Simulating Gravitational Interactions for an N-body Interaction: Select Solar Bodies and Known Steady-State 3-Body solutions

Student Anthony Gerg
Promotor Dr. John Donor
Class Computational Physics

Intro and Discussion of the Problem

In Computational Physics, we have discussed many different computational methodologies to approach many different problems. To myself, the most applicable of these would be numerical differential equation integration. While other methods are discussed, the most covered methodology for approximating these differential equations is Runge-Kutta. Runge-Kutta is a technique used to solve and approximate these equations as mentioned above[1]. While a simple method to approximate, Runge-Kutta is noteworthy for being easy to implement computationally. Therefore, it makes sense that it would be integral to our final project.

The final project was simple enough, approximate Newtonian physics with a noteworthy problem, an n-body simulation. The N-body simulation is notorious for being chaotic and difficult to approach analytically. Therefore, much of the work for an n-body problem has been numerical approximations. It is easy to see where our problem connects with our methodology. By defining basic differential equations with a solution to our acceleration being considered in the problem, it will allow for us to solve this problem numerically ourselves. An extra component of the problem is the problem to which we would be applying our methodologies: solar bodies. For this project, we will be considering the Sun, Mercury, and Venus (I do not understand why I chose these two bodies out of all the bodies I could have and regret my decisions in this manner), and finally Saturn. My own bad decisions may have caused issues which will be discussed later, the intricacies of the problem can further be expounded upon and the mathematical underpinning shall now be discussed.

Theory

Before we can even begin discussing implementation, we need to discuss the actual theory on how to get the differential equations needed. To start, our bodies will a few initial conditions to even begin the problem. Our bodies will all have mass (m), position (r), and velocity (v). Our positional values are based on a Cartesian coordinate system with the origin being the sun's initial starting position. Therefore, our radial component to whatever body we are considering will be based off of a two-dimensional plane. It is important to note that the factor of being two-dimensional, we are not considering changes in depth to our bodies. This is already a shortcoming in regards to our implementation. Our initial parameters are going to be coming from a three-dimensional vector obtained from NASA's Jet Propulsion Lab (JPL) Horizon program. This shortcoming will be discussed in greater detail later. Our radial component will then take the form of $r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where our x_1, y_1 and x_2, y_2 are the positions in space. With at least this being defined, we can now apply our gravitational force being felt on object one, being implemented as

$$|F| = G\frac{m_1 m_2}{r^2}. \tag{1}$$

This is only our magnitude of our force, however. We have to also apply direction to our force to have our simulation be accurate. This is simple enough, applying the positional unit vector $\hat{r}$ and sticking it into our equation will yield our same initial equation in vector form. Further, our unit vector can be decomposed into $\hat{r} = \frac{r}{|r|}$. This gives us all the necessary tools needed to obtain our accelerations in both of our dimensions.

By taking our vectorized form of gravitational force, it is possible to then decomposed to our equation further using trigonometry. Our dimensional forces can be written as $F_x = F_{total} sin(\theta)$ and $F_y = F_{total} cos(\theta)$ and our $sin(\theta) = \frac{\Delta x}{r}$ and $cos(\theta) = \frac{\Delta y}{r}$ our final equations are

$$F_x = G\frac{m_1 m_2 \Delta x}{r^3}, F_y = G\frac{m_1 m_2 \Delta y}{r^3}. \tag{2}$$

By dividing this equation by the mass being considered, we can find our acceleration. Due to Newton's second law $F = m_1 a$ the final equation for acceleration is.

$$a_x = G\frac{m_2 \Delta x}{r^3}, a_y = G\frac{m_2 \Delta y}{r^3}. \tag{3}$$

Since we now have a method for finding our acceleration at a particular position, we can discuss our differential equations.

For our equations, we will be analyzing and relying on being accurate, they are thankfully relatively simple. We can define our equations purely based off the definitions of velocity and acceleration: $\frac{dx}{dt} = v$ and $\frac{d^2x}{dt^2} = \frac{d}{dt}(\frac{dx}{dt}) = \frac{dv}{dt} = a$. These are the only two equations we will be considering for our differential equations. In reality, our actual equations were considering is $x_{i+1} = x_i + v * \Delta t$. This relates to our equation by $x_{i+1} - x_i = v * (t_{i+1} - t_i); dx = v * dt; \frac{dx}{dt} = v$. The acceleration relation is the same way. A nice part of our differentials is that we do not have to worry about updating based on our time steps, and the differential already accounts for our time step. With this, we have defined the physics we will be using in our simulation. We can now move onto our computational points of emphasis.

Computational

The computational components of this problem were not hard, but rather had to be implemented meticulously. This section will not discuss the pitfalls that were made, but rather the pure computational basis that the final project ended with. To being, the differential equations and the acceleration due to gravity were both described as functions in the code. Since both would need to be called many times, the easiest implementation of them is functions. For the gravity function, it was fed in the masses and positions of the two bodies being considered and returned a numpy array of the components of the acceleration (Equation 3). For the differential equations are fed positions (they are not used in the diff eq but are fed in because of the format of our Runge-Kutta), velocities, and accelerations. The function returns a numpy array of the change in x, y, v_x, and v_y.

Besides what has already been defined, another massive choice in the creation of the simulation was to forgo using class definitions and only using numpy arrays. My reasoning for this is the following: "classes are okay when they work and the worst creation ever when they don't work"-Jonah Otto. Using classes and trying to use their strengths resulted in massive amounts of issues with the code. Therefore after a few hours of trying to adjust to the classes more, the decision to remove the classes left and instead using numpy arrays. While not being able to update our velocities and positions as well, arrays made it easier for myself to make adjustments later on due to the familiarity with array formats.

The majority of the challenge was implementing the Runge-Kutta. The main issues with the Runge-Kutta is that usually we are only accounting for one object, not four. As well, inside our Runge-Kutta, the forces are depending on accurate changes to positions and velocities. Therefore, adjustments need to be made to a nominal Runge-Kutta methodology. With our array set up, we can initialize multiple arrays with shape 4 to feed in. In the Runge-Kutta, we first initialize the time-frame (via for loop). This essentially stops our system, allowing us to find forces at this time point. We initialize our first body to look at (for loop). Then, while passing through all the other bodies calculating the accelerations via summation. After finding the accelerations, the actual Runge-Kutta begins. Because our accelerations were found before the Runge-Kutta, they were fed in as keyword arguments. The Runge-Kutta looks like

$$q_{i+1} = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4}$$

with the coefficients as $k_1 = f(q, t)$, $k_2 = k_3 = f(q + \frac{k_{i-1}}{2}, t + \frac{h}{2})$, and $k_4 = f(q + k_3, t + h)$. In these coefficients q here is the array of position and velocity feeding into the differential array. This is also what we adjust after getting our changes in positions and velocities. That is the reason I am feeding in the positions into my differential equation even though I do not need it, just to keep my q vector the same shape as my returned array. After that, we append to our positional and velocity arrays. These are different than our initial parameters, but rather track all the positions in total. The initial array fed in is account for the 'present' positions and velocities, and those other arrays are keeping track of all the positions and velocities that bodies ever go through. The new time point ticks over, and the process repeats. It is relatively simple, but one mistake can lead to cascading issues that won't reveal themselves easily. The Runge-Kutta was not apart of anything function, but rather initializes upon running. This isn't due to any reasoning, it is purely how I built the algorithm. If I was to use this code often enough, I would create a function for the Runge-Kutta.

With the description of algorithm done, I should report on how I recorded gifs as well. This plotting was done in matplotlib, and the animations with matplotlib.animation.FuncAnimation. At this point in time, all the points

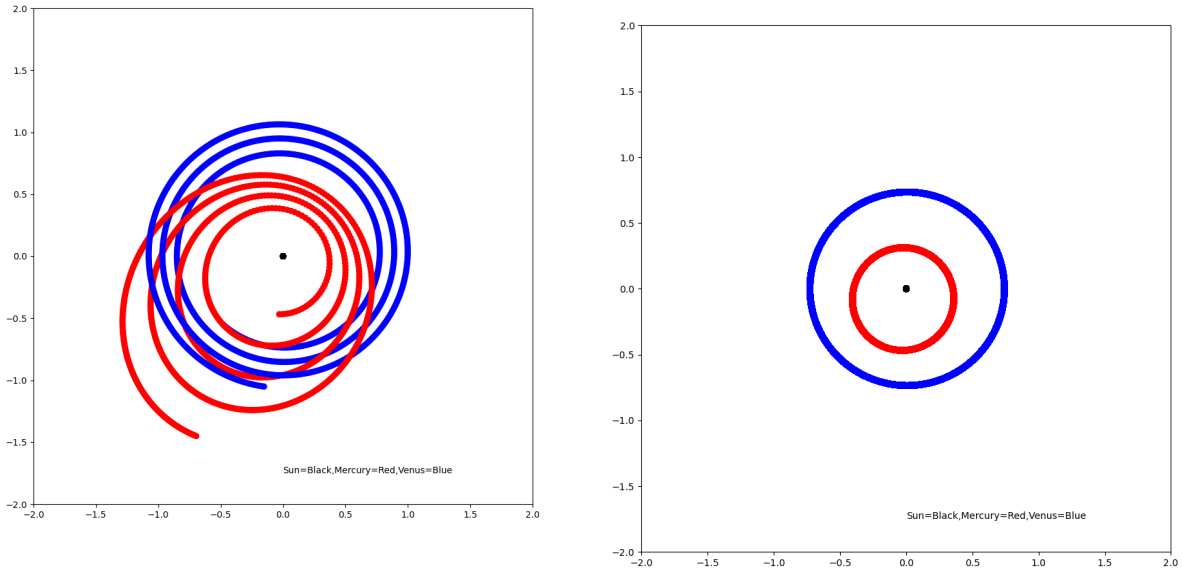

Figure 1: Sun, Mercury, and Venus after 1000 days. Different time steps of 1 day (left) and 0.001 days (right). The perihelion of Mercury in the larger time step becomes larger than Venus's regular orbit in the larger time steps

and the algorithm was done running. Therefore the function I used was simply for plotting, nothing extra. This function for plotting is then sent to a wrapping function using Pillow to get gif files.

Problems and Results for Part 1

Problems did arise, yet they can be easily explained to the numerical approach to the problem. Notably, this system is not a steady-state solution. As discussed earlier, the initial velocities and positions for the solar bodies were in relation to all 8 bodies. In our equation, we are only considering the interaction between the four bodies. Therefore our initial values are not proper solutions to steady-state, and our system will eventually collapse. This can be seen in the interaction with Mercury and Venus. When running with JPL Horizon initial conditions, Mercury cannot maintain it's proper orbit due to no other forces pulling it away besides Venus with a large time step. This leads to eventual degradation of the orbit of Mercury, only really good for roughly 10,000 years before Mercury ends up somewhere between Mercury and Saturn due to the Venus and Mercury getting too close to each other. To remediate the issue, a softening parameter was added to the positional vector, but it ended up causing errors to the other orbits as well.

Another problem was with the conservation of energy. Energy conservation is a large component of this problem, but my algorithm was leaking energy at each time step. While not ideal, this is due to our time not being a continuous function. If the function was running at infinitesimal time steps, no energy would leak. But I am running my time steps in days usually, and so the curve is not going to be as accurate as possible. On the daytime frame, the equation leaks at roughly 5e-5 every time step. While not huge on a day basis, the leaking energy cascades to a change of roughly 0.02 after 1000 days. The leak in energy is not as bad as one would think, and surprisingly, after 1000 days, it is sinusoidal, meaning that our average change in energy is 0, which is a good sign. The leak is also present due to the initial conditions, as it will dip after a time.

This leads to the results of part 1, our four bodies, while not being perfectly stable with large intervals, have orbits that work. When increasing the time step you get steady solutions, but increase the computational intensity of the problem by scales of magnitude. Most of the computational time was with creating images as matplotlib is not great with large data sets. Dealing with Mercury and Venus' interaction was not fun, but Mercury largely remained in orbit around the sun quite well with proper time steps. The perihelion of Mercury ends up farther out than Venus, but continues to orbit the sun. Many of the results can be seen in Figures 1 and 2

Saturn's orbit was unsurprisingly unaffected by the initial conditions. This makes sense in regards to how IVP work, the less likely to perturbation a body is, the more stable it will be to changes in velocity and acceleration. This was nice to see compared to the anxiety Mercury and Venus caused. Other than that, Saturn's orbit was largely uneventful.

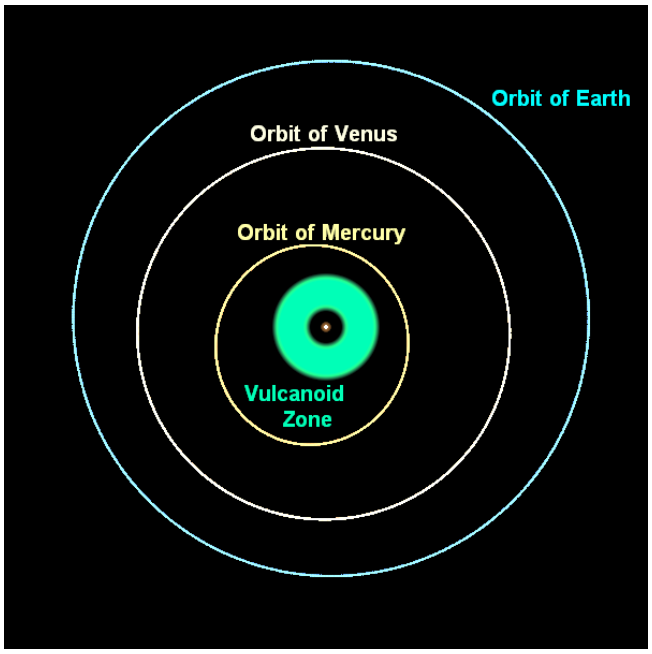

Figure 2: Accurate image of mercury's orbit. We can see the eccentricity is close to what we observe in our own simulation. From [2]

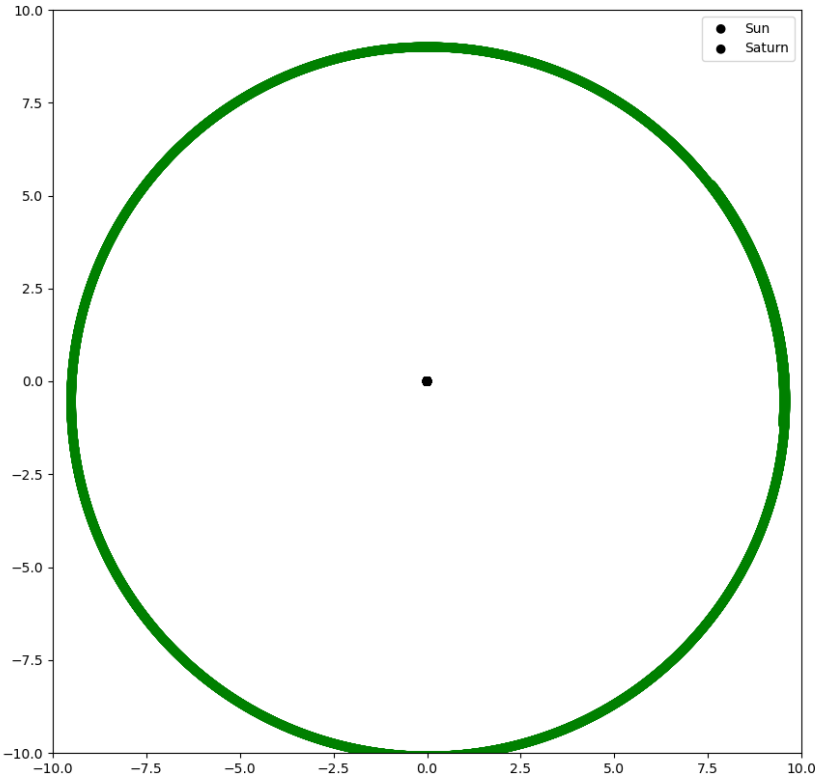

Figure 3: Saturn and Sun after 12000 days (1 day time step). Saturn remains largely normal with it's orbit, leading to confirmation that the code is working properly.

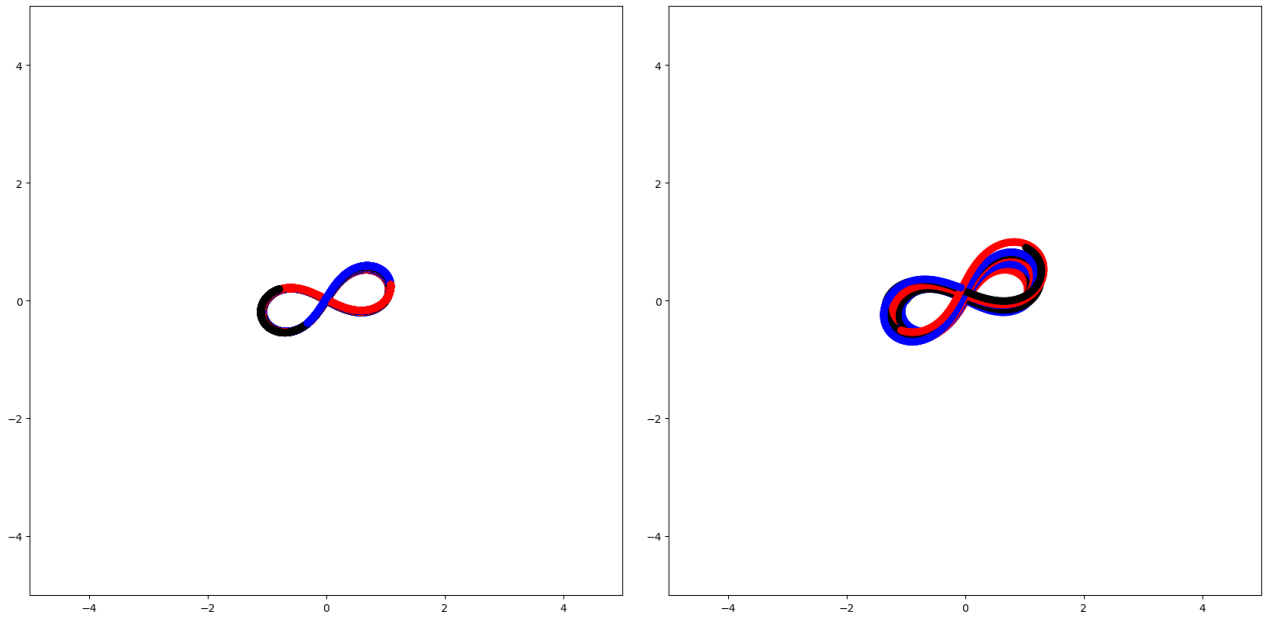

Figure 4: 3-body simulation over 200 seconds, with time steps 0.002 seconds (left) and 0.2 seconds (right). We can see the difference in the perturbation due to discrete time

Results for 3-Body Problem (Part 2)

Using the same methodology that was defined above, we now approach a steady state solution. I decided to investigate the figure eight solution to the three-body problem. My reasoning for this is it is the most noteworthy of them all, and has the easiest initial conditions to find. The initial parameters were found from a website that keeps track of these solution [3]. A steady-state three body is defined by the repetition of the processes being seen. And it is true, we do not see any type of issues with the steady state at large time steps, only slightly skewing is still relatively good. Again, when investigating energy there is loss present, but this is due to the time steps and a non-continuous time series. It is worse in this problem purely do the system being very picky about the solutions. The three-body is also proof that the closer to continuous time you get, the better approximation you obtain. This is shown in the photos given for the 3-body problem. This extreme sensitivity is interesting to consider when looking at n-body. The solution you find online for n-body are usually designed theoretically, and computationally only work with extremely small time steps. It was interesting investigating this phenomena and see how computational methodologies can change the problems at hand.

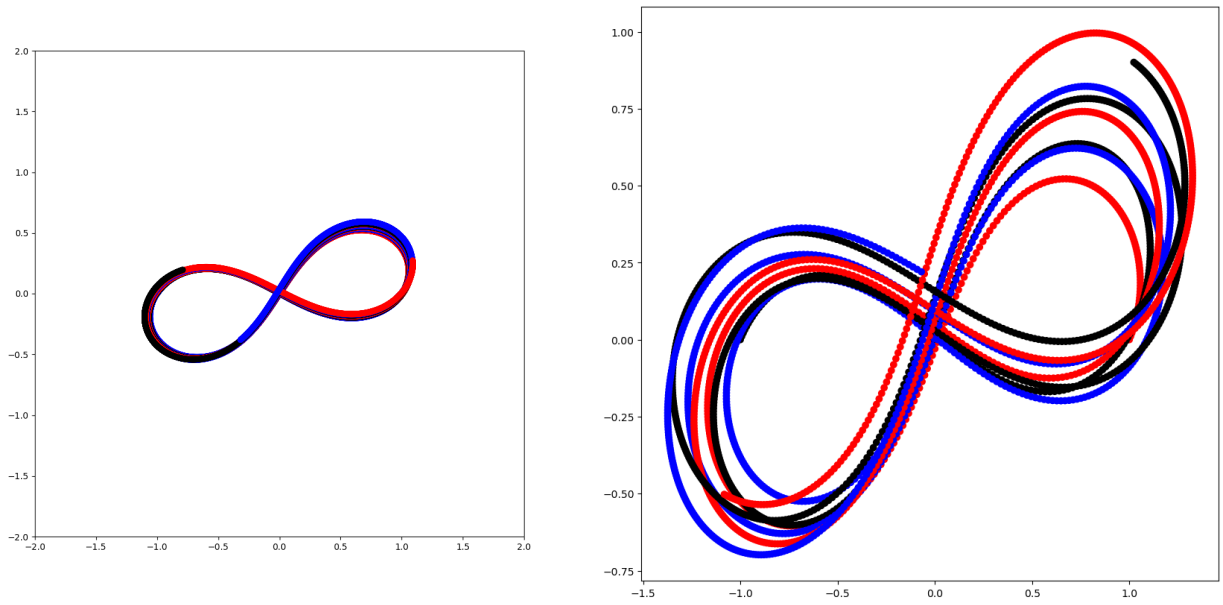

Figure 5: A close up of Fig 4. The differences are much more present and the skewing due to the large time steps are evident (right).

References

[1] K. Kaw, Nyugen, Textbook: Numerical Methods with Applications, 2008.
URL https://nm.mathforcollege.com/textbook-numerical-methods-with-applications/

[2] Exactly how elliptical is mercury's orbit, visually, without exaggeration?
URL https://astronomy.stackexchange.com/questions/19126/exactly-how-elliptical-is-mercurys-orbit-vis

[3] Three-body gallery.
URL http://three-body.ipb.ac.rs/