(6EMA02) Particle Based Simulations
Assignment 1 - Planetary Motion

Erik Nijkamp 1416057, Marlo Cieraad 1331310, Robbert Lippens 1372270

September 20, 2023

1 Introduction

Planetary (or other heavenly body) motion can (mostly) be modeled using Newtonian mechanics. Doing so has the potential to yield predictive data for the bodies of interest, which may have some relevance when one is, for example, considering the path of an asteroid. Such a simulation would require accurate data regarding the existence, masses, positions, and velocities of the bodies to be simulated to properly initialize the simulation. Such data has been provided to us for September 2021, as has a similar file for September 2022. Using these files, the simulation can be run with actual initial data and can be compared to actual data 1 year later.

2 Modeling

2.1 Gravitational force between particles

Since gravity is a force that acts between all bodies (except interactions with the same body), one can describe the force by looping over all particles and, per particle, computing the force between all other particles. The basic algorithm is described in Equation (1). This, of course, includes many particle interaction repetitions, which is not very efficient. Improvements upon this will be discussed in section 3.3.

$$\sum_{i=1}^{N}\sum_{j\neq i}^{N} G\frac{m_i m_j}{r_{ij}^2} \tag{1}$$

2.2 How to handle bodies of unknown mass?

Some of the bodies included in the provided .dat files do not have a defined mass. This poses problems during both the gravitational force calculations and the translations of these forces to accelerations, since both have a mass dependence. In order to circumvent this issue, when loading the data from the .dat file, a check is performed for masses equaling zero. When this is detected, the mass is instead set to one kg, enabling its inclusion in the calculations. We assume the masses being zero is caused by it being too small to be measured/detected, and thus we can set it to a sufficiently small value, such that its inclusion will not throw off the simulation significantly.

2.3 Equations of motion using Velocity-Verlet scheme

The Velocity-Verlet algorithm is a method where the global error for the position is third order, while the global error for the velocity is second order.The position and velocity are originally defined in the following equations (Equation (2) and Equation (3)).

$$\vec{r_{n+1}} = \vec{r_n} + \vec{v_n} * \Delta t + \frac{\vec{f_n}\Delta t^2}{2m} + O(\Delta t^3) \tag{2}$$

$$\vec{v_{n+1}} = \vec{v_n} + \frac{\Delta t}{2m}\left(\vec{f_{n+1}} + \vec{f_n}\right) + O(\Delta t^3) \tag{3}$$

For implementation in the programming part of the assignment, the above equations can be divided into three equations. This allows the availability of the velocity for the position. In other words, the half-time step (half Δt) calculated velocity can be used for calculating the position. Finally, the velocity is calculated for a full time step (Δt). The implementation in the code can be found in the function updatePlanets3D in Planet.c.

$$\vec{v_{n+\frac{1}{2}}} = \vec{v_n} + \frac{\vec{f_n}\Delta t}{2m} \tag{4}$$

$$\vec{v_{n+1}} = \vec{r_n} + \vec{v_{n+\frac{1}{2}}}\Delta t \tag{5}$$

$$\vec{v_{n+1}} = \vec{v_{n+\frac{1}{2}}} + \frac{\vec{f_{n+1}}\Delta t}{2m} \tag{6}$$

2.4 Total kinetic energy and total potential energy

The kinetic energy per simulated body is determined using Equation (7).

$$E_{kin,i} = \frac{1}{2}m_i\left(\vec{v_{x,i}^2} + \vec{v_{y,i}^2} + \vec{v_{z,i}^2}\right) \tag{7}$$

The total kinetic energy can then be determined by summing the kinetic energy contributions of all bodies.

$$\sum_{i=1}^{N} E_{kin,i} \tag{8}$$

The potential energy between bodies i and j can be calculated using Equation (9):

$$E_{pot,i,j} = \frac{Gm_im_j}{r_{ij}} \tag{9}$$

Where G is the gravitational force, m represents the mass (condition: $i \neq j$) and r_{ij} is the distance between the two planets.

Similarly, we can then sum over all potential energy contributions, making sure only include each pair once.

$$\sum_{i=1}^{N}\sum_{j=i+1}^{N} E_{pot,i,j} \tag{10}$$

```
1  void KEandPE(struct Planet3D *planets[], int numbPlanets, double t, FILE *filePointer)
2
3  {
4      double KE[230];
5      double PE[230];
6      double CurrentPE;
7      double gravConst = 6.6743015 * pow(10, -11);
8      double SumPE = 0;
9      double SumKE = 0;
10     double SumE = 0;
11
12
13     for (int i = 0; i < 230; i++)
```

```
    {
        PE[i] = 0;
    }
    for (int i = 0; i < numbPlanets; i++)
    {
        KE[i] = 0.5 * planets[i]->mass * (pow(planets[i]->vel3D.x, 2) + pow(planets[i]->vel3D
    .y, 2) + pow(planets[i]->vel3D.z, 2));
        for (int j = i + 1; j < numbPlanets; j++)
        {
            struct Vector3D distanceVec = subVec3D(&planets[i]->pos3D, &planets[j]->pos3D);
            double distanceMagnitude = sqrt(inProdVec3D(&distanceVec, &distanceVec));
            CurrentPE = gravConst * planets[i]->mass * planets[j]->mass / distanceMagnitude;
            PE[i] = PE[i] + CurrentPE;
        }
    }
    for (int i = 0; i < numbPlanets; i++)
    {
        SumPE = SumPE + PE[i];
        SumKE = SumKE + KE[i];
    }
    SumE = SumPE + SumKE;
}
```

Listing 1: Kinetic energy and potential energy implementation

3 C Implementation

3.1 Dynamic Memory Allocation

Our implementation of the Planetary motion has dynamic arrays for the planets and all of their characteristics, as well as the forces at every time step. This is shown below. You many notice that we are only reserving the memory for a pointer to the Planet3D struct. This is the case, the storage of the struct itself is also dynamic and is showed in the next section.

```
// Declare pointers to arrays of structures
struct Vector3D **force=(struct Vector3D **)malloc(N_PLANETS * sizeof(struct Vector3D*));
struct Planet3D **planets=(struct Planet3D **)malloc(N_PLANETS * sizeof(struct Planet3D*));
struct Planet3D **planetsEnd=(struct Planet3D **)malloc(N_PLANETS * sizeof(struct Planet3D*))

// Allocate memory for each individual element
for (int i = 0; i < N_PLANETS; i++)
{
    force[i] = (struct Vector3D *)malloc(sizeof(struct Vector3D*));
    planets[i] = (struct Planet3D *)malloc(sizeof(struct Planet3D*));
    planetsEnd[i] = (struct Planet3D *)malloc(sizeof(struct Planet3D*));
}
```

Listing 2: Dynamic memory allocation code.

For reference, the struct Planet3D contains the following data for the planets.

```
struct Planet3D
{
    struct Vector3D pos3D;
    struct Vector3D vel3D;
    struct Vector3D acc3D;
    double mass;
    char name[32];
    int id;
};
```

Listing 3: Struct containing the position, velocity and acceralation vectors, as well as the mass, name and id number of the planet

3.2 Data reading and initialization of simulation

Data is read in via the sscanf() command used in a for loop, such as in the following way.

```
sscanf(line, "%d %s %lf %Lf %Lf %Lf %Lf %Lf",
                    &SimPlanets.body[j].id,
                    SimPlanets.body[j].strName,
                    &SimPlanets.body[j].mass,
                    &SimPlanets.body[j].xpos,
                    &SimPlanets.body[j].ypos,
                    &SimPlanets.body[j].zpos,
                    &SimPlanets.body[j].xvel,
                    &SimPlanets.body[j].yvel,
                    &SimPlanets.body[j].zvel);
```

Listing 4: Code which reads the data from the provied xx files and stores the values for use in the simulation.

The simulation also sets the sun to the center of the simulation. The simulation can also set the momentum of the simulation to 0 in the x, y, and z direction. The idea was that it would prevent the simulation from moving too far from the starting location. However, in practice the initial momentum is not that large, and if the momentum is removed in 2021 then the results do not match the result at 2022.

In section 3.1, it was showed that dynamic memory allocation is used. However, the memory reserved was only used for **pointers** pointing to the Planet3D struct. The memory for the planets is reserved here. Furthermore, the data intake is done in a separate function. This function has no output, but rather takes in the array of array of pointers towards the planets, and alters the pointer to point towards the newly loaded data. It is important to **not** free the memory before exiting the function, as that would prevent the array of pointers from working. A code snippit is shown here, but the full source code for the below function can be found in Appendix A

```
//Function taking in the array of pointers towards the planets
void getData(struct Planet3D *planets[], int numbPlanets, int StartEnd)
{
    struct Planets3D SimPlanets;
    struct Planet3D Sun2021Pos;

    if (numbPlanets > 230)
    {
        printf("Maximum number of planets to simulate is 230, you entered, %d. Simulation set
     for 230 planets.", numbPlanets);
        numbPlanets = 230;
    }

    // Allocating memory for the planet array
    SimPlanets.body = (struct Planet3D *)malloc((numbPlanets) * sizeof(struct Planet3D));

    // Reading the  file
    char line[nMax];
    FILE *file = NULL;
    char *fileToOpen;
    if (StartEnd == 1)
    {
        fileToOpen = "solar_system_13sept2021.dat";
    }
    if (StartEnd == 0)
    {
        fileToOpen = "solar_system_13sept2022.dat";
    }

    /*  Loading in data here    */

    printf("Creating a list of pointers to complete the loading\n");
    for (int j = 0; j < numbPlanets; j++)
    {
        planets[j] = &NewSimPlanets.body[j];
        printf("%s is now stored at %p \n", planets[j]->name, planets[j]);
    }
```

```
}
```

Listing 5: Function used for loading the data of the provided .dat files into the simulation

3.3 Force and acceleration algorithm

The most simple force algorithm is as follows. This simply calculates the force of every planet on each other using a double nested for loop.

```
void CalcForces(struct Vector3D *force[], struct Planet3D *planets[], int N)
{
    for (int i = 0; i < N; i++)
    {
        // Setting the force at the start to 0
        *force[i] = subVec3D(force[i], force[i]);

        for (int j = 0; j < N; j++)
        {
            if (i != j)
            {
                struct Vector3D forceContribution = CalcGravityForce3D(planets[i], planets[j
    ]);
                *force[i] = sumVec3D(force[i], &forceContribution);
            }
        }
    }
}
```

While this does work, the big O is N^2 with regard to the number of planets. The easiest optimization is to only calculate the force once for each pair, this could be done changing the second for loop to start at i + 1, and saving the value of force[j]. This would reduce the run time by half. For further improvement, we need to look into not calculating all the force pairs. The planets included in the .dat file had a very handy ID number. All ID numbers in the 100's are in the Mercury system, 200's are in the Venus system, 300's in the Earth system. So the Moon has ID number 301. Earth has an ID number of 399, as the planet in the system always has x99, however this is not relevant. What is relevant is that all the planets come presorted into systems, or spheres of influence. To limit the number of calculations, the following rules are created.

1. All bodies are influenced by the Sun.
2. All bodies are influenced by Jupiter.
3. All bodies are influenced by every other body in their system (moons and the planet)

The big O notion of this is probably still N^2, but it cuts the computations significantly. This method requires 7223 calculations, while $1/2 \cdot N^2 = 26450$, so a reduction of about 75%. A semi-quantitative analysis will be done later, however it is safe to say that this sphere of influence assumption is a good one. A matrix of the computations is shown in Figure 1 and the implementation of these rules can be found in Appendix A.

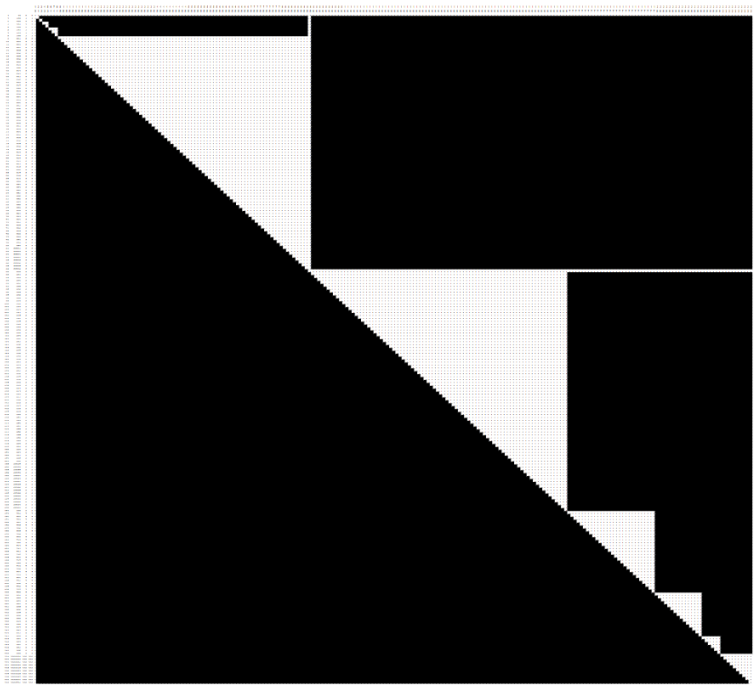
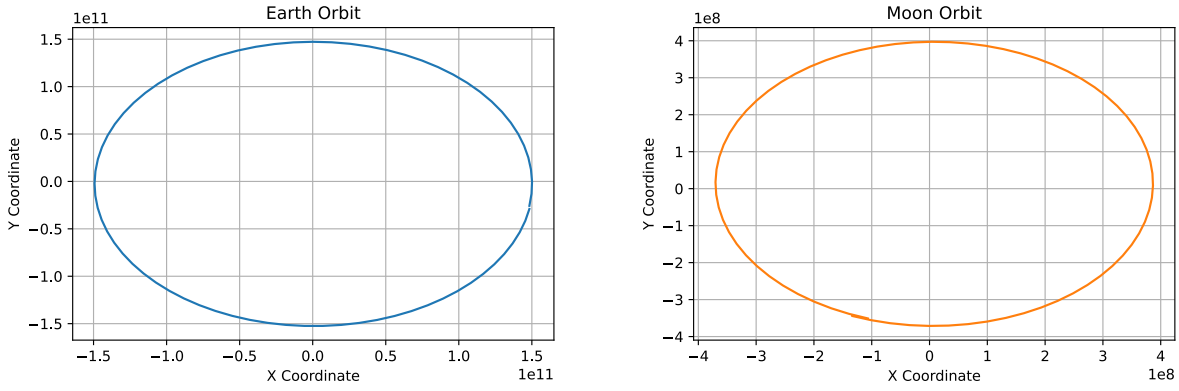


Figure 1: Matrix of computations for the fast force finder

4 Results

4.1 Validation of 3 body system

For a primary validation, we can see if the orbital period of the earth and sun are roughly correct, these are shown in Figure 2. While these may only look like circles, that is the whole idea. From this we can conclude that the bodies complete a full orbit in the specified time.



(a) Earths orbit over 365.25 days

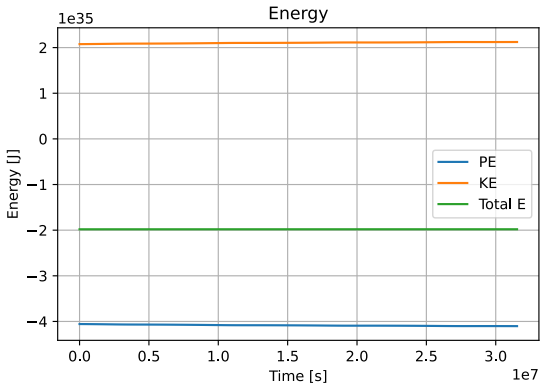
(b) Moons orbit over 27.32 days

Figure 2: Simple orbital period validation.

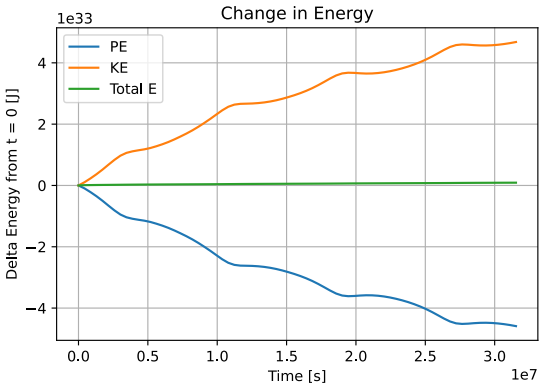
4.2 Energy Conservation

As shown in previous sections, we can calculate the potential energy (PE) and kinetic energy (KE), and the sum of these 2 should always be the same. The amount of energy over 365 days is shown in Figure 3a. As you

can see, the lines are flat, which might indicate that energy is conserved, however, if we look at the difference in energy since the start, we can see that energy is not fully conserved. This is shown in Figure 4b, however it is not bad, and the amount lost depends on the time step. Besides, it is barely visible within a year. (Fig 4b is over 20 years.) The effect of time step size on energy conservation is discussed more in section 4.2.1



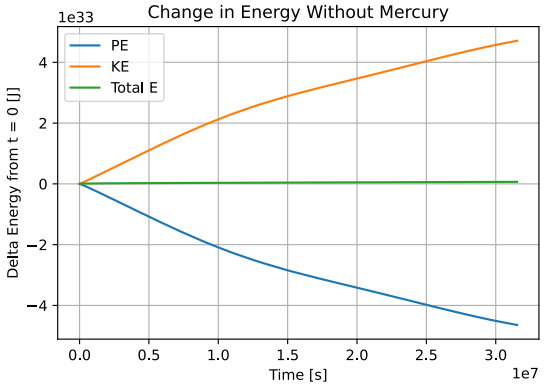
(a) Energy over 365 days



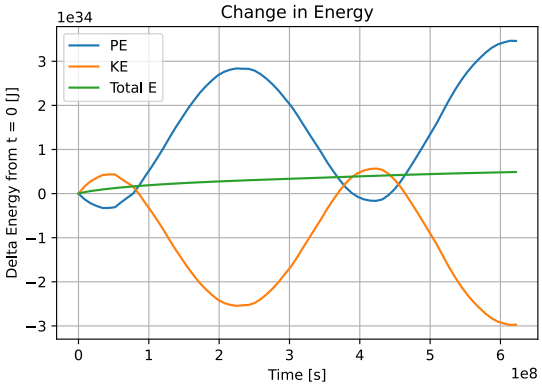
(b) Change in energy over 365 days

Figure 3: Conservation of energy validation.

Interestingly, there is some wobbling in the change in energy. This has a period of about 88 days, or $0.76 \times 10^7 s$ (There is a peak at 0.3, 1, 1.8, and 2.6 e7 s) One body that is somewhat large and has an orbital period of 88 days is Mercury. If we subtract the kinetic and potential energy of Mercury, then the fig. 4a is obtained. Furthermore, if a period of 20 years is examined, some periodicity can still be seen. Here, the period is $\approx 3.6 \times 10^8 s$, or 4166 days or 11.4 years. For reference, Jupiter has an orbital period of 4,333 earth days, so this is the cause of the swings.



(a) Change Energy over 365 days without Mercury



(b) Change in energy over 20 years at dt = 28800

Figure 4: Conservation of energy validation.

4.2.1 Effect of time step on Energy conservation

Figure 5 shows the energy difference as a function of time step. The fast force algorithm is used for this analysis, both algorithms are compared in section 4.2.2.

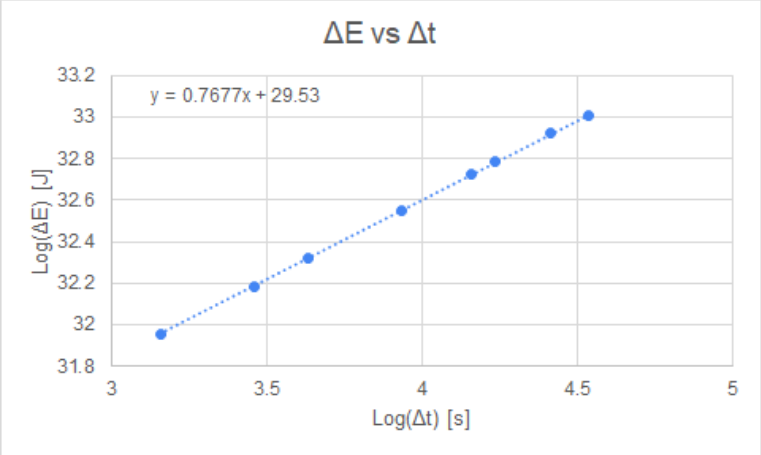


Figure 5: Energy difference at various time steps, shown in a log-log plot with a line of best fit.

The slope of this line is 0.7677. This is not expected as the Velocity-Verlet method is 3 order with respect to time step size. It is not clear why this is not the case. One possible explination say that Δ E may not be a good metric for order of convergence. As shown in the previous sections, potential and kinetic energy swings are dominated by just a couple planets, namely Jupiter, and Mercury. It is possible that this dependence on just a few data points renders the energy difference an insufficient metric for an order of convergence study.

4.2.2 Comparison between the two algorithms

As mentioned in section 3.3, there were 2 force calculation algorithms implemented. One that calculated all the forces in the system, and one that only calculated forces within the same system. The ability to conserve energy for both these systems is shown in Figure 6

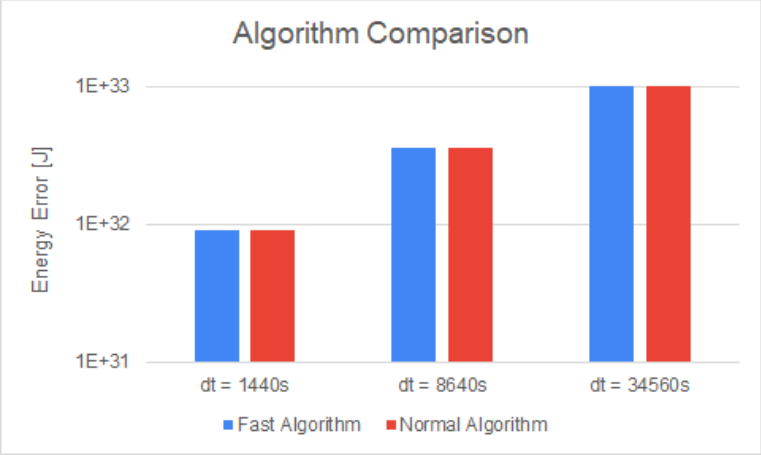


Figure 6: Energy conservation comparison for both algorithms.

Both algorithms give the similar energy errors. While it is difficult to see on the bar graphs, there is a difference of about 0.01% between the 2 methods. Considering that the error of the model is 0.005% at dt = 1440s and the difference between the 2 models is 0.01% of the 0.005% error, or 0.00005% in total, it is safe to use the fast algorithm as is just as valid as the normal algorithm.

4.3 Comparison with actual data

While the change in energy at a dt = 34,560s is only 0.5% per year, that does not mean that the planets are accurate at the time step, and in fact some moons of the gas giants are wildly off. Figure 7. This shows the inner solar system, and Jupiter on the right over a time span of 1 year. As you can see, some of the moons of Jupiter leave in "random" directions, this is a sign that the simulation cannot render these tight, forceful orbits. Luckily, such a simulation only takes ≈ 2 seconds, so it is feasible to reduce the time step time.

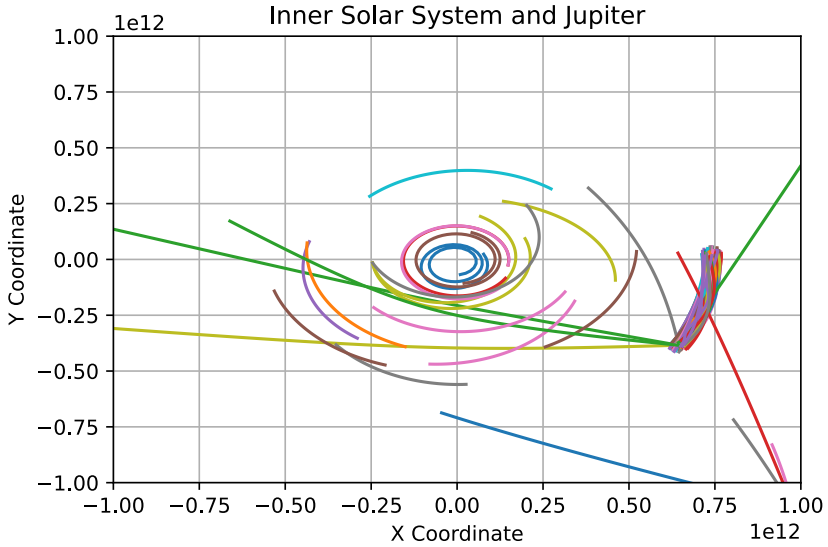


Figure 7: Simulation at dt = 1440 × 24 s = 34,560s

Running the simulation with a dt = 1,440 s gives the following results, which is much better!

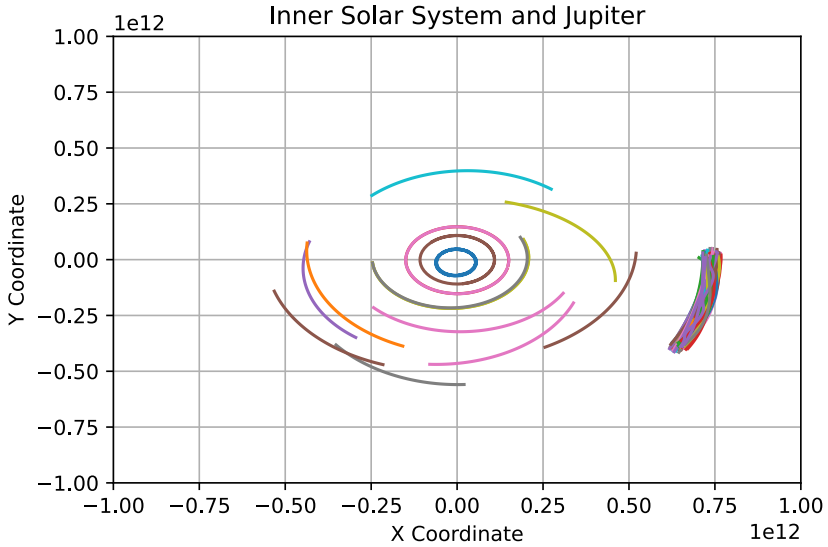


Figure 8: Simulation at dt = 1440 s

To get a quantitative description of the accuracy of the simulation, Equation (11) can be used. This can

be interpreted as the mean percentage difference between the simulation and real life. This is calculated for every planet and shown in a frequency bar chart in Figure 9 As shown, the mean position difference is between 10^{-4}% and 1% with the majority of the bodies having a positional difference of less than 0.1% in each dimension. However, there is still a planet with more than a 10% difference with the expected value. This planet is Mercury, and it has a position difference of 30% in the x and y direction. The reason for this is likely the relativistic effects from the sun, as the increased gravity causes space-time to warp in a way that is not accounted for in this model.

$$\epsilon = \left(\frac{|x_{sim} - x_{real}|}{x_{real}} + \frac{|y_{sim} - y_{real}|}{y_{real}} + \frac{|z_{sim} - z_{real}|}{z_{real}} \right) \times \frac{100\%}{3} \tag{11}$$

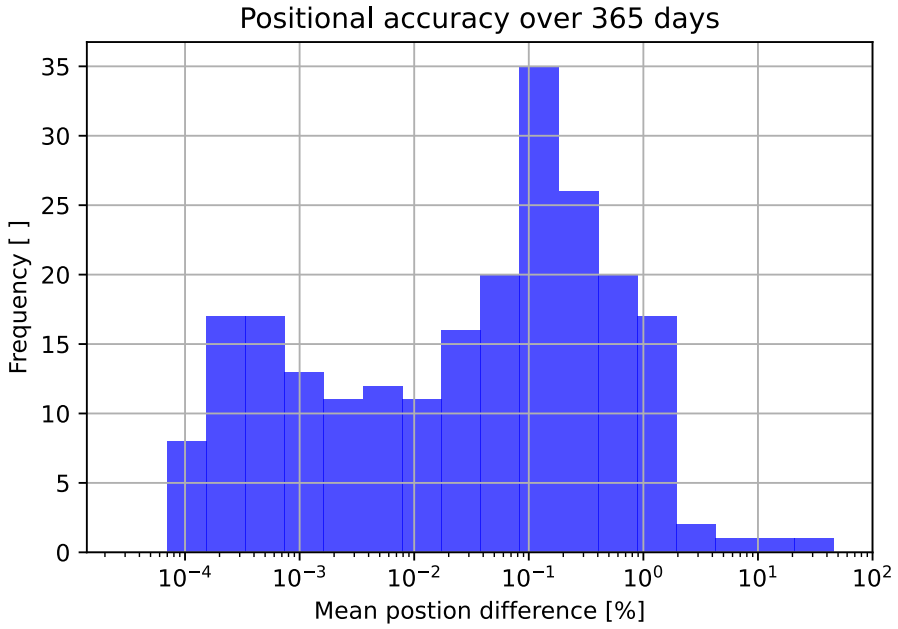


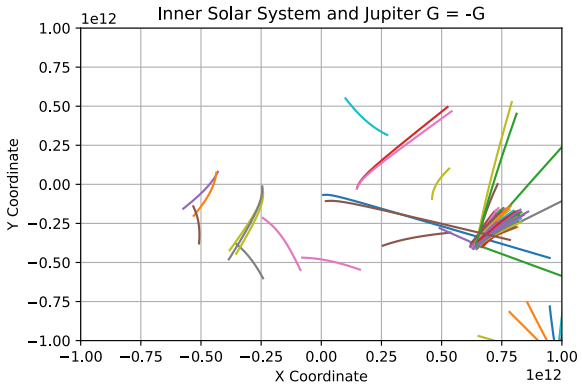
Figure 9: Positional accuracy over 365 days at dt = 360 s

It can also been seen that the frequency distribution looks as if it has 2 peaks. A possible explanation for this is that both these peaks are based on Jupiter and Saturn. Collectively, these 2 planets have 160 moons, which is the majority of the bodies in these simulations. The error of these moon should be similar to similar to their host planet, as they are almost tethered to their host planet. So if the host planet is off by 0.1% then all the moons will be off by $\approx 0.1\%$.

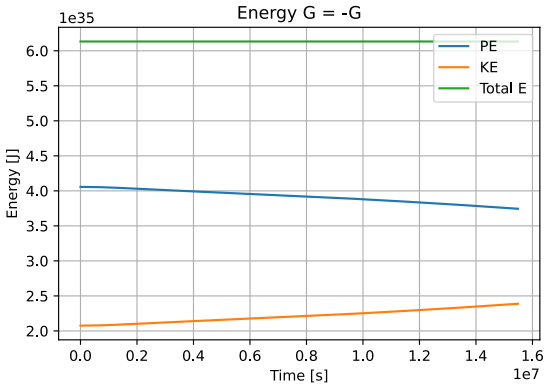
5 Fun Graphs

This concludes this serious analysis of our code and results. However, in our many, many hours of making this code and report, we found a few interesting things, some of which we will show here.

First thing we can look at is the is when we had the gravity constant not negative. This caused the planets to behave much more like computer science students, as they tried to create as much distance between each other as possible. This is shown in fig. 10. It is still nice that energy is conserved in such a case. It took us longer than we care to admit to figure out why the planets were not going in a circle, as we had G = -G to begin with.



(a) The path of the planets with reversed gravity over 180 days



(b) Change in energy over 180 days with reversed gravity

Figure 10: Results in the case of reversed gravity

We would also like to show the limiting bodies in the simulation. One might ask why can we not make dt as large as we want? Moons with very fast orbits limit this. Below is the Uranus system over **1 day.** While it is hard to see, there a lot of bodies with orbital periods with less than 1 day, with the shortest one being 1/3 of a day, or 28,800 seconds. If you wanted to sample the orbit 80 times, then a time step of 360 seconds would be needed, which was what was used for the figure. Also, note that the figure has axis x and z coordinate, based on Uranus at 0,0.

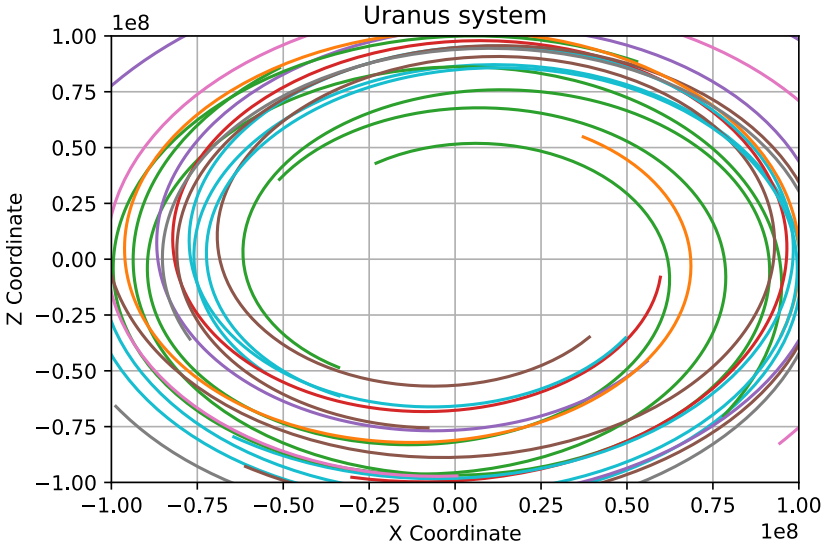


Figure 11: Uranus system over 1 day

The last image will be an Earth centered view of the inner solar system. This was accomplished by subtracting Earths positions from every other planet during post-processing. In the center you can see a pink dot, that is the moon. The brown loop closest to the Earth is Mars, and gets the closest to Earth of any planet. The blue line is Mercury, and you can see its loops at it goes "retrograde." The only perfect circle here is the sun $\approx$ 2e11 m from the Earth.

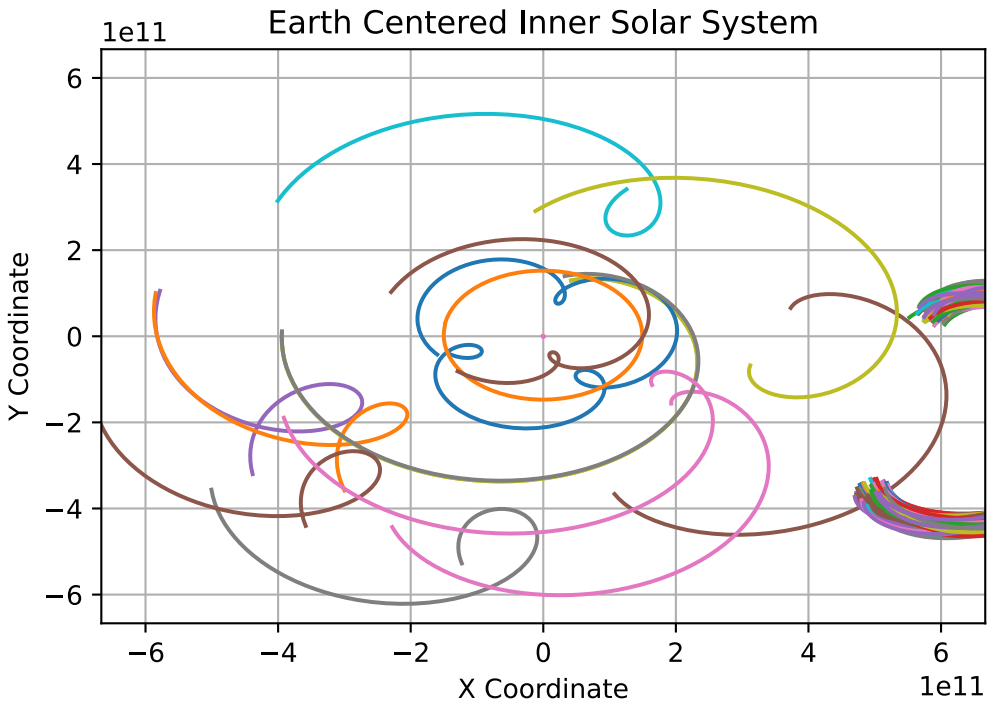


Figure 12: Earth centered view of the solar system over 1 year

A Appendix

```
#define nMax 1024

void getData(struct Planet3D *planets[], int numbPlanets, int StartEnd)
{
    struct Planets3D SimPlanets;
    struct Planet3D Sun2021Pos;

    if (numbPlanets > 230)
    {
        printf("Maximum number of planets to simulate is 230, you entered, %d. Simulation set
     for 230 planets.", numbPlanets);
        numbPlanets = 230;
    }

    // Allocating memory for the planet array
    SimPlanets.body = (struct Planet3D *)malloc((numbPlanets) * sizeof(struct Planet3D));

    // Reading the file
    char line[nMax];
    FILE *file = NULL;
    char *fileToOpen;
    if (StartEnd == 1)
    {
        fileToOpen = "solar_system_13sept2021.dat";
    }
    if (StartEnd == 0)
    {
        fileToOpen = "solar_system_13sept2022.dat";
    }

    file = fopen(fileToOpen, "rb");
    if (file == NULL)
    {
        printf("Cannot open specified .dat file");
        exit(1);
    }

    // looping over everyline to grab the stuff
    int j = 0;
    while (fgets(line, nMax, file) != NULL)
    {
        if (j >= 1)
        {
            if (j <= numbPlanets)
            {
                j = j - 1;
                sscanf(line, "%d %s %lf %lf %lf %lf %lf %lf %lf",
                       &SimPlanets.body[j].id,
                       SimPlanets.body[j].name,
                       &SimPlanets.body[j].mass,
                       &SimPlanets.body[j].pos3D.x,
                       &SimPlanets.body[j].pos3D.y,
                       &SimPlanets.body[j].pos3D.z,
                       &SimPlanets.body[j].vel3D.x,
                       &SimPlanets.body[j].vel3D.y,
                       &SimPlanets.body[j].vel3D.z);
                j = j + 1;
            }
        }
        j = j + 1;
    }

    file = fopen(fileToOpen, "rb");
    fgets(line, nMax, file);
    sscanf(line, "%d %s %lf %Lf %Lf %Lf %Lf %Lf",
           &Sun2021Pos.id,
           Sun2021Pos.name,
```

```
            &Sun2021Pos.mass,
            &Sun2021Pos.pos3D.x,
            &Sun2021Pos.pos3D.y,
            &Sun2021Pos.pos3D.z,
            &Sun2021Pos.vel3D.x,
            &Sun2021Pos.vel3D.y,
            &Sun2021Pos.vel3D.z);

    for (int i = 0; i < numbPlanets; i++)
    {
        // If a planet is "massless, set the mass to 1 kg"
        if (SimPlanets.body[i].mass == 0)
        {
            SimPlanets.body[i].mass = 1;
        }
        // Setting the sun to the center of the simulation
        SimPlanets.body[i].pos3D.x = SimPlanets.body[i].pos3D.x - Sun2021Pos.pos3D.x;
        SimPlanets.body[i].pos3D.y = SimPlanets.body[i].pos3D.y - Sun2021Pos.pos3D.y;
        SimPlanets.body[i].pos3D.z = SimPlanets.body[i].pos3D.z - Sun2021Pos.pos3D.z;

    }

    for (int j = 0; j < numbPlanets; j++)
    {
        planets[j] = &SimPlanets.body[j];
    }
}
```

Listing 6: Function to load data from the .dat files.

```
void CalcForcesFast(struct Vector3D *force[], struct Planet3D *planets[], int N)
{
    int system[230];
    struct Vector3D forceContribution;
    // Setting the system of the planets
    for (int i = 0; i < N; i++)
    {
        if (planets[i]->id < 1000)
        {
            system[i] = planets[i]->id / 100;
        }
        else
        {
            system[i] = planets[i]->id / 10000;
        }
    }
    // Doing the force calcuations
    // Looping over every planet
    for (int i = 0; i < N; i++)
    {
        // Setting the force at the start to 0
        // The sun has influnce on everything
        if (planets[i]->id == 10)
        {
            // Setting the force on the Sun to 0
            *force[i] = subVec3D(force[i], force[i]);
            for (int j = 1; j < N; j++)
            {
                // Setting the force of the planet to 0
                *force[j] = subVec3D(force[j], force[j]);
                forceContribution = CalcGravityForce3D(planets[i], planets[j]);
                *force[i] = sumVec3D(force[i], &forceContribution);
                // The force on the planet is oppisete of the sun
                *force[j] = subVec3D(force[j], &forceContribution);
            }
        }
        // Loop for planets
        else
        {
```

```
            // Jupiter's pull is also omnipresent
            if (planets[i]->id == 599)
            {
                for (int j = 1; j < N; j++)
                {
                    // don't double update Jupiter's system
                    if (system[j] != 5)
                    {
                        forceContribution = CalcGravityForce3D(planets[i], planets[j]);
                        *force[i] = sumVec3D(force[i], &forceContribution);
                        // The force on the planet is oppisete of the Juipter
                        *force[j] = subVec3D(force[j], &forceContribution);
                    }
                }
            }
            else
            {
                // NOTE: Looping from i+1 onwards.
                for (int j = i + 1; j < N; j++)
                {
                    // Only calc forces if they are in the same system
                    if (system[i] == system[j])
                    {
                        forceContribution = CalcGravityForce3D(planets[i], planets[j]);
                        *force[i] = sumVec3D(force[i], &forceContribution);
                        // The force on the planet is oppisete of the other planet
                        *force[j] = subVec3D(force[j], &forceContribution);
                    }
                }
            }
        }
    }
}
```

Listing 7: Improved function for calculating the forces between the bodies.