Lab1 Report

Zirui Wan, Rundong Zhou

September 18, 2020

1 Qestion 1

1.1 Part (b)

1.1.1 Pseudo code

Main function with Euler-Cromer method

```
def Euler_Cromer(x_0, y_0, vx_0, vy_0, d_t, t):
    steps = t / d_t     #calculate total number of steps
    x = array(steps)    #initialize arrays with length of steps
    y = array(steps)
    vx = array(steps)
    vy = array(steps)
    x[0] = x_0          #take initial values
    y[0] = y_0
    vx[0] = vx_0
    vy[0] = vy_0
    for i in range(0, steps-1):         #Euler Cromer method
        r = sqrt(x[i] ** 2 + y[i] ** 2) #calculate r
        x[i+1] = x[i] + vx[i] * d_t
        vx[i+1] = vx[i] - G * M_s * x[i+1] * d_t / r**3
        y[i+1] = y[i] + vy[i] * d_t
        vy[i+1] = vy[i] - G * M_s * y[i+1] * d_t / r**3
    return x, y, vx, vy
```

Plot

```
result = Euler_Cromer(x_0, y_0, vx_0, vy_0, d_t, t)
#run the simulation
new_plot()
#plot velocities
plot(x, vx, color = 'red')
plot(y, vy, color = 'blue')

new_plot()
```

```
#plot x vs. y in space
plot(x, y)
```

1.1.2 Explanation

Euler-Cromer function

In the first part of the pseudo code, I defined the function `Euler_Cormer` to take initial values of x, y, v_x, v_y as `x_0, y_0, vx_0, vy_0` respectively. Also, the function takes the total time interval as `t` and the time interval of each step as `d_t` to calculate the total number of steps as `steps`.

Then the function initializes four arrays `x, y, vx, vy` with length of `steps` to store the results. The first element of each array is set as the initial value which is taken by the function.

Then the function calculates the values step by step using Euler Cromer method with for loops.

The difference between Euler Cromer method and regular Euler method is that, when calculating v_{i+1}, Euler Cromer method uses x_{i+1} while regular Euler method uses x_i.

So, in my code, I calculate `x[i+1]` first and use its value to calculate `vx[i+1]`.

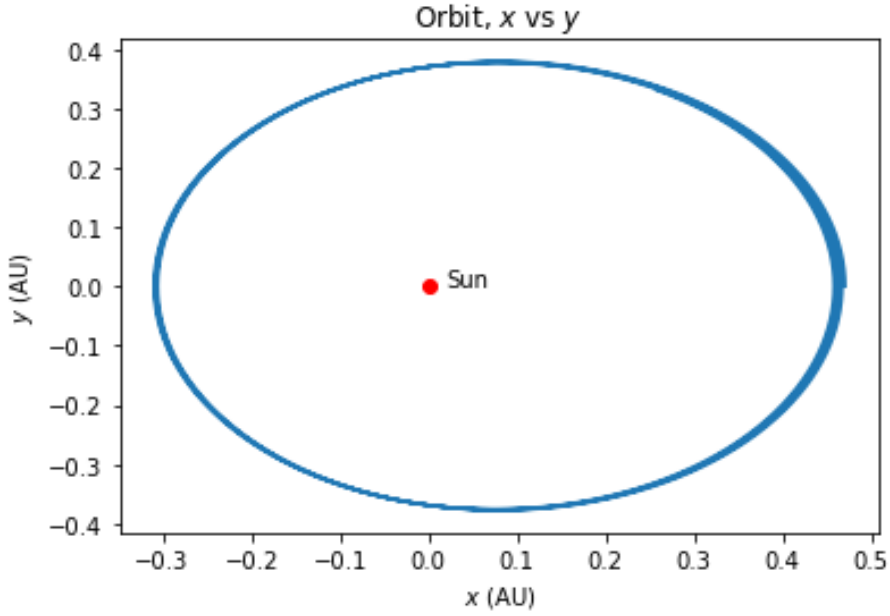
The function returns four arrays `x, y, vx, vy` which contains the simulation results.

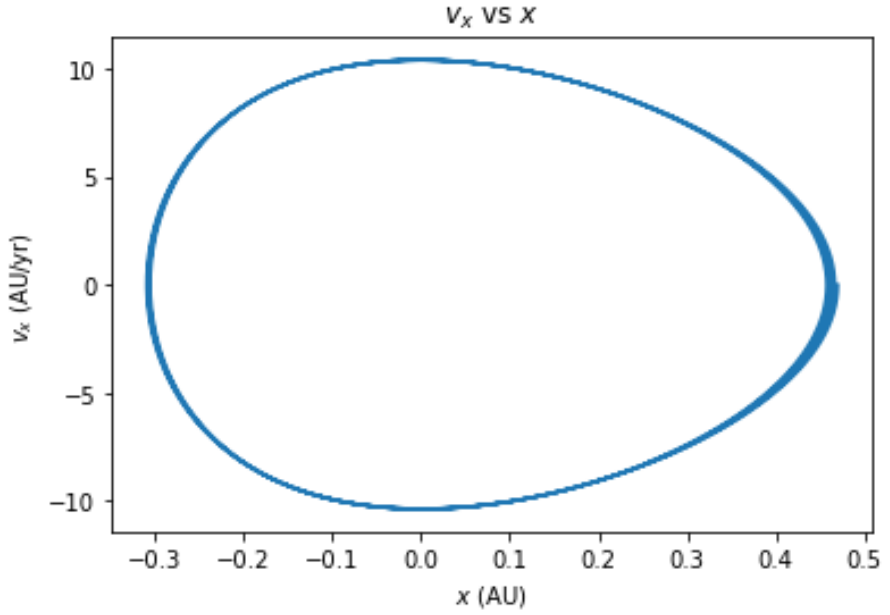
Plot

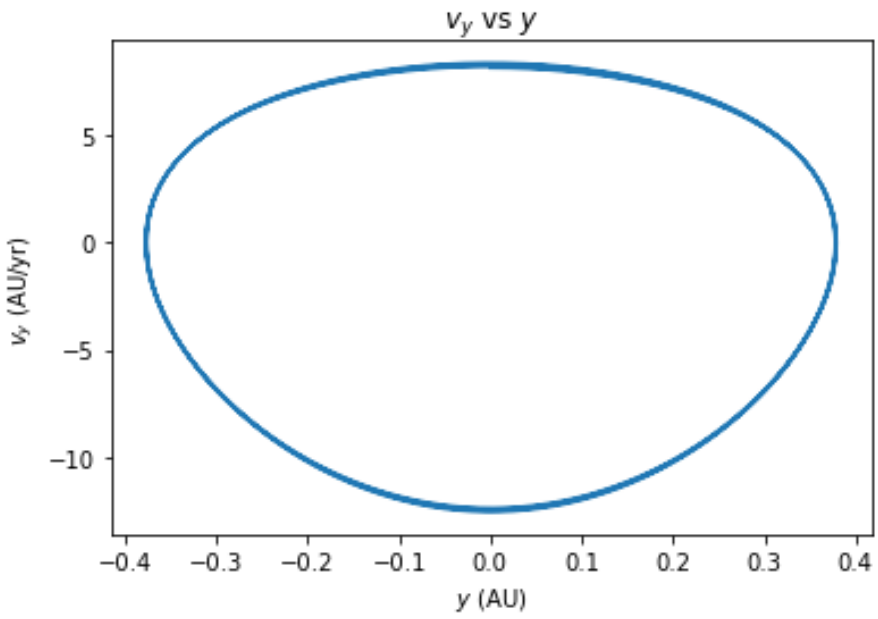
The plot part is very straight forward. It runs the simulation with `Euler_Cormer` function, and plots the desired result.

1.2 Part (c)

1.2.1 Plots







1.2.2 Explanation

Function
The function itself is basically the same as the pseudo code I showed in Q1(b). The differences are I imported the `Scipy` constant package in the header:

```
import scipy.constants as spc
```

and so I changed the gravitational constant `G` into `spc.G`.
I also defined the mass of sun `M_s` in the beginning of the function:

```
M_s = 2*(10**30)
```

Plot and Simulation
Since all the given values are in AU and year, I converted them into SI units:

```
x_0 = 0.47 * spc.au
y_0 = 0
vx_0 = 0
vy_0 = 8.17 * spc.au / 31622400
dt = 31622400 * 0.0001
t = 31622400
```

After the simulation is done, I converted the results back to AU and year for plotting convenience:

```
x = [i / spc.au for i in result[0]]
y = [i / spc.au for i in result[1]]
```

```
vx = [i * 31622400 / spc.au for i in result[2]]
vy = [i * 31622400 / spc.au for i in result[3]]
```

and plot them all.

1.2.3 Angular momentum

Please note, all the calculations are rounded to 2 significant digits.
At the beginning:

$$|v_0| = v_y = 8.17 \text{ AU/yr}$$

$$r_0 = x = 0.47 \text{ AU}$$

So

$$L_0 = mvr = m \cdot 3.84 \text{ AU}^2/\text{yr}$$

At the end of simulation, we obtain the value by:

```
x_f = x[int(t/dt)-1]
y_f = y[int(t/dt)-1]
vx_f = vx[int(t/dt)-1]
vy_f = vy[int(t/dt)-1]
print(x_f, y_f, vx_f, vy_f)
```

So

$$x_f = 0.26 \text{ AU} \qquad y_f = 0.33 \text{ AU}$$

$$v_{xf} = -8.19 \text{ AU/yr} \qquad v_{yf} = 4.38 \text{ AU/y}$$

$$r_f = \sqrt{{x_f}^2 + {y_f}^2} = 0.42 \text{ AU}$$

$$|v_f| = \sqrt{{v_{xf}}^2 + {v_{yf}}^2} = 9.29 \text{ AU/y}$$

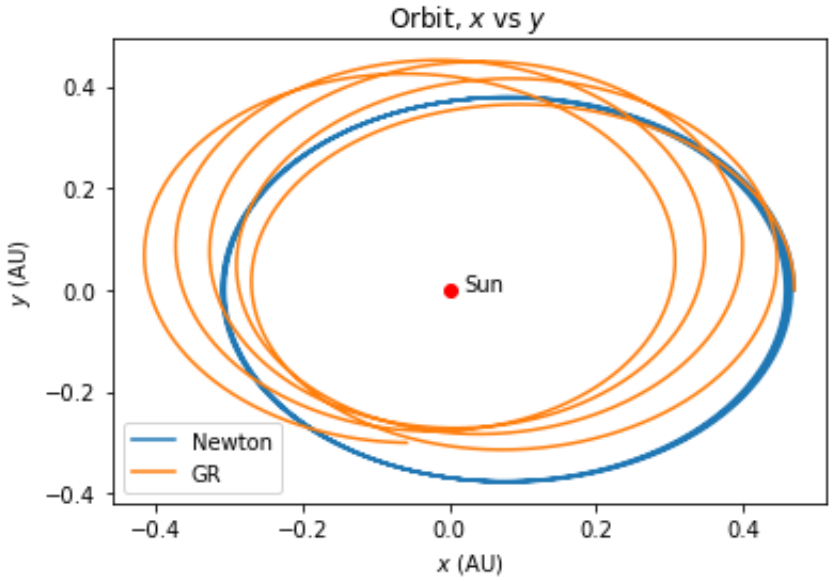
and

$$L_f = mvr = m \cdot 3.90 \text{ AU}^2/\text{yr}$$

We can tell that the angular momentum is differed due to the error of simulation.

1.3 Part (d)

1.3.1 Plots



1.3.2 Explanation

So I created a new function `Euler_Cromer_GR` with an additional line to define $\alpha = 0.01\text{AU}^2$:

```
a = 0.01 * spc.au**2
```

And I changed the calculation of speed to:

```
vx[i+1] = vx[i] - (1 + a/r**2) * spc.G * M_s * x[i+1] * d_t / r**3
```

for the additional factor $(1 + \frac{\alpha}{r^2})$.
From the plot we can tell that there is a precession happening to Mercury's orbit.

2 Question 2

2.1 Part (a)

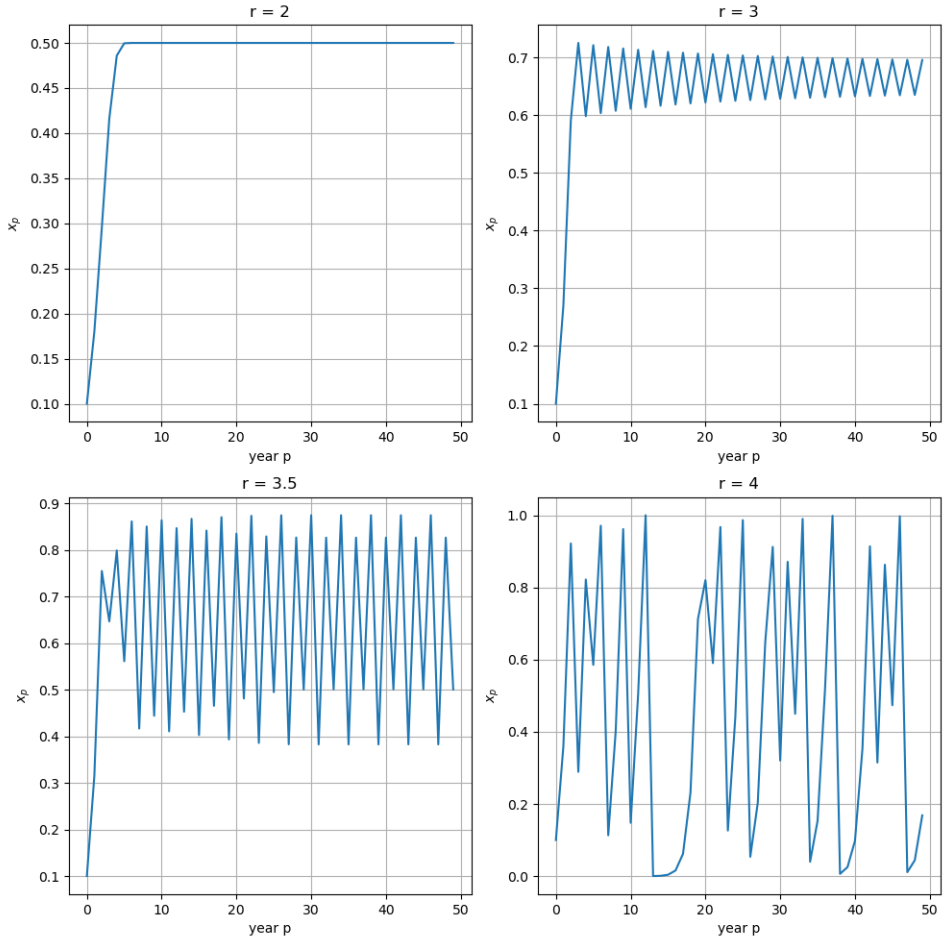
2.1.1 Pseudo code

```
r = some constant  # define the maximum reproduction rate r
x = np.zeros(( number of years ))   # define an array of x with size of number of years

# loop over years to calculate x for each year
for p in range(1, pmax):
    x[p] = r*(1 - x[p-1])*x[p-1]    # calculate xp and store in the array
```

2.2 Part (c)

2.2.1 Plots

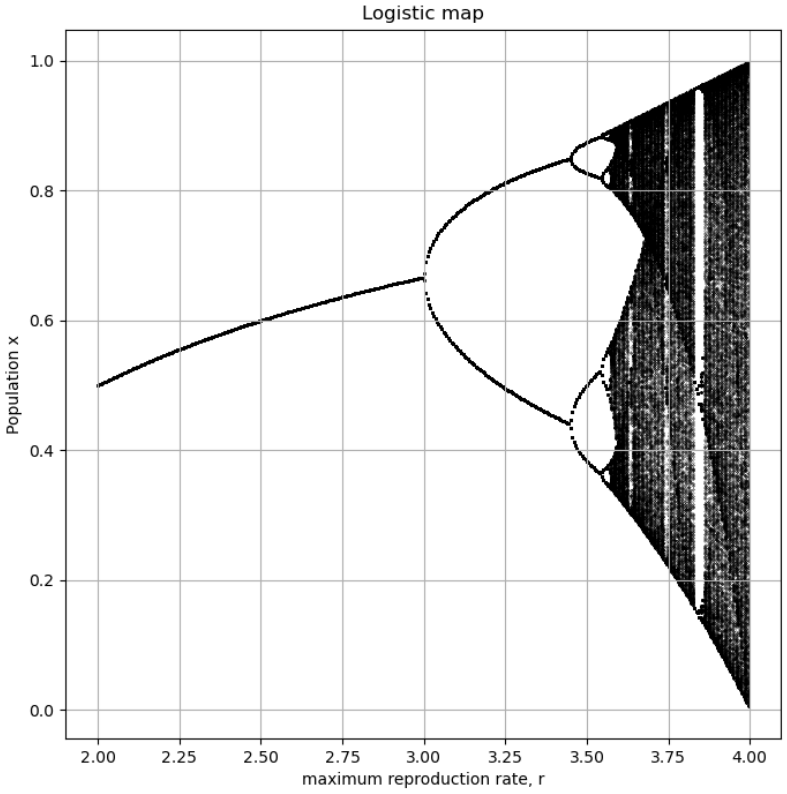


2.2.2 Comments

As shown in the figure above, the system transits from the stable regime to the chaotic regime. When $r = 2$, there is 1 stable state of population $x_p = 0.5$ after the system stabilizes. Similarly, there would be 2 ($x_p = \{0.63, 0.7\}$) and 4 ($x_p = \{0.39, 0.5, 0.83, 0.88\}$) stable states after the stabilization, for $r = 3$ and $r = 3.5$ respectively. However, when $r = 4$, the system becomes chaotic and the population x_p changes to a random state every year. This means that the threshold of r for the regime transition could be estimated to be between 3.5 and 4.

2.3 Part (d)

2.3.1 Plots

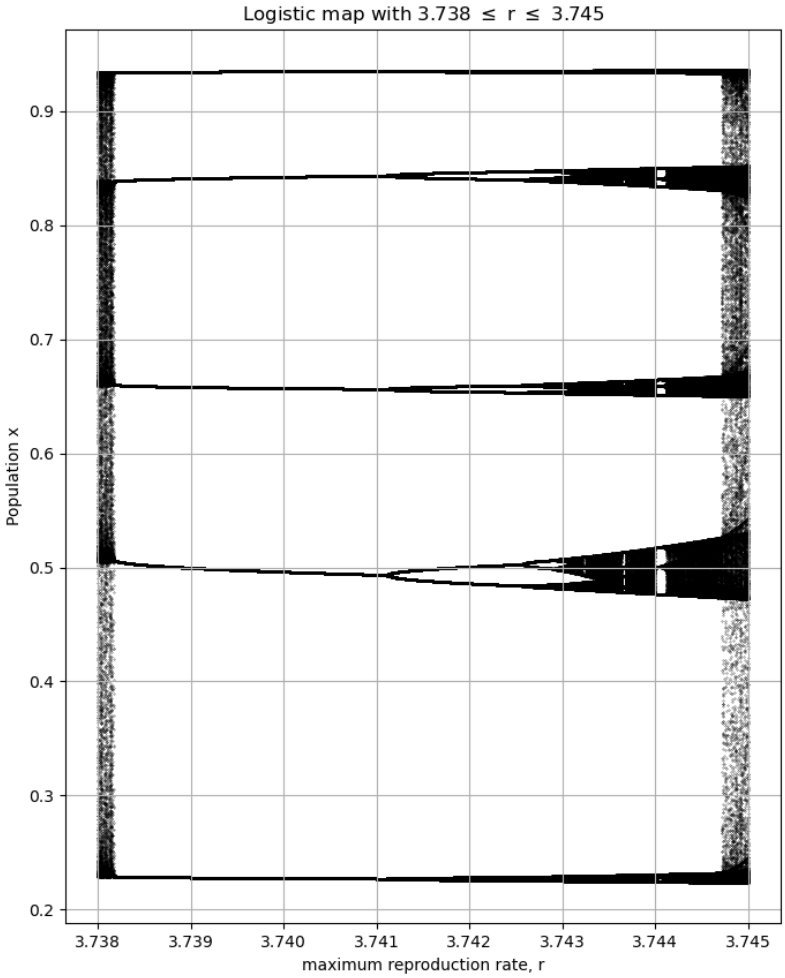


2.3.2 Comments

As shown in the figure above, this bifurcation diagram shows the value of states after the stabilization of the system. When the value of r is between 2.00 and 3.00, there would be only $2^0 = 1$ stable state of population x_p. This regime corresponds to the top left panel in Part (c) figure. The value of this stable x_p increases with the value of r. Similarly, for r between 3.00 and 3.45, there exist 2^1 stable states, corresponding to top right panel in Part (c) figure. Between 3.45 and 3.55, the number of stable states further increases to $2^2 = 4$, which corresponds to bottom left panel in Part (c) figure. This suggests that the number of stable states would increase following an 2-based exponential rule ($\mathcal{O}(2^n)$). And at $r = 4$, the number of possible states of x_p is so large that the system would just behave chaotic by randomly choosing one of these values every year. The threshold of value of r between stable and chaotic regime could be estimated to be 3.65 from this bifurcation diagram.

2.4 Part (e)

2.4.1 Plots



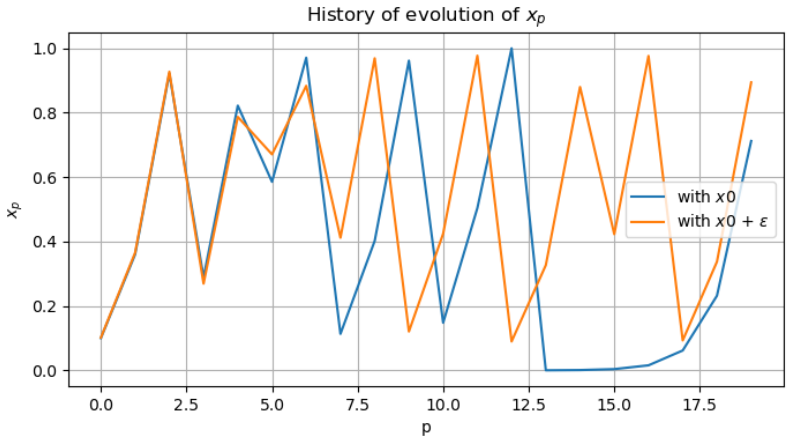
2.4.2 Comments

As shown in the figure above, we can see that indeed, each sub-branch of the state of x_p will follow another same bifurcation structure. Namely, as the value of r increases, each state will double the number of itself just like its "parent

state" did before.

2.5 Part (f)

2.5.1 Plots

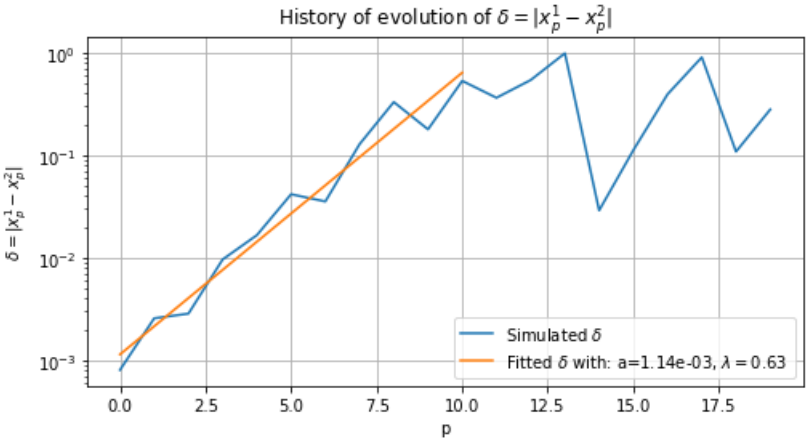


2.5.2 Comments

Initially I tried the original value of $p_{max} = 50$. And I found that the system saturated to the maximum allowed population x_{max} quickly. So I gradually decreased the number of iterations by changing p_{max} to smaller values. After several trials, I found that the system would always saturate before 10 iterations. For the purpose of better illustration, I chose the value of $p_{max} = 20$. r remains at 4 to ensure a chaotic regime, and other parameters remain the same as Part (c).

2.6 Part (g)

2.6.1 Plots

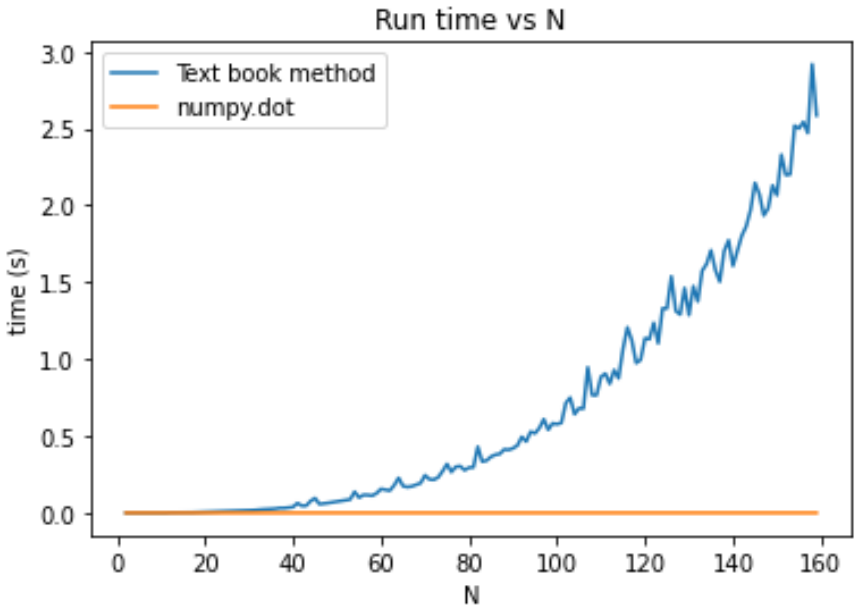


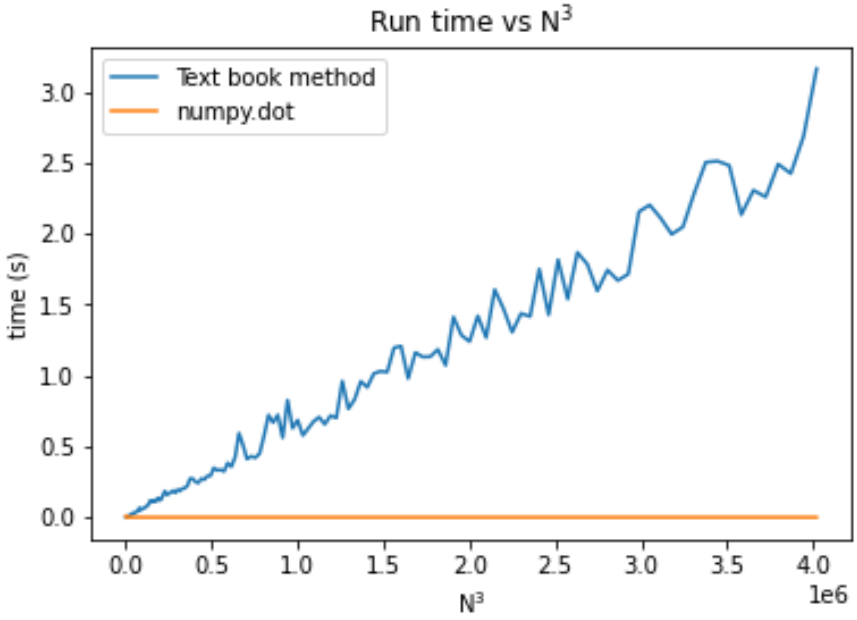
2.6.2 Comments

As shown in the figure above, we can see that the Lyapunov exponent λ could be estimated to be 0.69, and the initial value of δ could be estimated to be very small ($a = 1.14e - 3$), which makes sense because the two systems are almost identical before the evolution of population x_p.

3 Question 3

3.1 Plots





3.2 Explanation

The textbook method has a complexity of $\mathcal{O}(N^3)$. It takes extremely long when N gets high, so I only went up to 160. When its run-time is plotted against N, it tends to curve up, and it shows the tendency of a straight line when it is plotted against N^3.
While `numpy.dot` does not show much difference in run-time for N up to 160. It must have some way better strategy to calculate matrix multiplication.