ANALYSIS REPORT

Solar System Simulation with Ordinary Differential Equation Solvers

Overview

The task for this assignment is to build reusable code for incremental software development. The Ordinary Differential Equation (ODE) solvers implemented in Programming Assignment 4 (PA4) will form the base of the reusable code. In PA4, the ODE solvers were used to simulate circuits given their time-varying behavior. In this assignment, we will show how the solvers can be used to simulate any system whose behavior can be described with time-varying differential equations. Specifically, we will be using the ODE solvers to simulate astronomical bodies, such as the Solar System.

ODE Description

The force acting on each celestial body is given by a ratio of the gravity and distance of other bodies in the system:

$$\boldsymbol{F}_i = -G \sum_{i \neq j} \frac{m_j m_i (\boldsymbol{r}_i - \boldsymbol{r}_j)}{\left|\boldsymbol{r}_i - \boldsymbol{r}_j\right|^3}$$

where $\boldsymbol{F}_i$ denotes the force acting on the i-th body (of interest), m_i is the mass of the i-th body, $\boldsymbol{r}_i$ is the position of the i-th body and G is the gravitational constant. The acceleration of the i-th body is then obtained using Newton's Second Law of Motion:

$$\ddot{\boldsymbol{r}} = \boldsymbol{F}/m$$

Integrating the acceleration vector over time will yield the position of the body over the specified time interval. The goal of the software developed in this assignment would be to solve for the positions and velocities of n bodies for a user-specified duration, given their initial starting positions and velocities. The results will be validated through a simulation of the solar system.

Description of the Software Architecture

Software Functionality

The software was designed in an incremental manner. Functions to solve the ODEs and functions to describe the behavior of astronomical bodies were clearly delineated to maximize code reusability. The following section provides a brief rundown of the functionality of the code:

i) The files `celestial_body_functions.hpp` and `.cpp` define the behavior of the astronomical bodies. Users can define each body using an identifying name, its mass, and its

initial position and velocity (together defined as the "state" of the body). The user can then define the duration of the simulation and the time interval for each step in the simulation.

ii) The `files ode_solvers.hpp` and `.cpp` define the functionality of the ode solvers.

iii) The files `wrapper.hpp` and `.cpp` provide an interface between the celestial body functions and the ODE solvers. The `simulate_system()` function is the main function that will be used for obtaining the state of the new state of the system at each time step.

Code Reuse

A significant bulk of the functions in this assignment were modified or improved versions of the code used in previous assignments.

i) The utility functions are amassed from all the helper functions written from PA2 through PA4. They are functions to enable the programmer to perform easy manipulations (such as adding two vectors, or scaling a vector by a constant, etc.) and debugging (such as printing vectors and matrices in an easy to read format) without having to constantly define loops.

ii) The ODE solver functions are imported from PA4, with slight modifications. The first is the change in the function signature of the function pointer; an additional `vector<double>` was incorporated to allow the passing in of additional parameters to the system. It should be noted that the ODE function to be solved is still generic in nature, and does not require knowledge of system specific structures. Secondly, Heun based solvers were included in this implementation (the original implementation in PA4 only had Euler and RK34 methods).

Design of the ODE System

In PA4, the ODE solvers were designed to accept any function with the standard function signature of (`current time , vector of inputs , vector for outputs`). This allows the ODE to be implemented with ANY time varying function, regardless of what the physical system is. In this project, the function signature was modified slightly to accept an additional parameter variable without loss of generality, the new function signature is therefore (`current time , vector of parameters , vector of inputs , vector for outputs`).

In the astronomical system, the function for finding the acceleration of the bodies was designed to fit this standard function signature. The `find_state_dot` function accepts (`current time , vector of body masses (parameters) , vector of current states (inputs), vector for accelerations (outputs)`), which follows the function signature required by the ODE solvers. This was achieved by creating transformation functions that parsed and flattened

the user-input data into standard vectors. This allowed the ODE solvers to be operational without requiring knowledge of the `body{}` structure.

Additionally, the states of all the bodies in the system were also flattened out to the single concatenated vector (and not a vector of vectors) to reduce the looping time required. A single flat vector would be less computationally intensive to manipulate than a vector of vectors.

Software Usability

ODE Solvers

As mentioned above, the ODE solvers are designed to take in a pointer to any generic ode function with the standard function signature. Therefore this suite of ODE solvers can be used to simulate ANY system, from circuits, to financial market profiles, or in this case, astronomical bodies.

Dynamical Systems

While the `find_state_dot()` function specifically defines the behaviour of astronomical bodies, the rest of the system defines the generic behaviour of mechanical systems with mass, based on Newton's Second Law. Hence, if a user wishes to simulate a different dynamical system, he only needs to redefine how the forces are calculated in the `find_state_dot()` function. The software will be able to simulate the system to find the positions of the bodies at each time step without any additional modifications.

Astronomical Systems

For problem-specific implementations, the software will be able to handle any arbitrary astronomical system (e.g. binary star systems, or multi-body systems) that the user wishes to define. The implementation is not limited to the solar system shown in the results.

User Interface

The user interface works by prompting the user for an input data file path. The file is expected to be in `.txt` format. If the file path is correctly pointing to the location of the input data file, it will report that it has been successfully read, otherwise, the output will state that it was unsuccessful in reading the file, either because the file does not exist in the path or there is something wrong in the file.

Next the user interface asks the user to input the ODE solver method that the input planet system should be solved with. The user interface outputs the available options to the user to make the selection process quick and easy. The user interface provides a safety check that when an input method is given, it is the parsed to its representation as an int and then back to a string. If the output of the

method does not match, then you know the program misinterpreted the input, or the user input the name in incorrectly. Next the user must define the time duration and time step for the simulation. These values are also printed back out to the user so they can validate that their input matches the recorded values.

Finally, using the user-input planetary system and the ODE solver variables, it will begin the simulation. Before the simulation starts, it prompts the user to define an output file path to store the output log. The user provided the file path and the simulation runs. When the simulation is complete, it will repeat the output log file path to verify the file location to the user.

The safety checks are helpful for the user trust that the program is parsing their input information properly. Additionally, the input file path for planetary system also helps make the user interface easier and prevent less setup in code (which also cuts down on the repetitive read file function calls) needed for each simulation.

Testing and Safety Checks

The ODE tests are similar to the tests implemented for the Programming Assignment 4. This code is another example of code-reuse in this assignment. There were only slight modifications that needed to happen for these test functions to fit this assignment. The function signature defined in the function needed to be updated so that one of the previous doubles now was changed to another vector. Each ODE solver was tested utilizing the hacker practice exponential function and known true results. If the ODE solvers could not return values that were numerically equivalent to the true results, then the test would return false, otherwise it would return true.

The celestial body "body" and "system" structs were tested to ensure that their constructor and destructor functions are working properly. These tests were conducted by evaluating where or not the system could be used to calculate the value of a arbitrary test 3 planet system. The accelerations for the test 3 planet system were calculated by hand to get the true results. If the vector returns different results, the test would return false, and the user would know that the body or system structs are not working properly. The differing results cannot be caused by the ODE solvers, because those are tested in a separate function and it is seen where they are passing or not.

The wrapper tests are used to confirm that the input values for the ODE solver are being updated properly for the next time step. This means that the provided slope values from the ODE solver functions are correctly multiplied by the time step (march) and then added to the current input values. The result of the wrapper should provide the input values for the next time iteration of the ODE

solver. The test is conducted by implementing the Forward Euler method in matlab to do cross-validation with the updating input values calculated in the C++ implementation. If the values are not updating the same, then the test returns false.

The user interface does not have tests, but as explained above is implemented with safety checks that are provided back to the user to confirm the input was interpreted properly.

Results and Discussion

Inner Solar Orbital System

We first tested a 5 planetary system, with the Sun, Mercury, Venus, Earth, and Mars, which is referred to as the inner solar system. For all five planets to achieve a fully complete orbital, the time duration was set to 700 days with a 1 day time step. This inner solar system was simulated by all the ODE solvers. The results for each ODE solver method appeared to be what we would have expected.

The forward euler had the lowest orbital accuracy compared to the other four ODE solver methods, as seen in Figure 1. With the propagating error in each step of the Forward Euler ODE Solver, the orbital is unable to get close to passing its initial position at the start of the orbital. This is what results in the orbital looking like a spiral.

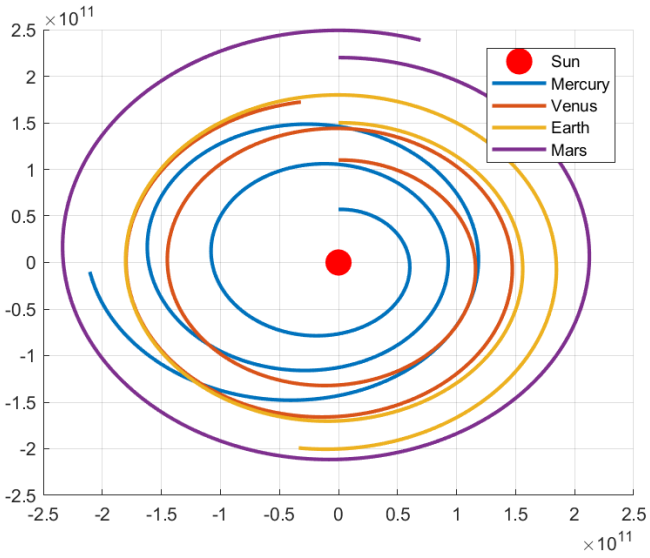

Figure 1: Inner Solar System Orbital Position Calculated by Forward Euler ODE Solver

The other four ODE methods had comparable results, where the orbit was able to return close to the initial position that was provided for the start of the orbital, as seen in Figure 2. The RK34A method seemed to have slightly more error propagation for the planets closer to the sun, as seen in Figure 3.

The growing size of the mercury dimension, could suggest that the hp adaptivity which alters the size of the march we take at each time step, could be influenced more by the larger orbitals in the system. Thus the march used to update the time step for RK34A is too large for estimating the smaller inner orbitals, thus creating large errors in its accuracy.

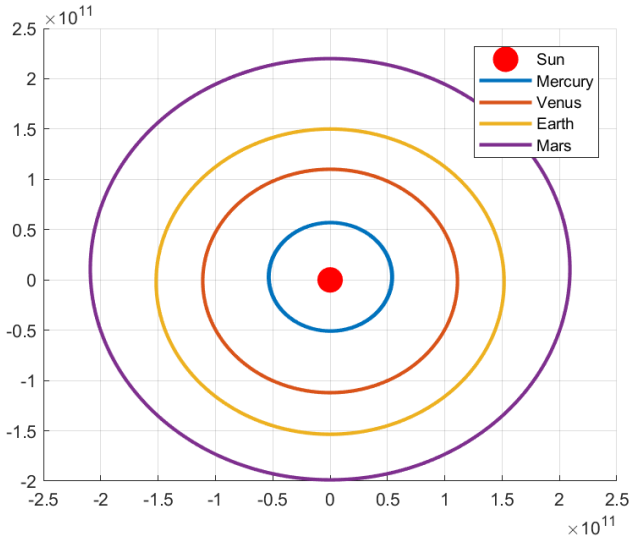

Figure 2: Inner Solar System Orbital Position Calculated by Heun Iterative ODE Solver

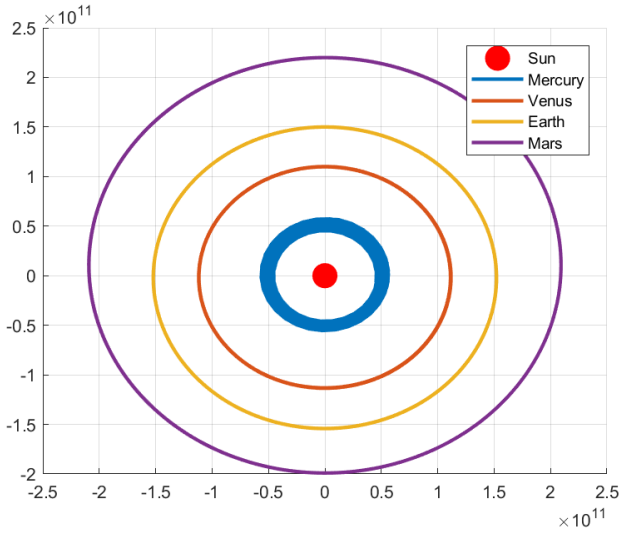

Figure 3: Inner Solar System Orbital Position Calculated by RK34A ODE Solver

Full Solar System Orbital Results

The next simulation test we look at was the full solar system, adding, Jupiter, Saturn, Uranus, Neptune, and Pluto. The result of this solar was performed incrementally to see how the inner system's orbitals were affected by the longer simulation duration needed for the outer planets to complete a full orbital.

First, we tested the same instance as before, with a time duration of 700 days and time step of 1 day for the 10 planets.

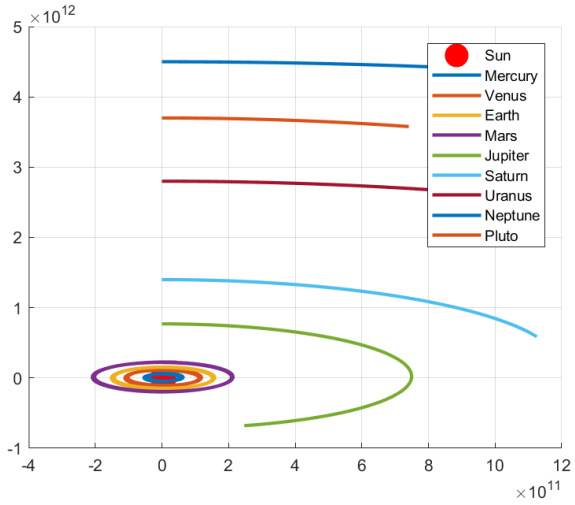

Figure 4: Full Solar System Orbital Position for Heun Iterative ODE Solver - 700 days

As seen before with the inner system, the Heun Iterative ODE seemed to behave as expected. All the solvers were tested, all the outer planets appeared to have relatively the same positional behavior. However, by adding the outer planets to the system the previous error propagation problem mentioned for the RK34 adaptive solve for the inner system, was amplified in the full solar system. The results of solving the system with the RK34 adaptive ODE Solver can be seen in Figure 5.

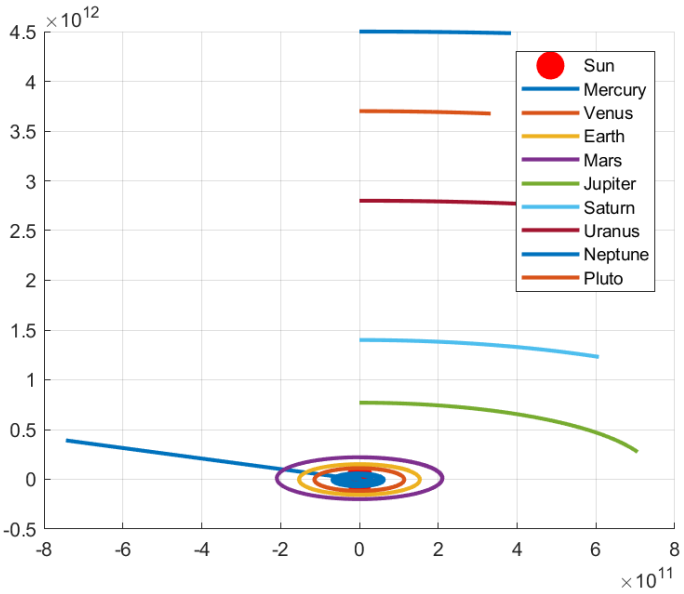

Figure 5: Full Solar System Orbital Position for RK34 Adaptive ODE Solver – 700 days

The results show the mercury planet, flying away after a certain time step. We are fairly certain that mercury's orbit will not behave like this within the next 700 days, thus we believe this problem is due to issues on the adaptive time step. When the outer planets are added, their positional distances and velocities are much larger than the inner planets. The adaptive time step is likely trying to find the optimal time step for the entire system, however our system has relative magnitude differences between all the planet bodies. Thus, adapting the time step taking into consideration the outer planets, is likely causing the time step to become larger than the mercury orbit can handle. When examining the data, the time step appears to be 432000 seconds, which is 5 times the time duration of 1 day. Additionally, a full orbital of mercury can be achieved within 88 days. Thus the large time step is likely too large and as the orbital position errors begin to propagation through the iterative ODE solver process, the result will continue getting worse.

We ran the full solar system for longer durations of time, while always maintaining 1 day as the input march time. As, the time duration got longer, the Heun Iterative and RK34 (with fixed time step) ODE solvers remain stable for the outer planets and relatively stable for the inner planets, as seen in Figure 6. The inner planets must complete multiple orbits for the outer planets to achieve one orbit. Thus the image shows a visualization of how the error propagates as additional orbits are completed. The error propagation ends up showing the center of the inner planets' orbits are moving linearly. This could make sense because the solar system is not stationary, and could perhaps be showing how the full solar system can move together while orbiting each other.

The images in the following page show the results of the simulation (with iterative Heun) for the full solar system for different time durations (corresponding to the orbital period of each planet). To achieve the full orbit of Pluto seen in the last Figure, the time duration provided was 248 years.

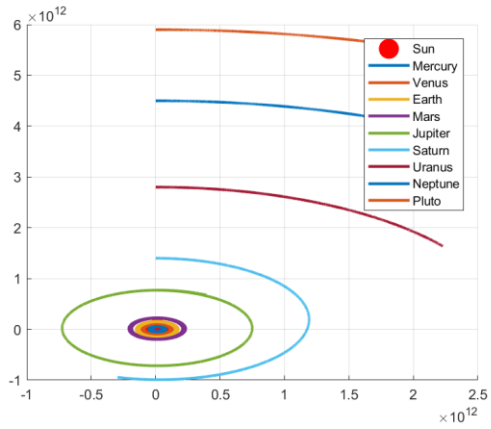

Figure 6: System after 12 years (orbital period of Jupiter)

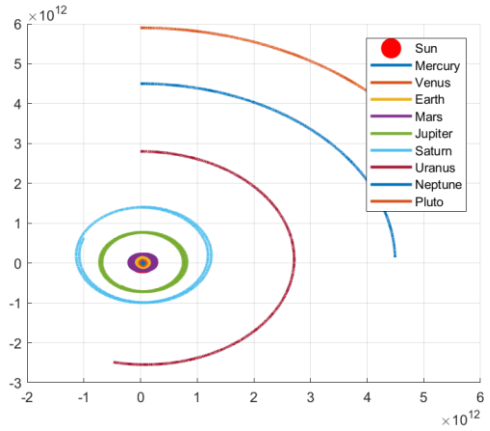

Figure 7: System after 40 years (orbital period of Saturn)

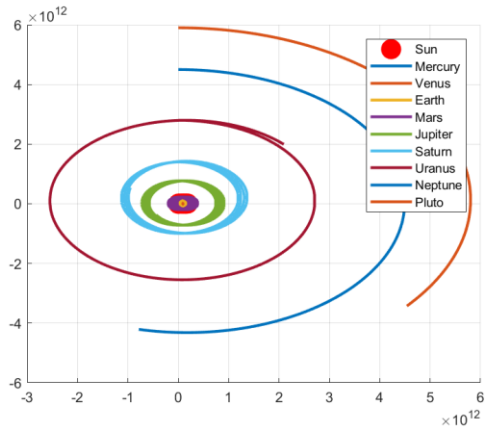

Figure 8: System after 85 years (period of Uranus)

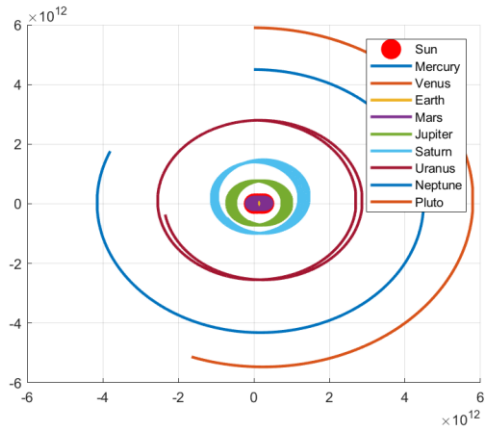

Figure 9: System after 128 years

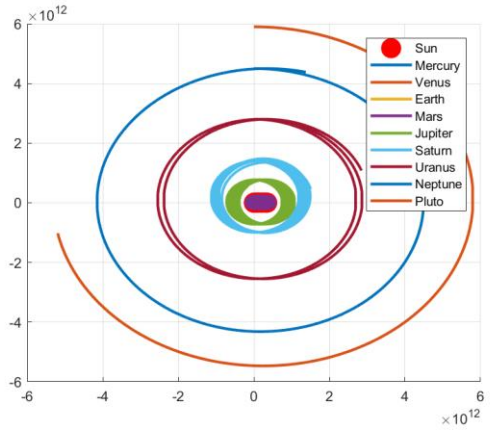

Figure 10: System after 165 years (period of Neptune)

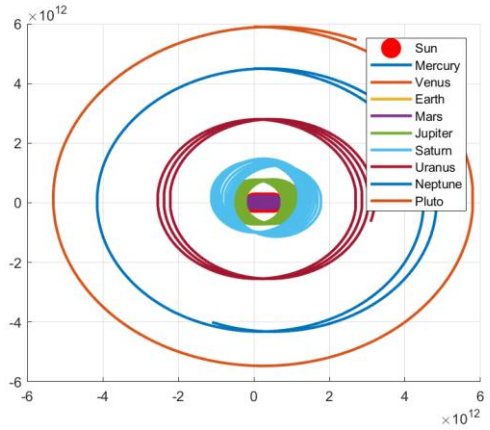

Figure 11: System after 248 years (period of Pluto)

*The rest of the results can be seen in the results folder in the github repository.