Computational Physics Lab Assignment 1

Satchel Page Vincent Fan

Question 1:

a) We rearranged the equations provided in the lab 1 handout to be in a form which allows for us to time step. We obtained the following equations to integrate into our python code:

- **v_x += c * x * delta_time**
- **v_y += c * y * delta_time**
- **x += v_x * delta_time**
- **y += v_y * delta_time**

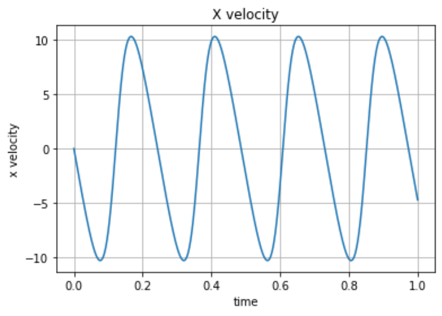
Where "c" is a constant defined as ($-G*M_s/r^3$)

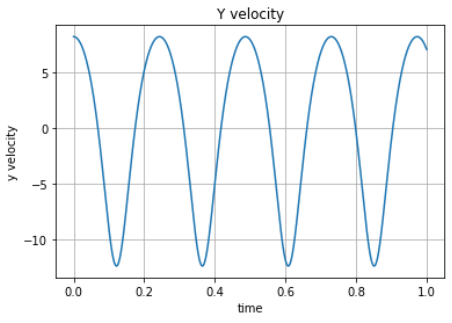
b)
```
for $\Delta t$ in year
        $v_{x\,i+1} = v_{x\,i}$ – gravity const * $x_i$ *$\Delta t$
        $v_{y\,i+1} = v_{y\,i}$ – gravity const * $y_i$ * $\Delta t$
        $x_{i+1} = x_i – v_{y\,i} * \Delta t$
        $y_{i+1} = y_i – y_{y\,i} * \Delta t$
        vxList.append($v_{x\,i+1}$)
        vyList.append($v_{y\,i+1}$)
        xList.append($x_{i+1}$)
        yList.append($x_{i+1}$)
        update gravity const
plot(xList,yList)
plot(vxList,year)
plot(vylist,year)
```

since both the velocity and position depends on each other's value, we have to calculate their values within the same Δt, so I put all the operations within a single for loop that loops over a year. Noticed that $v_{x\,i+1}$, $v_{y\,i+1}$, and x_{i+1}, y_{i+1}, would use their values before each loop, that is $v_{x\,i}$, $v_{y\,i}$, and x_i, y_i. Then, we would also need to update the gravity constant since the position has changed, thus the radius has changed as well. Finally, we would simply plot the graphs of x vs y and each velocity components versus time over the year.

c) There are a few things we want to achieve here. First, we need to use our python code to model the components of velocity (x and y) for the planet Mercury as a function of time. Then we need our program to output a plot of the position of the planet mercury in space to show an elliptical orbit, and finally we need to check that angular momentum is conserved over the entire process.

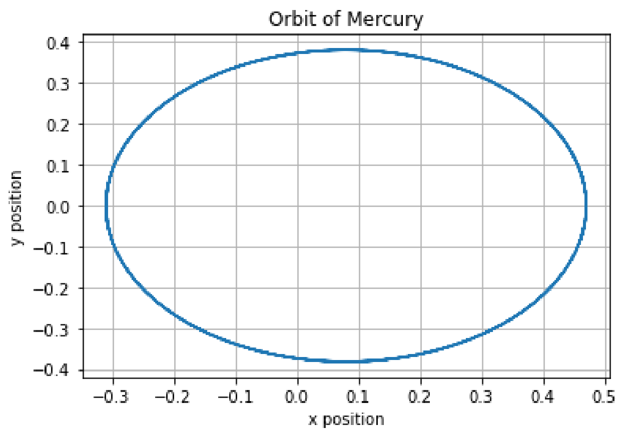
For the first part, the lab manual provided lists a few initial conditions that we can use to start modelling. These being:
x = 0.47 AU
y = 0.0 AU
vx = 0.0 AU/yr
vy = 8.17 AU/yr
In order to satisfy the first requirement, we wrote a program which takes an initial velocity, and time steps based on the formula given in part 1a). Time is written in years in our program, and is a list from 0 –> 1 increasing by steps of 0.0001. We can apply these times to the function we defined as v_x in our program, as well as v_y to see how the component velocities evolve over the span of a year. When plotted, we obtain the following figures



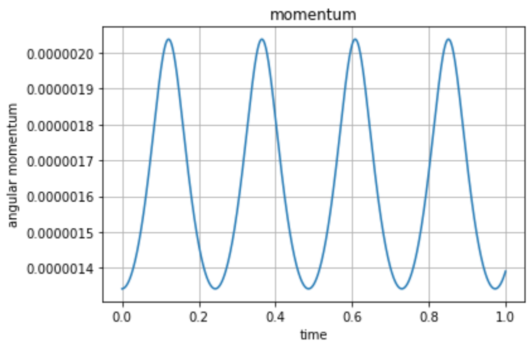


Displaying the two component velocities, gives us a nice image of how they are evolving in time. These plots make a lot of sense when we think about how the components should be changing as the planet moves around the sun. Oscillating between high positive and negative values as the planet passes through equilibrium, and hitting zero as the planet moves perpendicular to the component of motion specified.

Since these looked good, we now applied the formulas derived to times step the position of the planet. We created a list for the resulting x and y values, and used the same time list as we had used for the velocities. Plotting the function as x vs y outputted the following model:

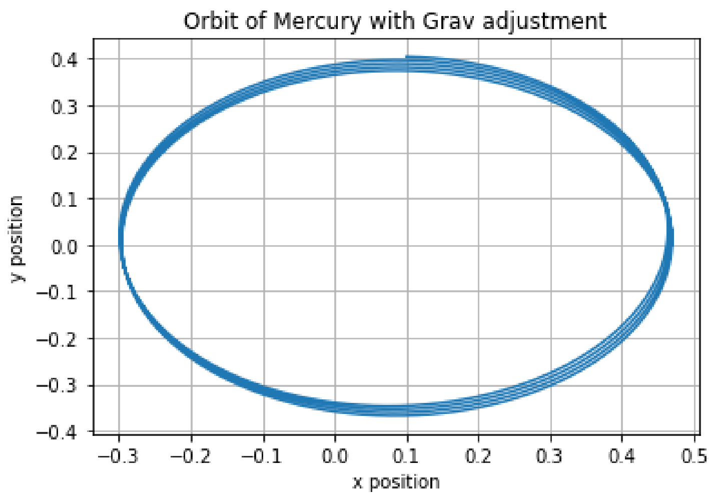


We can clearly see the elliptical pattern of Mercurys' motion, with a slightly skew in the x position towards the right of the origin. Now to check that angular momentum was conserved, we calculated the total angular velocity at each interval by defining a function which takes our component velocities, squares them, and then takes the root of their sum (Pythagorean theorem), which we can do because our components are always orthogonal to one another. Multiplying this function by the Mass of the planet yields its momentum, so we put the momentum into a list and allowed it to evolve in time by the same method we used for our component velocities. We then plotted the result, yielding:



Although it fluctuates slightly, that could easily be due to a programming error, as the scale by which it fluctuates is very small, so we are inclined to conclude that angular momentum is conserved.

d) The objective for this section is to remodel the orbit of Mercury, only this time by changing the gravitational force by a fairly exaggerated amount. The only change to our program in this question was the constant we tagged onto the velocity components "c". We changed this value to have an initial value of: - (39.5 / (0.47 ** 3)) * (1.0 + (0.01 / (0.47 ** 2))) to incorporate the constants provided. Then we changed how we update the constant inside the for loop (See python code for more detailed changes). The rest of the program remained the same. The output of this plot yielded the following figure:



This beautifully captures the clear precession of Mercury's orbit over a year. We expected multiple loops because Earths year (which is the time we are using here) is much longer than Mercurys, so it makes sense that it would complete a few cycles in the time allotted.

Question 2:

The idea for the question is to explore the scattering effects of a particle hitting a cylinder by a trajectory along the x-axis. The direction of scatter will depend on where the particle hits the cylinder, or in other words the height of the particle when it makes contact with the cylinder. The angle of scatter with respect to the positive x-axis will be Theta, and the height of the particle with respect to the center of the cylinder will be defined as "z". We want to write a program which will randomly select heights (z), and output the angle of scatter corresponding to those heights in the form of a histogram.

b)

```
for z in lists of random z
    $\theta = 180^o - 2*\sin^{-1}(z)$
    append $\theta$ to angle list
for i in angle list
    if $175^o \leq i \leq 185^o$:
        count1++
    if $20^o \leq i \leq 30^o$:
        count2++
relative prob of (175->185) = count1/sample size
relative prob of (20->30) = count2/sample size
plot(angle list)
```

First we would have a list of random z, and for each z we would calculate the outgoing θ and add it to a list for the histogram. Then to calculate the relative probabilities, we simple go through the list of angles and find how many is within the ranges and divide them by the sample size.

c) To program this, we started by creating a random number generator for our values of z. to make sure our values were between -1 and 1 (rather than 0 and 1) we wrote z as z = random()*2 – 1. Then we incorporated the function we thought would best model the resulting angle. We needed a height of 0 to output and angle of 180 degrees, so we used that as our initial condition. To incorporate z, we knew theta would be dependent on some sine function because z would be perpendicular to our direction of travel (opposite to theta). After a small amount of trial and error inputting different values and checking outputs, we settled on

theta = 180 - 2 * np.arcsin(z)*180/np.pi

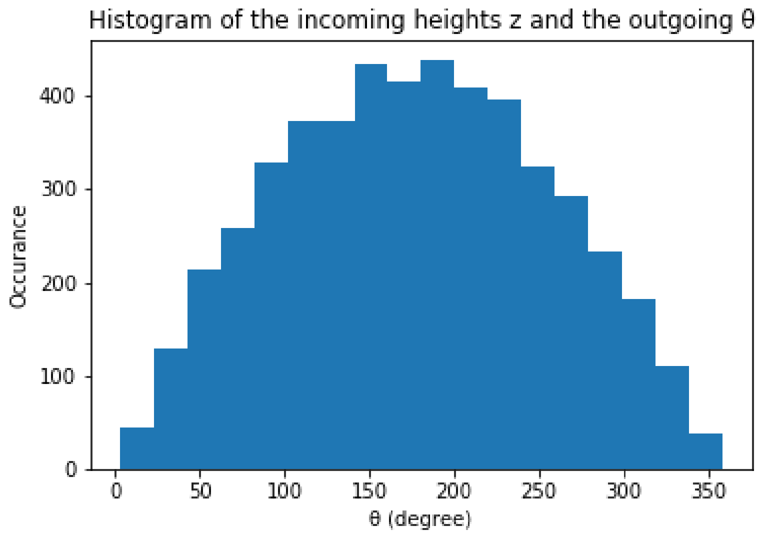
Where 180/np.pi is just there to convert from radians to degrees.

The majority of the programming is done by creating lists for each variable and choosing a sample size, and then creating a histogram. We also however want to check whether certain angles are more likely to be achieved than others. More specifically, whether an angle in the range of 170-190 degrees is more common than an angle of 90-110, and by how much roughly.

Rather than comparing a bunch of sample values and counting which happens more often, we created code to count the amount of angles in the range 170°-190°, as well as 90°-110° using the "if" function, and then divide the numbers out to obtain a ratio of the number of events in each range, which is approximately the relative probabilities. Using a sample size of 2500, we obtained the relative probabilities of 0.0858 for 170°-190° and 0.0676 for 90°-110° respectively, suggesting that values in the range of 170°-190° have a higher relative frequency/probability compared to angles between 90°-110° deg by around 2%.

This of course is only one trial, and not a definite solution as the answer varies with each output. Doubling the sample size instead will yield a more accurate answer on average, so we tried again with a sample size of 5000. This relative probabilities of 0.0906 for 170°-190° and 0.0682 for 90°-110° respectively. A very small change, which we can say that the number didn't change much even if we double the particles.

Finally, the histogram outputted with a sample size of 5000 produced the following graph:

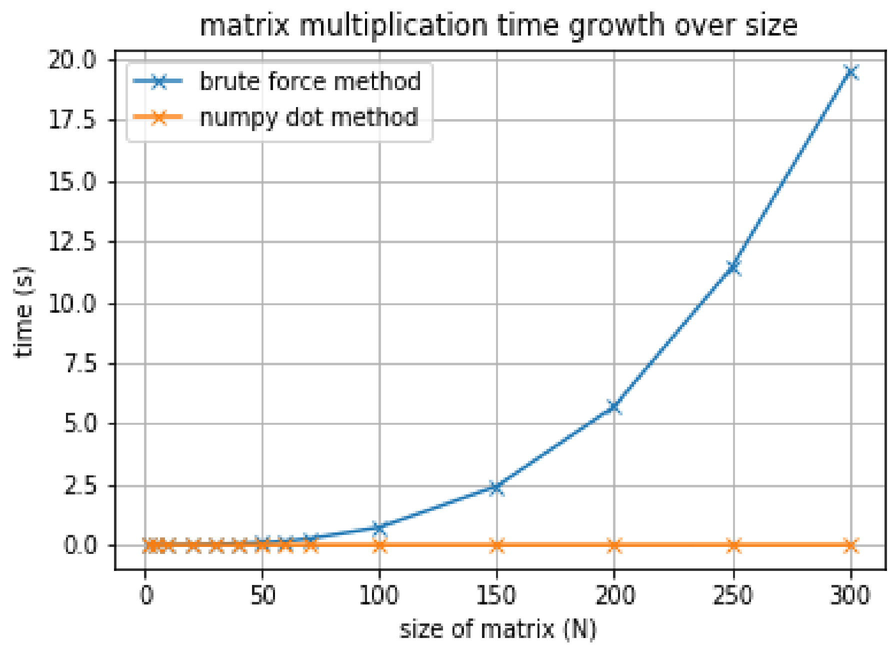


With the x-axis representing the different angle options between 0 and 360 and the y axis giving the standard count of each event. We used a bin number of 18 to allow for each bar to represent a range of 20 degrees to help reflect the previous ranges of interest. The histogram is quite normally distributed, which corroborates our previous results. This suggests that the most likely angle of reflection is around 180 degrees, with other probabilities getting less and less likely the further you stray z from the origin in either direction. This makes sense, as closer to the origin a circle is almost perpendicular to the incoming direction of the particles travel, while being more and more parallel as you stray away from it. This gives more opportunities to reflect off of the more perpendicular side of the circle, and less as it gets more parallel, which would put the average around the origin (180 degrees).

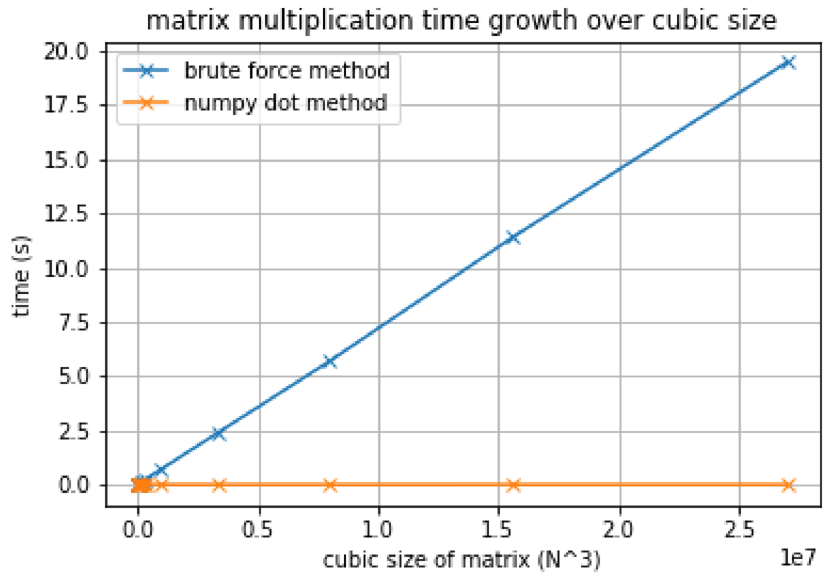
Question 3:

The goal of this question is to create constant matrices of varying dimensions on python, multiply them together, and explore how the size of the matrices we use effects the amount of time it takes for a computer to run the program.

Using a similar structure to the example outlined in *Computational Physics* by Mark Newman, example 4.3, we produced two matrices as well as an array containing various matrix sizes we wanted to try out ranging from 2 to 200. The matrices A and B were made to be constant matrices with each entry equalling 3. The program would take the time at the start of the session, multiply together the matrices, and then output the time at the end, and then take the difference of the start and end time to provide us with how long the actual processes took. We then repeated the process using a slightly different "dot" method using the numpy.dot function, and graphed each to compare the results.



The "brute force" method we used at first originally took same amount of time as the dot method, however it kept increasing in time exponentially as we increased the size of our matrix. The dot method however, stayed relatively consistent regarding the time it took to complete the operation with respect to matrix size. Finally, we reanalyzed the graph by plotting N^3 on the x-axis instead, this produced the following plot.



This time, the brute force trend is completely linear while the dot trend remains consistently low. This adjustment likely means that the ratio of time to matrix size grows something like x^3, and so adjusting the scale of the x-axis helped straighten out the plot.

Therefore, we can conclude that the numpy dot method is much more time efficient than the brute force method when the size of input becomes much bigger.