PHY407 (Lab 1: Introduction to Python)

Utkarsh Mali[1] and Aslesha Pokhrel[1,2]

[1]Department of Physics, University of Toronto
[2]Department of Computer Science, University of Toronto

September 17, 2020

Contents

Work Allocation:
Utkarsh: Question 1, Question 3 (Write-up)
Aslesha: Question 2, Question 3 (Code)

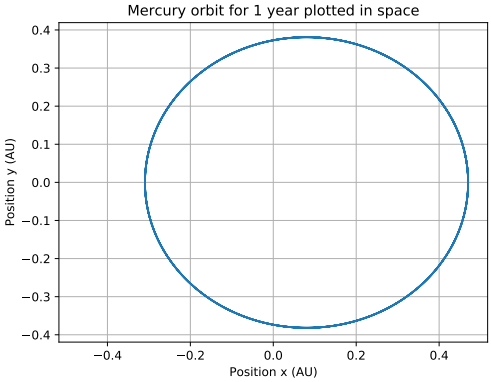
Question 1

(a) **In this question we are tasked with modeling the orbit of planets (mercury) using the Euler-Cromer method as well as demonstrating our competency in python.**
No answer required to this question.

(b) **Here we are asked to develop pseudo-code that calculates position and velocity of the planets orbit as a function of time as well as plot the orbit in x-y space.**

```
   # Pseudo Code Q1b #
""" Write a pseudo- code  for a program that integrates (using Euler-Cromer) your
equations to calculate the position and velocity of the planet as a function of
time under the Newtonian gravity force. The output of your code should include
graphs of the components of velocity as a function of time and a plot of the
orbit (x vs. y) in space.
"""
# gravitational constant (in terms of solar mass and au)
# initial x position
# initial y position
# initial x velocity
# initial y velocity
# time step
# empty array for x position
# empty array for y position
# empty array for time after time step
# empty array for x velocity
# empty array for y velocity
# empty array for angular momentum
# add initial x position to x array
# add initial y position to y array
# add initial time 0 to time array
# add initial x velocity to vx array
# add initial y velocity to vy array
# while loop calculating dt x and y positions and velocities (while time <= 1 year)
#      calculate and update angular momentum
#      update vx using equation 6a
#      add updated vx to vx array
#      use updated vx to update x (Euler-Cromer method)
#      update vy using equation 6b
#      add updated vy to vy array
#      use updated vy to update y (Euler-Cromer method)
#      update x,y arrays with new values
#      update time with time step
#      add updated time to time array
# new figure
# plot appended x and y array in 2d space
# label axis
# new figure
# plot appended vx and vx as a function of time
# label axis
```

(c) **We now convert this pseudo-code into python code. We use the initial conditions given by the question and also check if angular momentum is conserved.**
We noticed angular momentum was always between 6.48e-07 and 6.34e-07 hence not fully conserved. We do not expect the angular momentum in elliptical orbits to always be conserved since there is a $\sin\theta$ factor between the angle to the center of the orbit and the actual perpendicular vector to v. The full solution is given in the attached code.



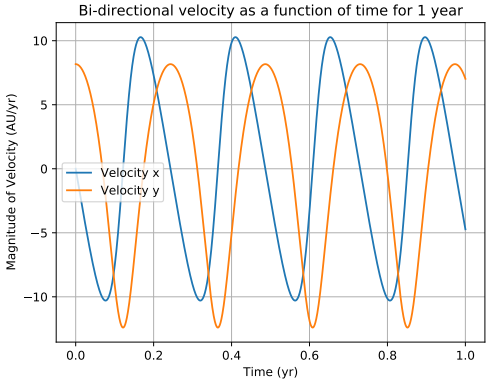
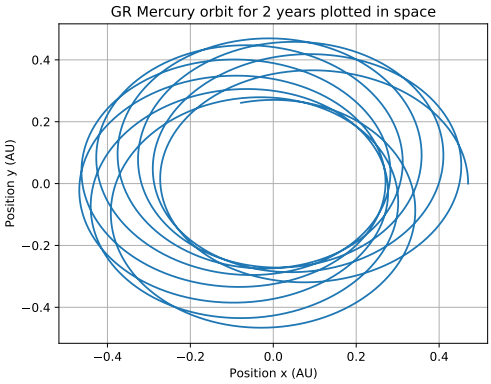


Figure 1: Here we plot an earth years orbit of mercury. We notice a slight offset in the Mercury orbit which is expected. Additionally, more than 1 orbit can be seen here since we are plotting an earth year instead of a mercury year.

Figure 2: The bi-directional velocity for the orbit is shown, it shows the velocity at each stage in time as an earth year passes. We notice a oscillating pattern in which the x velocity peaks slightly before the y velocity. Notice that when either x or y are at the max their respective counterpart is 0 representing characteristic circular orbit.

(d) **We now alter the code we have already written to account for general relativity with an increased α in order to show visible results in the plot. Here we decided to plot 2 years instead of 1 to show the perihelion moving with time.** The full solution is given in the attached code.



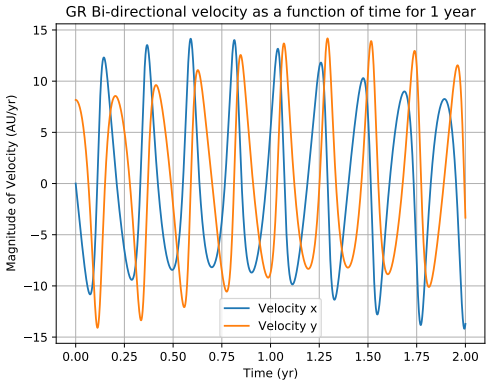


Figure 3: Here we see the Mercury orbit with the new alteration accounting for general relativity. The orbit is plotted for 2 earth years and shows the perihelion moving with time. The effects have been exaggerated due to a larger α.

Figure 4: The bi-directional velocity for the orbit is shown, it shows the velocity at each stage in time as an earth year passes. We notice a oscillating pattern in which the x velocity peaks slightly before the y velocity. We also notice a general oscillating patter which envelops the smaller one, this was due to general relativity and the shifting perihelion nature of the oscillations.

Question 2

(a) Here, we use logistic map equation to analyze the evolution of population over time.

The logistic map as defined in the handout is re-stated below:

$$x_{p+1} = r(1 - x_p)x_p \qquad \text{(where } x_p = n_p/n_{max} \in [0, 1]\text{)}$$

The pseudo-code to compute and plot the evolution of a population insects described by the equation above, as a function of the number of years p is given below:

```
Set the initial normalized population x_0
Set the maximum reproduction rate r
Set the total number of years p_max
Initialize a population_list with x_0 as the first element
for each year p in 1, ... p_max:
```

```
    append the population x_p to the population list calculated by using the formula
        x_p = r * (1 - population_list[p-1]) * population_list[p-1]
plot population (population_list) vs years ([0,1, ..., p_max])
```

(b/c) Here, we use the pseudo-code above to write a function that returns a list of population x_p for each year p given the initial condition $x_0 = 0.1$ and total number of years $p_{max} = 50$ for various values of reproduction rate (r) between 2 and 4. Then, we plot the evolution of population over time for various reproduction rates.

The plot for evolution of population for various reproductions rate (r) is shown in *Figure 5*.

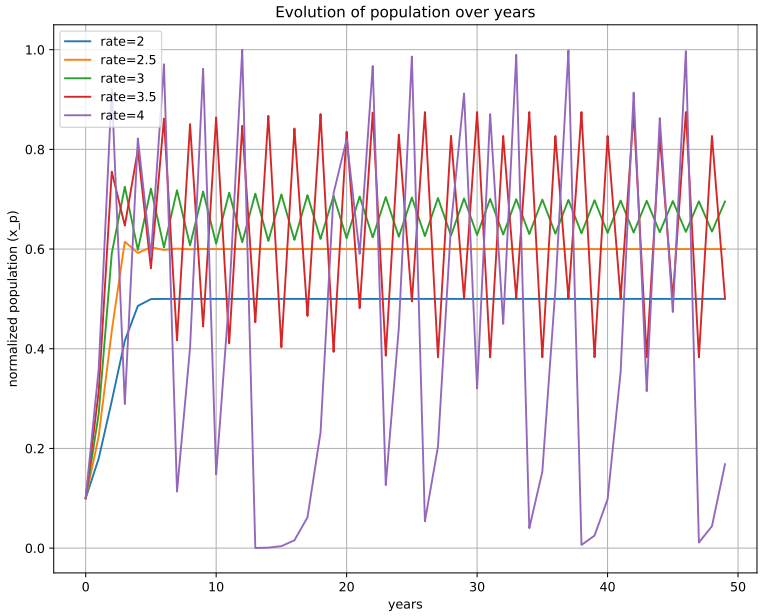


Figure 5: Plot showing the evolution of population over years for various values of max reproduction rate (r).

From *Figure 5* we can see that for $r = 2$ and $r = 2.5$, the population converges to one value. Similarly for $r = 3$ and $r = 3.5$, the population seems to be oscillating between a few values. However, for $r = 4$, the population seems to be oscillating between various values without any fixed pattern.

(d) Here, we add the code for plotting the bifurcation diagram where the horizontal axis represents values of r and the vertical axis shows values of x_p for a given r.

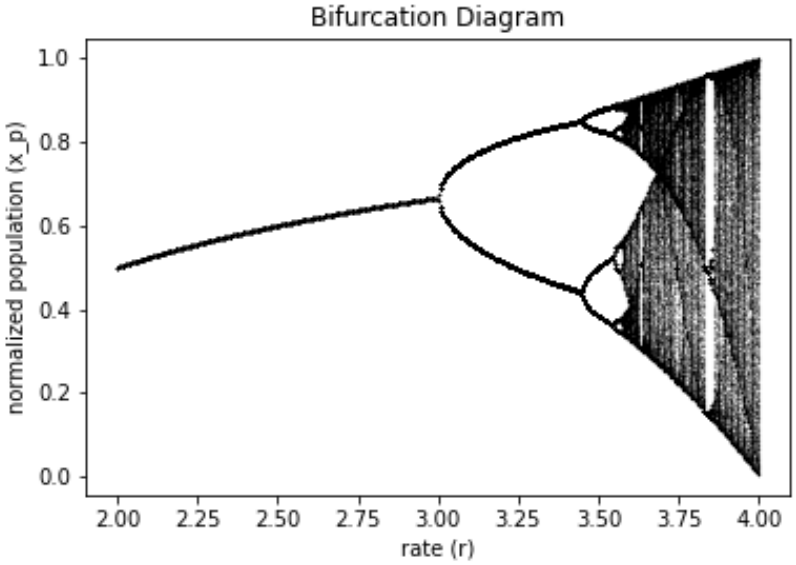


Figure 6: Bifurcation diagram using the rate of reproduction r as the bifurcation parameter.

The bifurcation diagram here agrees with the behaviour seen in the previous plot x_p vs. p as it shows the number of different population the logistic map converges to for a given value of r. That is, the bifurcation diagram shows that for r between 2 and ~ 3, the population converges to one value; for r above 3 and below ~ 3.45, population converges to 2 values and so on as we noticed in the previous plot.

The "period doubling" narrative fits here as we can see that the number of ways population converges, doubles at various points as r increases. For instance, near $r = 3$, a slight change in r value can lead the population to converge to two different values compared to just one value before that. Similarly, the number of ways population converges doubles approximately at $r = 3.45$, $r = 3.55$ and so on. Further, the notion of "chaos" fits here as we can see that for the higher values of r, the population doesn't seem to converge to any finite number of fixed values making the system chaotic. Beyond approximately $r = 3.60$, the system can be considered chaotic as it is hard to tell the number of values the population converges to.

(e) Here, we look at the bifurcation diagram for $3.738 \leq r \leq 3.745$**.**

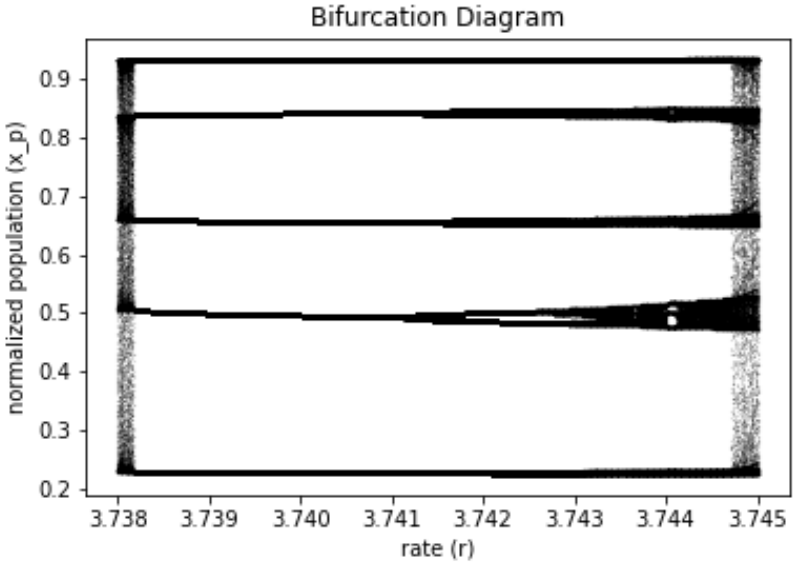


Figure 7: Bifurcation diagram for $3.738 \leq r \leq 3.745$.

From *Figure 7*, it seems that the population converges to a finite number of values for $3.739 \leq r \leq 3.744$ but around $r \approx 3.738$ and $r \approx 3.745$, the population can converge to infinite number of values. Similarly, for $3.738 \leq r \leq 3.745$, the normalized population never goes below 0.2.

(f) Here, we experiment with a major feature of chaos i.e. the sensitivity to initial conditions to see how a small perturbation in the initial condition leads to a different evolution of population over the years.

The value of r was chosen to be 3.745 as we can see from *Figure 7* that it will lead to a chaotic behaviour. Similarly, the initial condition ($x_0^{(1)}$) was set to be 0.2 and the second initial population ($x_0^{(2)}$) was slightly changed from the first by adding a random ϵ in the order of 10^{-13} to $x_0^{(1)}$. Likewise, the total number of years was set to 150. *Figure 8* shows the evolution of population over years for the initial condition and it's slightly perturbed version. We can see that the evolution of population starts to vary for two initial conditions after approximately 80 years.

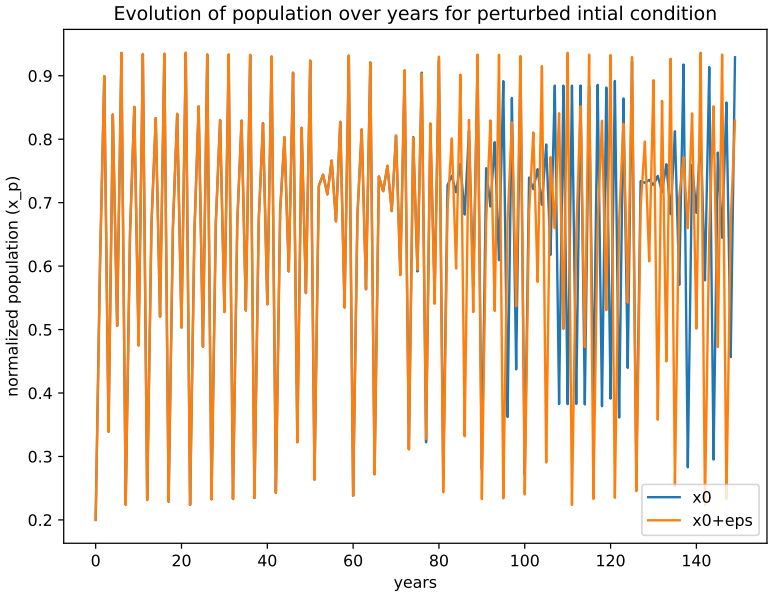


Figure 8: Evolution of population over years for a slightly perturbed initial condition.

(g) Continuing from (f) here we plot the difference $\delta = |x_p^2 - x_p^1|$ using a semi-log scale for the vertical axis.

From *Figure 9* we can see that spread grows almost exponentially until 80 years then saturates there after as the population is bounded. Before the saturation, the spread δ can be modeled as $ae^{\lambda p}$. The rough estimate of the numerical value of λ in this system was found to be -32.88 with $a \approx 0.33$ which was obtained by curve fitting the first 100 values to $ae^{\lambda p}$.

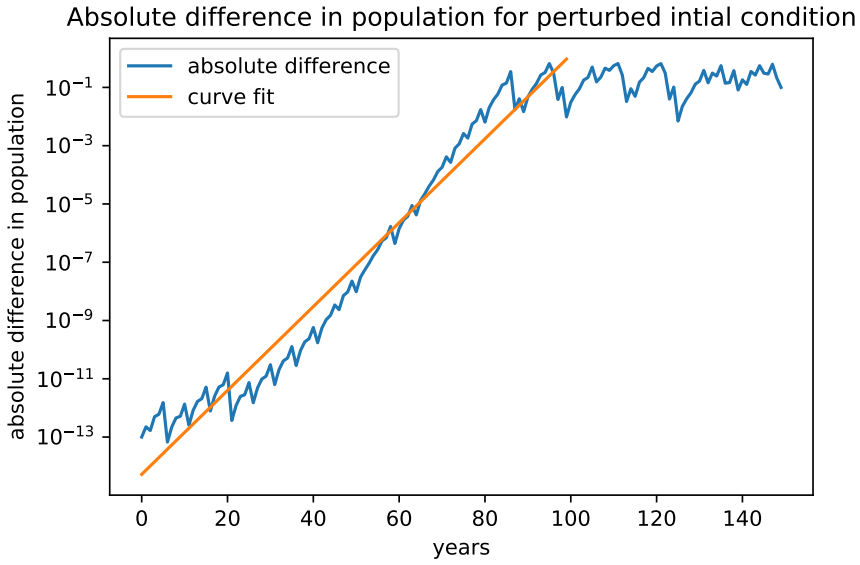


Figure 9: Plot showing the difference in population for two initial conditions.

Question 3

We referred to Example 4.3 in the Course Textbook and used pythons time feature to calculate the time taken for each matrix multiplication. We then plotted time taken as a function of N and N^3 dimension of matrix. We found that the `numpy.dot()` method was much faster than manual matrix multiplication given in example 4.3 of the textbook. This highlights the speed and optimization of the numpy code.

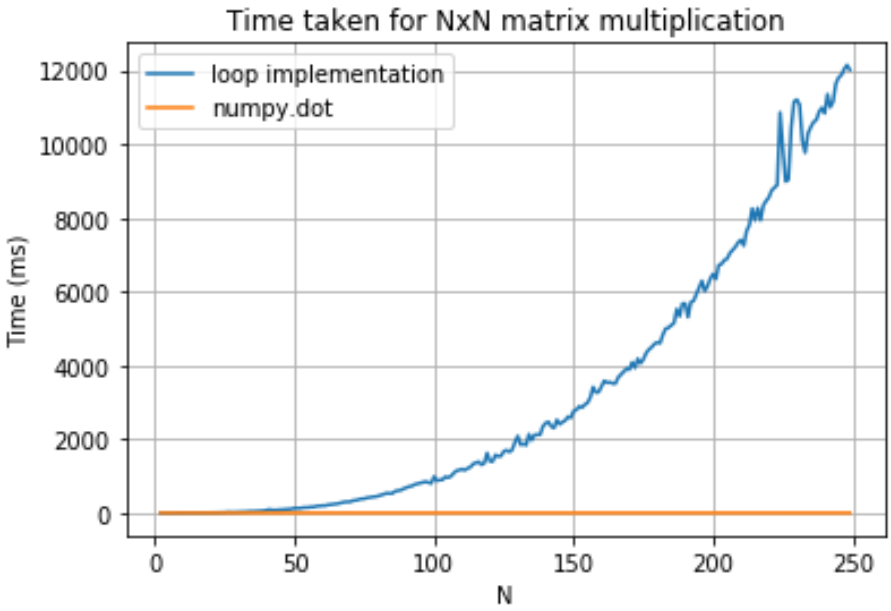


Figure 10: This plot shows the relationship between the time taken for N by N matrix multiplication and the size of the matrix N. It can be seen that while the loop implementation of the multiplication increases exponentially while the `numpy.dot()` implementation takes a much shorter (almost constant) time to execute.

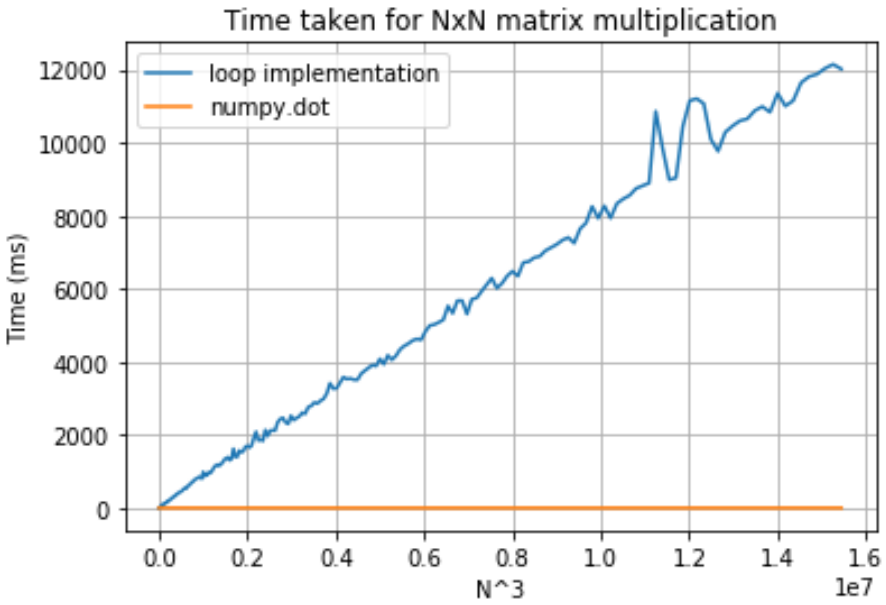


Figure 11: This plot is similar to the plot above but with a linear relationship for the run-time increase instead of an exponential one in the loop implementation of the multrix multiplication. The `numpy.dot()` still takes a much shorter time to execute and highlights the optimization power of numpy.