Simulating n-body Choreographies for $n \geq 3$

Seth Rohan Sharma

July 7, 2026

Contents

Introduction

By 1846, Uranus had completed nearly one full orbit from its discovery in 1781, and astronomers had detected many inconsistencies and errors in its path that were not explained by Newton's law of universal gravitation. However, these irregularities could be corrected by the existence of an unknown eighth planet beyond the orbit of Uranus. Astronomer Urbain Le Verrier began his calculations for the location of this new planet in 1845. Just after midnight on September 24^{th}, astronomer Johann Galle[1] made telescopic observations confirming the existence of a major planet at the position that Le Verrier reported. This was a revolutionary discovery and a dramatic confirmation of Newtonian gravitational theory, since the planet was mathematically predicted before it was directly observed. [1].

The n-body problem is the problem of calculating future positions of a system based on initial conditions. This is incredibly important in gravitational problems, however it also has applications in other fields such as electrostatics [2] and machine learning. The example above breaks down the essence of n-body problems, in which every gravitational body has some effect on every other body. Calculating various n-body problems of $n \geq 3$ has been heavily researched as it has practical uses for predicting the motions of the Sun, Moon and the Earth. This paper will discuss some numerical methods used to solve n-body problems and then provide some examples of n-body choreographies, where the bodies have periodic motion along a set path.

1 Integration

Integration is the process of relating an variables rate of change to its accumulated change over time (or x). It can be used to find areas, volumes and various other useful things in mathematics. In this chapter we will show integration in the context of linear motion, and how that can be used to create a physics simulation.

[1] Assisted by Heinrich Louis d'Arrest

[2] The simulation of the creation of proteins

1.1 Linear Motion

In simulations, calculating the relationship between displacement, velocity and acceleration is incredibly important, as it forms the foundations of the simulation, the ability for objects to move under newtons laws.

Velocity v is the rate of change of displacement s over time, and acceleration a is the change of velocity over time. This can be written as:

$$v = \frac{ds}{dt}$$

$$a = \frac{dv}{dt}$$

Inversely, velocity is the integral of of displacement with respect to time, and acceleration is the integral of velocity with respect to time.

Using these equations, we can create graphs of the displacement, velocity and acceleration of different objects by plotting their displacement s, $\frac{ds}{dt}$ and $\frac{d^2s}{dt^2}$ against t. Integration of these values measures the area under the line allowing you to go from acceleration to calculating velocity and differentiating the change in velocity allows you to measure the acceleration however we will only be needing the former of these.[3]

1.2 Gravity

To calculate the accelerations used in the simulation methods we are about to disucss, we will need to cover the formula for gravitational attraction. I started with the basic formula for gravitational attraction between two bodies. [2]

$$F = \frac{mv^2}{r} = \frac{GMm}{r^2}$$

I can then adapt this using my knowledge of vectors and resolving forces from further maths to generalise it. The formula below resolves all of the gravitational forces as a 3d vector.

[3] We will never need to find the acceleration of an object based on the rate of change of its velocity for this simulation.

For a body i, $\vec{r_i} = (x_i, y_i, z_i)$, $\vec{v_i} = (v_{xi}, v_{yi}, v_{zi})$, $\vec{a_i} = (a_{xi}, a_{yi}, a_{zi})$ and N is the total number of bodies, the acceleration of body i can be written as:

$$\vec{a_i} = -G \sum_{j \neq i}^{N} \frac{m_j}{|\vec{r_i} - \vec{r_j}|^3} (\vec{r_i} - \vec{r_j})$$

Computing this value for each object in a system gives us a 3d vector of acceleration for every object however we can change the formula slightly as we only need to create a 2-dimensional simulation. Unfortunately, this computation has a complexity of $O(n^2)$ so it becomes intractable with large datasets however this is unavoidable for a gravitational simulation.

1.3 Numerical Integration

In this section we will talk about some methods of numerical integration and how they can be applied to create a gravitational simulation. Numerical integration is used to iterate over a time series to calculate the area under a graph. This can be used to create a simulation for gravity by adjusting the acceleration based on the positions of objects in the system, then calculating what the expected change in position should be. When using a low resolution time-step these methods of integration are highly inaccurate, however when you increase the resolution the accuracy increases.

1.3.1 Time stepping

Before discussing specific methods of numerical integration, we have to talk about time stepping. Given that the displacement of an object on the x-axis is x then the simulated time in between updating the value of x is called the *time step*.[4] Most programs opt to use 120fps (or 8ms), however, I will be using 60fps (or 16ms).

The basic method of numerical integration can be demonstrated with a simple example. Let s be the position along the x-axis of an object travelling at s' m/s. The position of the object at time step n can be expressed as s_n at time t. This means that a singular time step can be expressed as:

[4]The use of the term *simulated time* is important here as the time step does not necessarily have to be the same amount of time it takes to calculate the next position.

$$s_{n+1} = s_n + s'\Delta t$$

From this equation, the new displacement can be calculated from the previous displacement plus the velocity multiplied by the length of the time step. The new displacement is based on the displacement in the previous time step, and thus it is an iterative process.

In this example, we have assumed that the velocity is constant and that the acceleration is zero. For a real simulation, you would sum together the forces on an object as shown above, calculate its acceleration, and then input that into the iterative method to calculate the new velocities and displacements.

We will now discuss some specific algorithms used for numerical integration compare them based on: complexity, ease of implementation and accuracy.

1.3.2 Explicit Euler Integration

Explicit Euler Integration (or simply Euler Integration) is the implementation of the algorithm discussed previously. It uses the first derivative and evaluates the new displacements and velocities using the acceleration calculated by summing the forces on the object at the current time. These equations are used to describe them:

$$v_{n+1} = v_n + a_n\Delta t$$

$$s_{n+1} = s_n + v_n\Delta t$$

This is the easiest and least complex algorithm to implement, however it is also the most innacurate and tends towards instability (unless very small time steps are used). Due to this inaccuracy problem, most games and simulations tend to use one of the other algorithms in this list.

1.3.3 Implicit Euler Integration

Implicit euler integration involves using a future acceleration to calculate the new velocities and displacements of objects. This is done using the formulas:

$$v_{n+1} = v_n + a_{n+1}\Delta t$$

$$s_{n+1} = s_n + v_{n+1}\Delta t$$

Whilst this is more accurate than Explicit euler integration, it is (obviously) not as practical for our applications as it only works if we know future accelerations, and if we did then we would already know the future state of the simulation and thusly would not need to calculate any new values. We will not be considering this method further for these reasons.

1.3.4 Semi-implicit (Symplectic) Euler Integration

Semi-implicit integration combines the techniques of the two previously stated algorithms to create a mostly stable simulation. We do this by calculating the next velocity using the accelerations calculated by resolving the forces on the object, then using the future velocity, calculate the new displacement.

$$v_{n+1} = v_n + a_n\Delta t$$

$$s_{n+1} = s_n + v_{n+1}\Delta t$$

This method, whilst being the same $O(n^2)$ complexity, produces far more stable systems than Explicit or Implicit Euler differentiation.

To improve this method further, we can use a leapfrog algorithm. This essentially means using a half step between n and $n+1$ called $n+\frac{1}{2}$. This allows us to refine our accuracy even further whilst keeping the complexity the same. The fomulas for this are:

$$s_{n+1} = s_n + v_{n+1/2}\Delta t$$

$$v_{n+3/2} = v_{n+1/2} + a_{n+1}\Delta t$$

This final step is the algorithm that most game engines and simulations use, at it is the most accurate with complexity $O(n^2)$, however due to time constraints I will be using basic symplectic integration.

1.4 How do we make a time stepping simulation

The time stepping simulation uses the equations we described above and puts them in an iterative code format. 60 time steps per second, this algorithm takes the displacement, velocities, accelerations and masses of every object in a two-dimensional space and applies a leapfrog integration to calculate the new values. We can adjust the initial values of the simulation to start from and have different outcomes occur, including finding a stable solution. [3]

2 Choreographies

Choreographies are periodic solutions of the n-body problem in the case of equal masses. This means that each point mass in the system follows an orbit that repeats. This can lead to some really interesting patterns when you graph out what it looks like. Some choreographies are easy to calculate, such as equidistant points on a circle, and others are more complicated such as the process to find the figure of eight solution. In this final section, we will cover how the program I created works and how we can use it to find n-body choreographies for $n \geq 3$.

2.1 The Program

Using the symplectic integration algorithm explained above, I created a program that plots on a canvas the locations of different objects, resolves their forces, calculates new positions and redraws the objects. It is very basic, using point masses with no collision however it can take values of found choreographies and simulate them. The code is a bit too complex to cover in this paper however it can be found on my website in the references.

2.2 n-body Choreographies for $n = 3$

The first choreography I created was the simplest one I could find. Two point masses at $x = -1000$ and $x = 1000$ orbit clockwise around a point mass at $(0, 0)$. The code for creating these bodies is:

```
objects.push(new object(0, 0, 0, 1e9, 0, 0));
objects.push(new object(1000, 0, 0, 1e9, 0, 2 * speed));
```

```
objects.push(new object(-1000, 0, 0, 1e9, 0, -2 * speed));
```

where the object attributes are: X position, Y position, radius, mass, X velocity and Y velocity respectively. Additionally, speed is half the initial speed of the points adjusted using a slider. After putting this into the program, it returned this nice output:

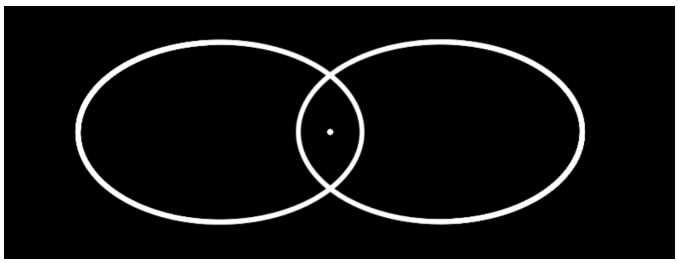

Figure 1: A 3-body choreography with one stationary central mass and two elliptical periodic orbits

The next choreography I wanted to create was a circle with three equidistant point masses. The code to create this looked like this:

```
objects.push(
    new object(
      1000, 0, 10, 1e12,
      (speed / 20) * Math.cos(Math.PI / 2),
      (speed / 20) * Math.sin(Math.PI / 2),
    ),
  );
  objects.push(
    new object(
      1000 * Math.cos((2 * Math.PI) / 3),
      1000 * Math.sin((2 * Math.PI) / 3), 10, 1e12,
      (speed / 20) * Math.cos(Math.PI / 2 + (2 * Math.PI) / 3),
      (speed / 20) * Math.sin(Math.PI / 2 + (2 * Math.PI) / 3),
    ),
  );
  objects.push(
    new object(
      1000 * Math.cos((4 * Math.PI) / 3),
      1000 * Math.sin((4 * Math.PI) / 3), 10, 1e12,
      (speed / 20) * Math.cos(Math.PI / 2 + (4 * Math.PI) / 3),
```

```
      (speed / 20) * Math.sin(Math.PI / 2 + (4 * Math.PI) / 3),
    ),
  );
```

This new code is very similar to the last but with all of the points starting on the circle. It created this choreography:

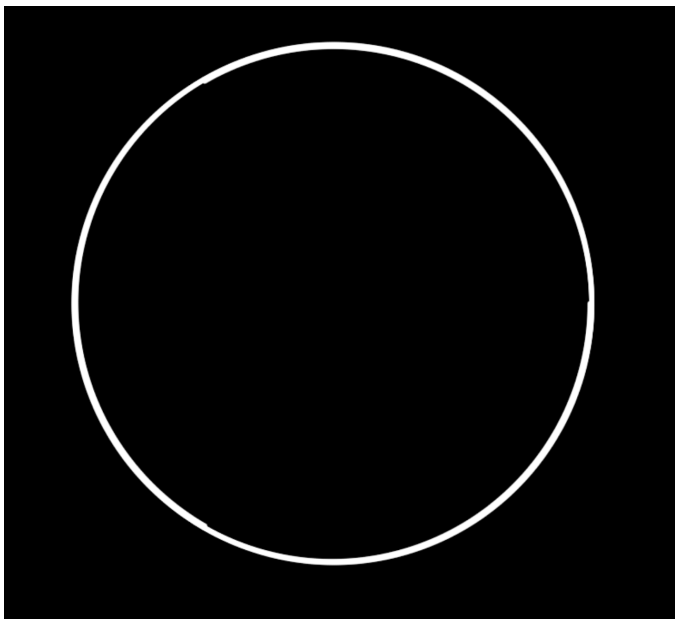

Figure 2: A relatively boring looking circle that represents the point masses staying in a stable orbit around $(0, 0)$

The final choreography I wanted to simulate was the figure of eight. I found some initial values from the initial conditions computed by Carles Simo [4] and inputted them into my program using this code:

```
objects.push(new object(970, -243, 10, 4.17e15, 466, 432));
objects.push(new object(-970, 243, 10, 4.17e15, 466, 432));
objects.push(new object(0, 0, 10, 4.17e15, -2 * 466, -2 * 432));
```

This code looks slightly different to the other snippets as the speed slider was not fine enough for me to tune the variables so I had to do it in the code. This starting position created the nice choreography seen in Figure 3.

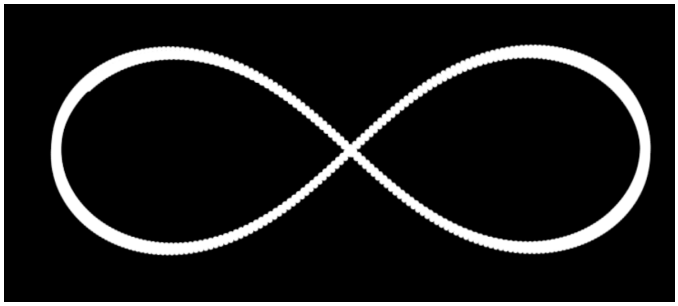

Figure 3: A nice figure of eight choreography

Now, whilst this was a nice choreography, I noticed that it had more variation than the other two. Seeing this, I decided to implement a time speed up feature to allow the simulation to run at x32768 times speed. Interestingly, after doing this for a short while I got the result seen in Figure 4.

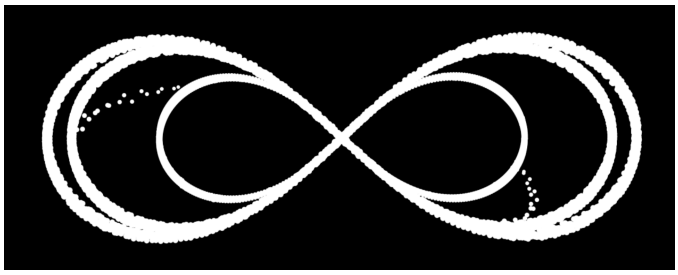

Figure 4: The figure of eight choreography after being sped up

I suspect the layers seen here in Figure 4 are where either the symplectic algorithm's errors accumulating or my inaccurate initial positions and velocities eventually caused a jump in the masses, which then resolved itself (somehow). I really nice result and exactly what I was hoping for going into this project.

2.3 n-body Choreographies for $n > 3$

The final section of this project (which is the whole reason for the name) was to create and simulate some stable n-body choreographies. To do this, I created a for loop that creates new point masses at n points around a circle, then gives them a tangential velocity. With the slider to adjust velocity and one to change n, I have now created a fully customisable simulation:

```
let nBodies = count;
let dist = 1000;
scale = 400 / dist;
let angle = 0;
let velocity = speed * 5;
for (let i = 0; i < nBodies; i++) {
    let x = dist * Math.cos(angle);
    let y = dist * Math.sin(angle);
    let x_v = velocity * Math.cos(angle + Math.PI / 2);
    let y_v = velocity * Math.sin(angle + Math.PI / 2);
    objects.push(new object(x, y, 10, 10e9, x_v, y_v));
    angle += (Math.PI * 2) / nBodies;
}
```

A lot of the code shown here isn't relevant to the project and is just part of the program, but the most important new variable is nBodies. This variable decides how many pieces the circle should be split into and what the angles should be. To stop the choreographies simply all being circles, I have the velocity lowered slightly below what it should be to create some more interesting shapes, seen here:

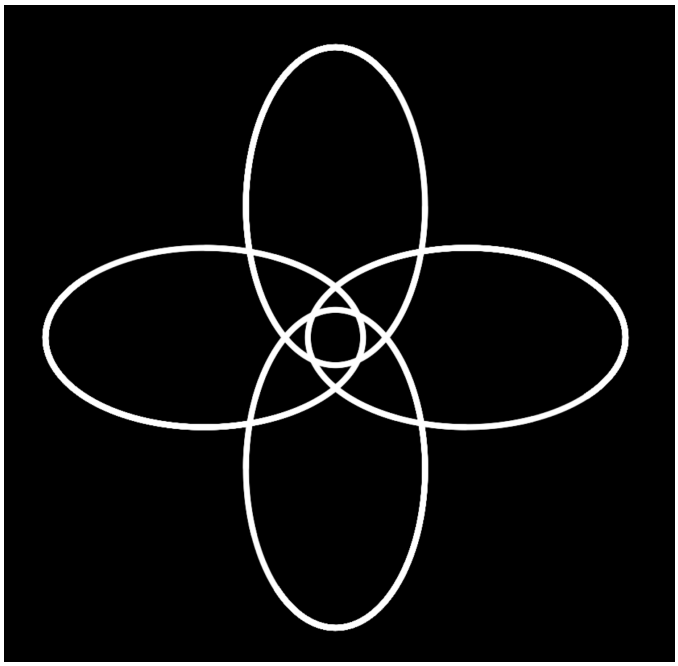

Figure 5: Nice symmetrical choreography with $n = 4$

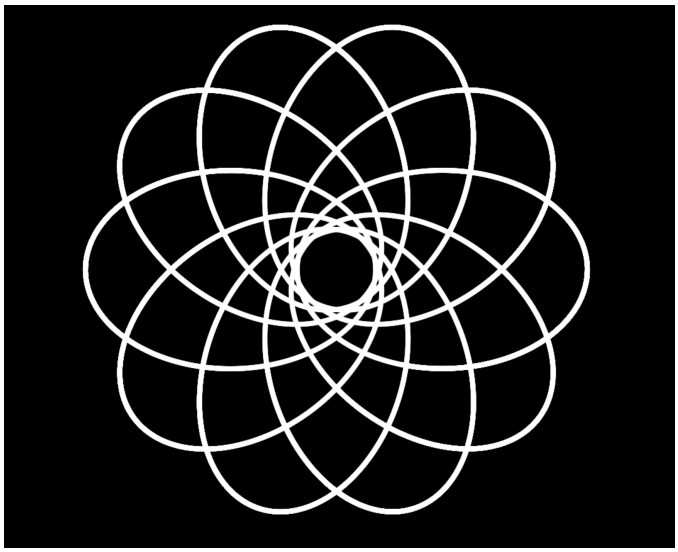

Figure 6: Choreography with $n = 10$

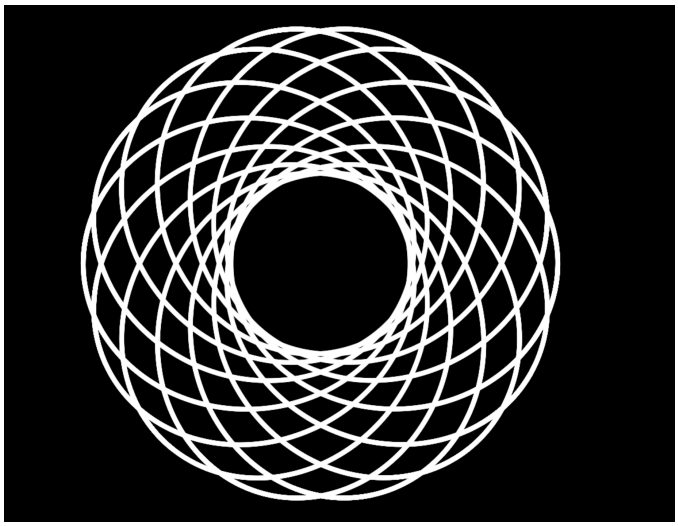

Figure 7: Choreography with $n = 14$ with the velocities very close to that which would form a circle

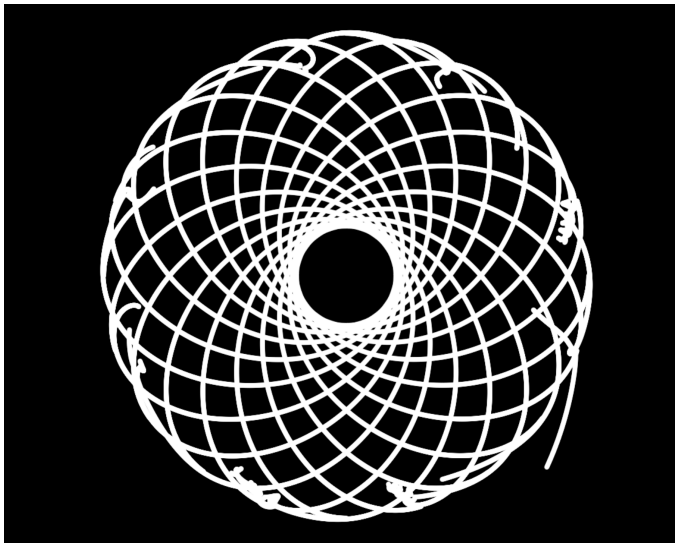

Figure 8: Choreography with $n = 20$ with some errors showing through at the end

Conclusion

This project has been eye opening for my understanding of orbital mechanics and how n-body systems can be simulated and understood. We have seen have seen different choreographies for $n = 3$ and multiple possible solutions for $n > 3$. Additionally we have explored how the different algorithms for numerical integration have different applications in different scenarios, ranging from games to simulations to economics. If I were to improve on this simulation, I would focus on making the simulation more accurate, perhaps by using the Range-Kutta algorithm (which I did not discuss in this paper) which is another method of numerical integration which is more complex but far more accurate.

References

[1] Wikipedia article on the discovery of Neptune: `https://en.wikipedia.org/wiki/Discovery_of_Neptune` (document)

[2] Resolving gravity equation: Bone, G., Chadha, G., Saunders, N. (2015). A Level Physics for OCR A. Oxford University Press. (Original work published 2015) 1.2

[3] 2017 Tutorial on Numerical Integration Methods: `https://shorturl.at/qrOdb` 1.4

[4] Figure eight choreograpy: `https://arxiv.org/pdf/math/0011268` 2.2

[5] My website with the simulation: `https://sethsharma.net`