Orbital Body Simulator

R. Joey Rupert[1], Pridhvi Myneni[2], and Nicholas Dodd[3]

[1-3]Stevens Institute of Technology

CPE 593-A: Applied Data Structures & Algorithms

Fall 2020

Abstract

An orbital body simulator is a program that is able to predict the trajectory of bodies through space based on the equations of gravity. Using partial differential equation solvers, it is possible to more efficiently calculate these trajectories with greater accuracy than possible using a simple linear system. Using the Runge-Kutta-Fehlberg method, as well as Predictor-Corrector using the Adams-Moulton methods, we attempt to match the predictions made by the JPL Horizons team.

Contents

1 Introduction

1.1 Purpose and Uses of Orbital Body Simulator

Orbital Body Simulators are used for many purposes, from basic satellite tracking (with air-resistance losses & solar pressure) to planning spacecraft trajectories, to planetary defense from large bodies.

With the amount of space junk increasing in recent years owing to failed launches, decommissioned satellites from as far back as the soviet-era[1], and the increasing number of satellites being sent up into orbit (including hundreds of networking satellites[2]), we are relying heavily on these programs to predict the paths of these satellites and plan for orbital maneuvers to avoid collisions, which can damage equipment and create further debris.

Although it has been some time since we have launched a long-range spacecraft, these programs are used to calculate the predicted paths of space craft to ensure they arrive at the correct planets, at the correct times, with the correct velocities. Indeed, the Voyager Missions are known for using gravitational assists to change trajectory via Jupiter and Saturn.[3]

As it has been a long time since the start of the universe, the majority of large objects that are capable of causing significant changes/damage to our planet & climate are in stable orbits. However, there are still such objects in orbits that may intersect with the Earth in the future. Thus, although most smaller asteroids burn up in the atmosphere, there are larger asteroids (diameter > 1 km), which have completely missed the Earth for hundreds of thousands of years. Perhaps the most pressing use of an orbital body simulator is its ability to track potentially dangerous asteroids, allowing us to prepare for and perhaps stop a future collision. Indeed, on April 29, 2020, a 2km wide asteroid, `1998 OR2`, passed by Earth only 6.3 million km away, close for an asteroid of this size. In a situation where a large asteroid was on a collision course with Earth, knowing when and where that asteroid will be before is comes is crucial in mounting any defenses against it.[4]

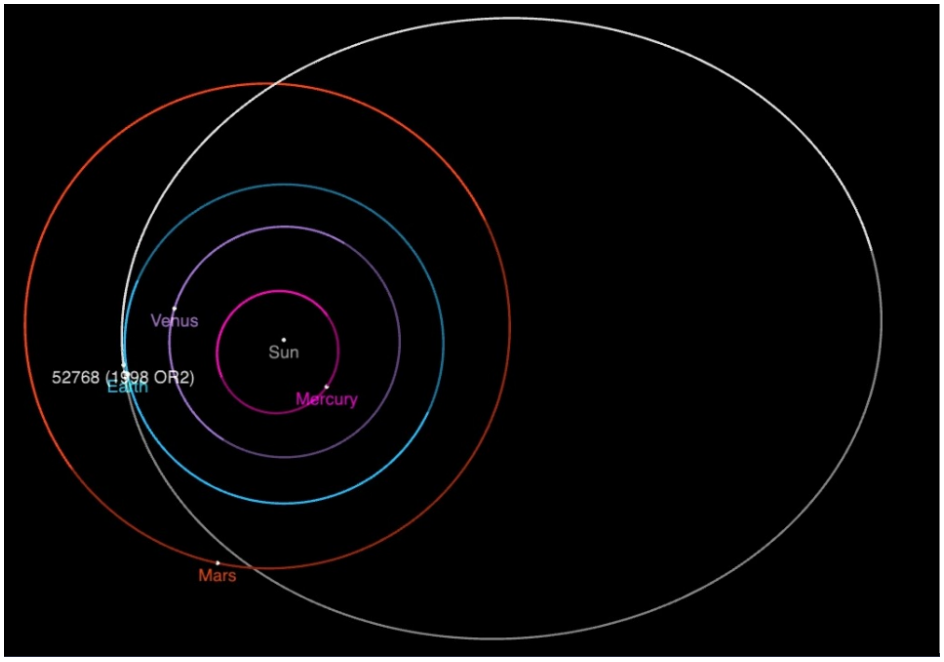

Figure 1: Orbit of 1998 OR2

A final major use of orbital body simulators is to track harder-to-see objects' trajectories. By checking expected positions of observable objects, astronomers are able to discern the presence of undetected objects, including black holes and other large-mass objects [5]. Additionally, planets that are difficult to detect using

conventional methods such as radiation can be found using orbital body simulators to track the deviation between expected and actual orbits [6].

1.2 Approaches to Creating an Orbital Body Simulator

A 1999 paper [7] takes a look at several approaches to creating an orbital body simulator. The paper discusses four different single-step algorithms and three predictor-corrector methods. The single-step method algorithms that were tested showed that in order from least to most accurate were Euler, first-order; Heun, second-order; Runge-Kutta (RK4), fourth-order, and Runge-Kutta-Fehlberg 4, 5 (RKF45), which is fundamentally a fourth-order algorithm. However, the higher-accuracy algorithms are also more computationally intensive.

The predictor-corrector methods are multi-step, requiring the calculation of values, followed by a correction to be applied based on the prediction. The predictor-corrector methods require the use of RK45 or RKF45 for calculating the first four time steps or four previously recorded timesteps in order to predict future timesteps. The three methods looked at in the paper are Hamming, Adams-Moulton, and Milne-Simpson, in order of decreasing efficiency and increasing accuracy. Using the derivative of the target variable from the past four timesteps, the current value of the variable can be both estimated and corrected.

2 Method

2.1 Single-Step Method

The single-step method we chose to implement was the Runge-Kutta-Felberg method, better known as RKF45. It's an extension of the fourth-order Runge-Kutta (RK4) integral approximation method. RK4 approximates an integral, using the derivative at multiple points between timesteps, and initial value of the function at a certain time. RKF45 does the same, but with an optional fifth-order implementation, allowing for an error estimate and the calculation of an adaptive timestep, allowing for high efficiencies in simulations while guaranteeing a tolerance. For our primary simulation, we used RK4 to calculate the initial set of four values, then switched to Predictor-Correcter (expanded on in the next section). However, we did also create a simulator that used exclusively RKF4, by calculating the accelerations, then using the initial values for position and velocity to integrate. The equations for RKF45 are shown below:

2.1.1 RKF45 Equations

$$y_{n+1} = y_n + (256/216) * k_1 + (1408/2565) * k_3 + (2197/4104) * k_4 - (1/5) * k_5 \tag{1}$$

$$z_{n+1} = y_n + (16/135) * k_1 + (6656/12825) * k_3 + (28561/56430) * k_4 - (9/50) * k_5 + (22/55) * k_6 \tag{2}$$

$$k_1 = h * f(t_n, y_n) \tag{3}$$

$$k_2 = h * f(t_n + (1/4) * h, y_n + (1/4) * k_1) \tag{4}$$

$$k_3 = h * f(t_n + (3/8) * h, y_n + (3/32) * k_1 + (9/32) * k_2) \tag{5}$$

$$k_4 = h * f(t_n + (12/13) * h, y_n + (1932/2197) * k_1 - (7200/2197) * k_2 + (7296/2197) * k_3) \tag{6}$$

$$k_5 = h * f(t_n + h, y_n + (439/216) * k_1 - 8 * k_2 + (3680/513) * k_3 - (845/4104) * k_4) \tag{7}$$

$$k_6 = h * f(t_n + (1/2) * h, y_n - (8/27) * k_1 + 2 * k_2 - (3544/2565) * k_3 + (18959/4104) * k_4 - (11/40) * k_5) \tag{8}$$

A variable timestep can also be used, with the fifth-order variant of the estimate. The optimal step size is s*h, where:

$$s = ((T * h)/(2 * |z_{n+1} - y_{n+1}|))^{1/4} \tag{9}$$

T represents the tolerance for error.

2.2 Predictor-Corrector Method

The predictor-corrector algorithm is a form of numerical analysis to integrate differential equations. Each algorithm follows a two step process: prediction and correction. The prediction step uses preceding derivative values to extrapolate the next value. That predicted value is then used in the corrector, along with preceding derivatives to calculate the value of the same variable with greater accuracy[8].

The predictor-corrector method employed by the simulator is Adams-Moulton. The formulas for the method are shown below where the predicted step: p_{n+1}, is the predicted value based on the derivatives of the current and last three steps:$f_n, f_{n-1}, f_{n-2}, f_{n-3}$, the step increment h, and previous step's corrected value y_n. That predicted value is then used to determine the derivative of the future f_{n+1} by taking the derivative of p_{n+1} at time t_{n+1}. Following, f_{n+1} is used in the corrector formula to determine the corrected value y_{n+1}[7][8].

$$p_{n+1} = y_n + h/24 * (-9 * f_{n-3} + 37 * f_{n-2}, -59 * f_{n-1} + 55 * f_n) \tag{10}$$

$$f_{n+1} = f(t_{n+1}, p_{n+1}) \tag{11}$$

$$y_{n+1} = y_n + h/24 * (f_{n-2} - 5 * f_{n-1}, +19 * f_n + 9 * f_{n+1}) \tag{12}$$

The implementation of predictor-corrector follows these four steps:

1. Predict the position in the prediction step.
2. Use the gravitational force equation to calculate acceleration for the next step ($F_G = \frac{G*m_1*m_2}{r^2}$).
3. Use correction to calculate the velocity using acceleration.
4. Use correction to calculate the position using velocity from the previous step

The first step runs the prediction equation by supplying the current position, current and three previous timesteps' velocities, and the time step, which is constant throughout the whole simulation. It should be noted that the initial four steps are calculated without the predictor-corrector, either requiring real-world data, data from single-step simulations, or data from multi-step processes using the previous two avenues. The second step uses the aforementioned equation to calculate the acceleration of each body using the predicted position from the first step. Since force due to gravity is the same on every pair of objects, we are able to make this step (most computationally intensive) an $O(\frac{n(n-1)}{2}) \approx O(n^2))$ process. In the third step, velocity is calculated from the previous velocity, the acceleration just calculated, and the cached accelerations from the last three time steps. Finally, in the fourth step, the position is corrected from the predicted current position, and the velocities from the current and previous four steps.

The process described above runs for each physical dimension (X, Y, Z), for each planet. This is a very computationally intense algorithm, but its accuracy compared to the Milne-Simpson method is greater and its computation speed is faster than the Hamming method. Additionally, the higher accuracy provided by this algorithm allows for larger timesteps with less loss in accuracy.

3 Implementation

3.1 Input Data Acquisition

A large problem in space is that it is very difficult to create reference points. Indeed, even the sun is known to have its own small orbit. To solve this problem, we use a barycenter, defined as the position of the center-of-mass (CoM) of a system because although orbital bodies are constantly in motion, without collisions or unconsidered bodies exerting influences on the system, the barycenter of a system is unmoving.

To find actual planetary positions described in reference to our solar system's barycenter, we used the NASA Jet Propulsion Laboratory's HORIZONS program, which has orbital data available for download via HTML and TELNET interfaces.

3.1.1 Data Processing

As the output data files failed to follow a standard format, a fair amount of data processing was required to put it into (1) SI units and (2) into a standard format [9] that could be more easily digested into a C++ program. For its expanded number of libraries and easier error-handling, Python was used for this step. This was achieved using regular expressions, followed by multiplying by conversion factors. From there, we dumped each planet's data into a save file, to allow changes to the script to process different files (again owing to non-standardization of input data). Finally, the save files were used to generate the input file of the format [9].

This format and system of units was vital and led to our originally failing to run successful simulations owing to a mix of astronomical lengths, masses, and constants meant for different units. By converting our data into a standard, agreed-upon format, we were able to fix many of our original problems.

3.2 Data Handling & Simulation Support

As the simulation was designed to be dynamic and required a significant amount of data that needed to be loaded and stored over the simulation in an efficient, yet reliable manner, we had to write and test our own structures. We used the following information-storage classes to facilitate this process and enable unit-testing.

`Planetary_Object`: A collection of static variables (global simulation parameters), as well as individual planets' information, such as name, position, etc.

`Matrix`: A regular matrix used to store position, velocity, and acceleration data over time.

With these basic datatypes, we were completely able to isolate the mathematical side from the details of the planetary data and their implementation.

Finally, some key decisions we made included using `uint64_t` for timestamps and timesteps, with the underlying unit of time being seconds. Although this might seem granular enough for planetary simulations, the RKF45 algorithm actually requires double-precision for both adaptive timestepping and regular timestepping as it attempts to find derivatives of the target function at intervals between the current and next timestamps. On the other hand, there are issues with precision that need to be considered using doubles for timestamps as roundoff error from small timesteps, as well as accumulated error can play a large role in determining the accuracy of the simulation. Thus, based on what we learned during this implementation, we would recommend using double-precision types for both timestamps and timesteps, but recording the number of timesteps crossed using unsigned integer types to eliminate accumulated timestepping error using the following formula.

$$\text{num\_timesteps\_passed} * \text{len\_timestep} + \text{start\_timestamp} = \text{curr\_timestamp}$$

3.3 Modified Implementation

As we were having difficulties with both modes in our primary implementation, we suspected that there was an issue in the RKF45 implementation that could be causing issues, as both modes relied on the RKF45 implementation to either start or entirely run the simulation. Thus, after verifying that the data input-output steps were indeed correct, we created a secondary implementation, viewable on our `pconly` branch[10], consisting of a Predictor-Corrector implementation that relied on JPL's data.

The data format, albeit undocumented, just required 5 timesteps of position data followed by 5 timesteps of velocity data in place of the single timestep of data in the documented model. Additionally, the units of the timestamp and timestep changed to days; however, to accommodate the updated format, the underlying data type was changed to double precision. Finally, JPL HORIZON's data was used to back-fill the initial 4 steps of time data.

3.4 Data Verification & Visualization

With limited time, Python was used to perform data verification and visualization to help determine the sanity of the simulation results. Using Pythagorean distance, we were able to determine the distance between each planet's ending position and it's expected location. From there, we charted the locations of the planets using the Matplotlib project[11], which allowed for 3-dimensional plotting and dynamic viewing, with rotation and zoom features included.

4 Results

4.1 Visualization

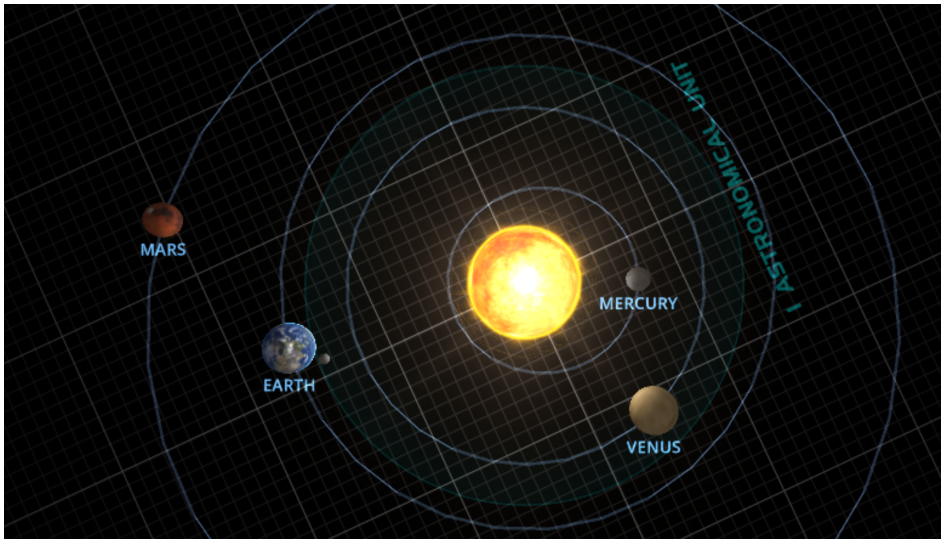

Figure 2: Inner planets as predicted by NASA.

Figure 3: Calculated positions after 1 day of simulation (same timestamp for both screenshots). Similar view with different angle to the X-Y plane. Blue point is the Earth, yellow point is the Sun.

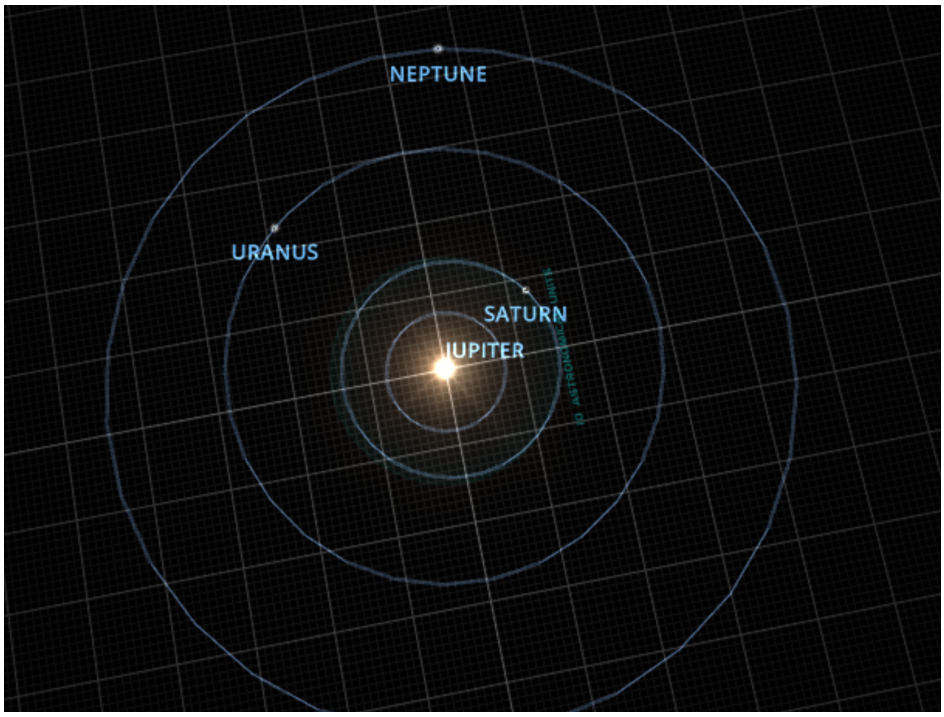

Figure 4: Solar system as predicted by NASA.

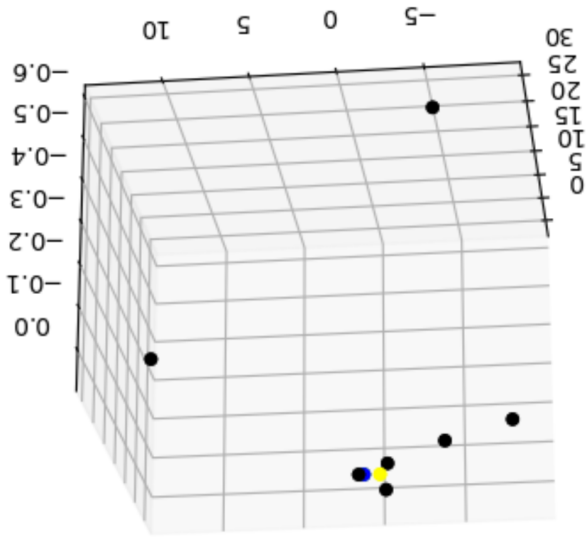

Figure 5: Calculated positions after 1 day of simulation (same timestamp for both screenshots). Similar view with slightly different rotation and angle to the X-Y plane. Blue point is the Earth, yellow point is the Sun.

4.2 Interpreting Output Data

The tolerances of our simulation were of the order of millions of meters for the sun and billions of meters for the outer planets. In attempting to improve our error, we reduced the timestep to fractions of a second;

however, this tolerance did not change. We will further discuss the potential sources of error and how they will be addressed in future revisions of the system.

4.3 Accuracy of Results

The accuracy is not adequate. Being able to lose an order of thousands to millions of kilometers for planetary bodies that are tens of thousands of kilometers wide over the course of a single day is simply unacceptable error tolerances. Attempting to use this simulation for the prediction of solar-system wide launches, let alone satellite trajectories, would be impossible with these error estimates.

5 Conclusion

The algorithms considered offer significantly more accuracy than was found through our simulation. Some possible sources of error we believe may be in the system include the following.

- P-C only fails to have accurate predictions because:
  a) Decreasing the length of each timestep is an inadequate way to increase accuracy given that the input data's timesteps are unchanged, making the derivatives used by P-C be farther apart than intended, creating the illusion of a faster moving body than is actually present.
  b) The timescales for which we predict are too small to reliably measure the accuracy of the system. Given more time, we should attempt to predict the trajectory of the system over the course of multiple years and compare with observed data (ie pick a starting point in 2000 and see how closely we can match 2020 observations).
- RKF45 has problems either in its implementation or in the mathematical logic behind our implementation owing to the fact that every timestep of RKF45 seems to increase the order of magnitude of the coordinates by one order per timestep. We have yet to find these errors.

5.1 Ways to Improve in The Future

In addition to fixing the aforementioned problems, we hope to include the following improvements in our future releases:

- Implement the adaptive timestep to take advantage of the fifth order estimation in the RKF algorithm.
- Implement calculations for solar pressure and air resistance for bodies given atmospheric observations of the planets (perhaps only the earth).
- Add a rocket-simulation feature that allows for estimating the trajectory of rockets launched from earth-orbit at different points in time.
- Pull in data for satellites to allow their simulation for additional system complexity and verification.

6 Code

Our code can be found here[12].

6.1 Dependencies

6.1.1 Python

We require the use of Python 3.x [13]. We also require the use of the following Python libraries: Loguru[14], TQDM[15], and Matplotlib[11]. These libraries can be installed using
`pip3 install loguru tqdm matplotlib`

6.1.2 C++

The latest version of C++ should be preferred owing to the (mostly) backwards-compatible nature of most implementations of the C++ compiler; however, we believe that C++11 should work.

6.1.3 Make

We use a Makefile to build our code, so if your system does not have make, please install it.

6.2 Preparing Input Data

For convenience, we have uploaded the (relatively small) source files directly from the JPL Horizons project to allow for duplication of results. We should note that one of the planets' files was missing a piece of information that we had to look up, meaning that one of the files was modified.

```
cd src/data_process/ && python3 parse_horizons.py && cd -
```

If you would like to procure your own data, please ensure you download the HORIZONS data using the instructions specified in the comments in the `parse_horizons.py` file, or write your own parser such that the input data format[9] is followed.

6.3 Compiling C++

We have moved everything into a Makefile. As we are not yet stable, the Makefile builds everything with the debugging flags enabled, which allows for debugging via GDB. To build, run `make -B`.

6.4 Running the Program

Please note that the path to the input data file is hard-coded relative to the project structure, requiring you to have changed directory into the build directory prior to running the program. If your program hangs without printing initial planetary positions, then there is likely a problem with how the program is reading in the input file. Please confirm that the input file exists, and is named as per the `#define`.

```
cd build && ./reader && cd -
```

References

[1] A. Witze, *The quest to conquer earth's space junk problem*, Sep. 2018. [Online]. Available: https://www.nature.com/articles/d41586-018-06170-1.

[2] J. Foust, *Spacex sets new falcon 9 reuse milestone on starlink launch*, Nov. 2020. [Online]. Available: https://spacenews.com/spacex-sets-new-falcon-9-reuse-milestone-on-starlink-launch/#:~:text=SpaceX%20has%20now%20launched%20955,northern%20U.S.%20and%20southern%20Canada..

[3] *Gravity assist*, Dec. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Gravity_assist#Timeline_of_notable_examples.

[4] T. Greicius, *Asteroid 1998 OR2 to safely fly past earth this week*, Apr. 2020. [Online]. Available: https://www.nasa.gov/feature/jpl/asteroid-1998-or2-to-safely-fly-past-earth-this-week.

[5] D. Clery, *Astronomers find closest black hole to earth, hiding in plain sight*, May 2020. [Online]. Available: https://www.sciencemag.org/news/2020/05/astronomers-find-closest-black-hole-earth-hiding-plain-sight.

[6] *What is a barycenter?* Jun. 2020. [Online]. Available: https://spaceplace.nasa.gov/barycenter/en/.

[7] R. Ellis, A. Perelson, and N. Weisse-Bernstein, "The Computation of the Gravitational Influences in an N-Body System," *International Amateur-Professional Photoelectric Photometry Communications*, vol. 75, p. 52, Mar. 1999.

[8] *Predictor–corrector method*, May 2020. [Online]. Available: https://en.wikipedia.org/wiki/Predictor%E2%80%93corrector_method.

[9] P. Myneni, *Orbital body simulator - data format*, Dec. 2020. [Online]. Available: https://docs.google.com/spreadsheets/d/10y0553157L4t-43h3ratr3Er87rziJQ7Q7Md4NY2gVQ/edit#gid=0.
[10] ——, *P-c only branch*. [Online]. Available: https://github.com/PMARINA/CPE593-FinalProject/tree/pconly.
[11] *Visualization with python*. [Online]. Available: https://matplotlib.org/.
[12] R. J. Rupert, P. Myneni, and N. Dodd, *Main branch*. [Online]. Available: https://github.com/PMARINA/CPE593-FinalProject.
[13] *Download python*. [Online]. Available: https://www.python.org/downloads/.
[14] *Loguru documentation*. [Online]. Available: https://loguru.readthedocs.io/en/stable/index.html.
[15] C. d. Costa-Luis, *Tqdm*. [Online]. Available: https://tqdm.github.io/.