f2[i]

#plot years vs xp
plt.figure(4)
plt.plot(np.arange(0,pmax_f,1), xp_f1, label="x0")
plt.plot(np.arange(0,pmax_f,1), xp_f2 , label="x0+$\epsilon$")
plt.title("Population of Insects Over Time")
plt.ylabel("Insect Population")
plt.legend()
plt.xlabel("Years")
```

```
#%%
#2g
#calculate delta as defined in the question
delta=np.zeros(pmax_f)
for i in range(pmax_f-1):
    delta[i]=np.abs(xp_f2[i]-xp_f1[i])

#define an exponential function
def expo(p):
    expo=a*np.exp(l*p)
    return expo
#test values that give a good estimate
a=5.0e-6
l=0.45

#plot years vs delta
plt.figure(5)
plt.semilogy(np.arange(0,pmax_f,1),np.abs(xp_f2-xp_f1), marker=".", label="Delta")
plt.semilogy(np.arange(0,20,1), expo(np.arange(0,20,1)), label="Exponential")
plt.xlabel("Log Difference Between Populations")
plt.ylabel("Y (years)")
plt.title("Difference in Populations Over Time")
plt.legend()
```

# Question 3

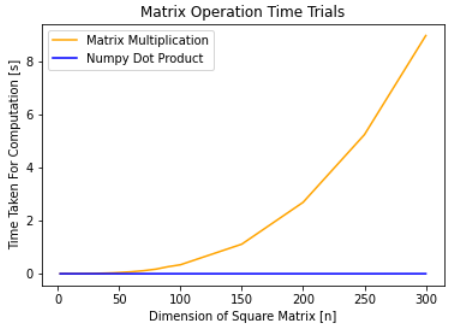


*Figure 12*

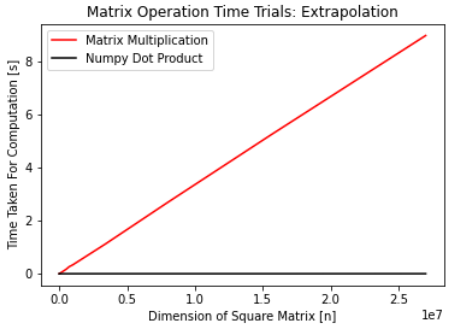


*Figure 13*

Clearly matrix multiplication with python loops is much slower than with the numpy dot function. For example, multiplying two square matricies with 300 columns is approximately 9000 times faster using numpy dot than it is with python loops. Plotting the values of time against their respective array-dimension, it seems that python loops are exponentially slower than numpy dot calculations. Plotting values of time against their respective dimension cubed seems to instead imply a linear trend.

## Code for #3

```
#list values of n to loop over
ns=[2,4,6,8,10,15,20,25,30,45,50,60,70,80,90,100,150,200,250,300]
#create lists that will hold time information
times_mm=[]
times_nd=[]
```

```
times_mc=[]
#loop through all chosen values of n
for i in ns:
    start1=time() #set a start time for the matrix creation (unneeded)
    A = np.ones([i,i],float)*7
    B = np.ones([i,i],float)*4 #set up matricies
    C = np.zeros([i,i],float)
    matrix_creation=time()-start1

    start2=time()
    for i in range(i): #from page 137 of the textbook
        for j in range(i):
            for k in range(i):
                C[i,j] += A[i,k]*B[k,j]
    matrix_multiplication=time()-start2

    times_mc.append(matrix_creation)
    times_mm.append(matrix_multiplication)

    start3=time()
    D = np.dot(A,B)
    matrix_dot=time()-start3

    times_nd.append(matrix_dot)

plt.plot(ns,times_mm,color='orange',label='Matrix Multiplication')
plt.plot(ns,times_nd,color='blue',label='Numpy Dot Product')
plt.xlabel('Dimension of Square Matrix [n]')
plt.ylabel('Time Taken For Computation [s]')
plt.title('Matrix Operation Time Trials')
plt.legend()
plt.savefig('3_1')
plt.show()
plt.close()
plt.plot(np.array(ns)**3,times_mm,color='red',label='Matrix Multiplication')
plt.plot(np.array(ns)**3,times_nd,color='black',label='Numpy Dot Product')
plt.xlabel('Dimension of Square Matrix [n]')
plt.ylabel('Time Taken For Computation [s]')
plt.title('Matrix Operation Time Trials: Extrapolation')
plt.legend()
plt.savefig('3_2')
```