The n-body problem
Mathematical Modelling

Janez Tomšič, Matevž Vidovič, Nace Kovačič

June 2022

1 The n-body problem in general

The n-body problem is the problem of predicting the dynamics of a system with n-bodies in $\mathbb{R}^3$, each with it's own velocity, mass and position. We consider bodies to have point masses.

There are two important laws in physics that help us understand this system.

Newton's second law states that the sum of all forces on the body equals the product of it's mass and acceleration.

$$\Sigma \vec{F} = m \cdot \vec{a}$$

Newton's law of gravity states that all bodies gravitationally affect one another. Suppose we have two bodies, with their respective masses and positions. We write a gravitational constant as G, the masses of the bodies as m_1 and m_2, their positions as $\vec{r}_1$ and $\vec{r}_2$ and the distance vector between the two objects as $\vec{r} = \vec{r}_2 - \vec{r}_1$. $\vec{r}^\star = \frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|}$ represents a unit vector from the first body to the second. The force with which the first body is attracted to the second can then be expressed as

$$\vec{F} = \frac{G m_1 m_2}{|\vec{r}|^2} \vec{r}^\star.$$

The problem is not so difficult from a mathematical point of view (the equations are quite straight-forward), but is rather difficult as a programming problem. The main challenge lies in optimising the program to compute a large system with many bodies in real time.

2 Implementation

2.1 Python

We decided to first implement the simulation of the n-body system in Python, because we wanted to start with something simple and then gradually improve from there. The two main Python modules we used are:

- `numpy` for working with vectors and storing positions, velocities and masses of bodies.
- `pygame` for real-time visual representation of the system dynamics.

Before we can start the simulation, we need initial conditions of the system - positions, velocities and masses. We decided to store these conditions in a csv file, which is read before the simulation starts. In Python, we can read a csv file using the `numpy` module. The number of bodies can be determined from the number of rows in csv file. First three values in a row are the body's initial position, second three its initial velocity and the last value is its mass.

```
data = np.genfromtxt(filename, delimiter=',')[1:]
num_bodies = len(data)
positions  = data[:, 0:3]
velocities = data[:, 3:6]
masses     = data[:, 6]
```

After reading the initial conditions we can start the simulation. Each iteration, we must first compute the gravitational forces between bodies and the accelerations caused by these forces. We can halve the number of computations of forces by only calculating the force between a pair of bodies once. The force between two bodies is oppositely directed, which means that two bodies attract each other with the force of equal magnitude, each of them pulling the other one towards itself.

```
accelerations = np.zeros((num_bodies, 3))
for i in range(num_bodies):
    for j in range(i + 1, num_bodies):
        # Newton's Gravity Equation
        dist = positions[j] - positions[i]
        a = G * dist / (np.linalg.norm(dist) ** 3)
        # Newton's second law
        accelerations[i] += masses[j] * a
        accelerations[j] -= masses[i] * a
```

When the forces and accelerations are computed, we only need to change velocity of the bodies and appropriately move them according to a small time step, denoted by `DT`. Here we are essentially applying Euler's method to the system.

```
velocities += DT * accelerations
positions  += DT * velocities
```

The finished implementation in Python is able to simulate a system of around 100 bodies in real time. In the end we also implemented basic movement around the scene in Python. You can drag and zoom in or out of the scene using the mouse and rotate the scene with W, A, S, D, E and Q keys.

2.2 Julia

As Julia is a more and more popular programming language in the field of scientific computing we decided to try it out. It is supposed to be a very fast language, only somewhat slower than C, while retaining the useful syntactic sugar needed for working with vectors and matrices. In many ways it is very similar to Octave, except that you need to declare types of numbers (e.g. float32 or float64), and that is in general a bit more robust. The implementation was a simple computation of forces on each step, and then computing the acceleration and thus computing the next position and next velocity for the object. After pre-baking the data with julia, we could easily plug the output data (positions and velocities at each step) to unity and render the behaviour. For the simulation of 2000 steps the program needed:

- 13 seconds for 100 bodies.
- 30 seconds for 200 bodies.
- 6 minutes for 800 bodies.
- 25 minutes for 1400 bodies.

We can see, that the growth was roughly a function of n^2.

2.3 Unity

Our first implementation in Unity was quite simple. The main focus of this step was for us to familiarize ourselves with Unity syntax and Unity in general and to create a rough draft upon which we could later build.

2.3.1 PlanetSpawner.cs

We created a `PlanetSpawner` object, which was responsible for reading the data from the csv and spawning the objects.

To populate space, we first initialized an array before proceeding to read the csv file. Each line of the csv file would contain seven separated values, allowing us to initialize the body. Those values were

- position on the x axis
- position on the y axis
- position on the z axis
- velocity on the x axis
- velocity on the y axis
- velocity on the z axis

- mass of the body

We iterate over read csv data, at each step initialising a GameObject, adding it a `PlanetScript` component and setting it's position, velocity and mass.

2.3.2 PlanetScript.cs

`PlanetScript` is a script, which we added as a component to all bodies in the system. It stores the velocity and the mass of the body and allows us to implement movement logic, using methods `applyForce` and `assignForce`.

`applyForce` method uses Newton's Second Law to calculate the acceleration of a body, accordingly increases the body's velocity and changes it's position.

`assignForce` adds the gravitational force caused by another body onto this one. It does not apply the force yet, however, allowing the rest of the bodies' to affect this body as well. Once all of the bodies have affected one another, the force is applied.

2.3.3 PlanetManager.cs

This is the main part of our program which connects all of the other scripts. Here is where all of the settings are set. Options include type of computing we wish to be used (CPU, CPU with step skipping, GPU), settings for extra features (colors, collisions), etc. The project is initialised in the `Awake()` function and then updated in the `Update()` function.

`Awake()` function first initialises and calls a `PlanetSpawner` object, which reads all of the .csv data and spawns the bodies.

`Update()` function is responsible for calculating forces for all bodies and applying those forces, using methods from our custom-made script `PlanetScript`.

2.4 Improving performance in Unity

We can increase performance to simulate more bodies in real-time by:

- improving rendering preformance of objects in Unity and
- improving forces computation time.

2.4.1 Step skipping

The implementation without step skipping was able to simulate around 200 bodies in real time. We wanted to try what would happen if we made a simplification for the calculation of forces: Each iteration only a ninth of the bodies would have their forces recalculated. Other bodies would simply keep their current accelaration. This way every ninth iteration a body's acceleration is

calculated since the acceleration doesn't change that much in so few steps. Using step skipping we were able to simulate more bodies in real time at the price of losing computation precision.

2.4.2 Faster rendering

It is not enough that force computation is fast when performing real-time simulations. The rendering speed of objects on the screen must also be efficient. One way to improve it is to remove Unity rendering components that negatively affect rendering performance. These are mainly lighting and shadow systems. As we do not need them for our simulation, we disabled them.

We first rendered bodies in $\mathbb{R}^3$ as Spheres - primitive objects in Unity. However, we soon realized that default spheres in Unity are far too detailed for the purpose of being viewed from far. We implemented our own sphere mesh generator, for which we could control the resolution. Each mesh in Unity is represented by a set of vertices and triangles that connect those vertices. To create a sphere, we first defined an octahedron mesh with vertices on unit sphere. Then we divided each triangle into four smaller ones and normalized the vertices so that they were placed on unit sphere. We repeated this step a desired number of times, depending on the resolution we wanted. Described mesh at different resolutions is shown in Figure 1.

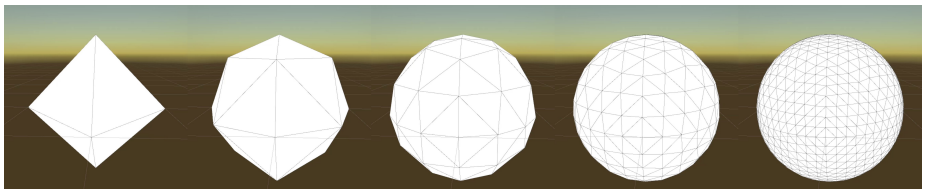

Figure 1: Our custom sphere mesh at different resolutions

Last improvement to rendering in Unity was to enable GPU Instancing in the sphere mesh renderer component. If this is disabled, a render call is executed for every mesh on the scene, which is very time consuming. Enabling this allows Unity to execute batch render calls of all objects that share the same mesh and same material, which greatly improves performance.

2.4.3 Computing forces on the GPU

To speed up computation of gravitational forces we can run the computations on the GPU instead of on the CPU. The main concept of GPU computing is parallelism - a big number of (simple) computations being executed in parallel. The n-body problem is well suited for GPU computation because it can be divided into smaller independent problems on each iteration.

Unity has built-in support for working with GPU computing using their `Compute Shaders` system. Shaders in Unity are written in High-Level Shader

Language (HLSL) and data can be passed between the CPU and the GPU using Unity's `Compute Buffers`.

First we declare a struct that holds position and mass of a body. It also needs to have space allocated for the result computed on GPU. We then create and array of Body structs.

```
public struct Body
{
    public Vector3 position;
    public float mass;
    public Vector3 force;
}
private Body[] bodies;
bodies = new Body[numBodies];
```

We must also define variables for computation in the shader. The code below is from the `GravityComputeShader.compute` script and is written in HLSL. We can see that it resembles code written for the CPU in C#.

```
float G;
int numBodies;
RWStructuredBuffer<Body> bodies;
```

Then we can write the main function of the shader. We must define the number of threads used by the GPU. Each thread has its own id, which we can use to determine for which body it needs to calculate the forces. We decided to use 100 threads. The downside of this approach is that we can only perform computations for systems that have $100n$ bodies, where $n \geq 1$. A solution to this problem could be to add dummy bodies to the system that are not displayed in the simulation.

```
[numthreads(100,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    float i = id.x;
    float3 force = { 0.0f, 0.0f, 0.0f };

    for (int j = 0; j < numBodies; j++) {
        if (i == j) continue;
        float3 distVector = bodies[j].position -
            bodies[i].position;
        float scalar = G * bodies[i].mass *
            bodies[j].mass / pow(length(distVector), 3);
        force += scalar * distVector;
    }

    bodies[i].force = force;
}
```

In our code in C# we must define a `Compute Buffer` for which we must specify the number of elements and size of each element. Then we can set the data of the compute buffer. We pass the `Compute Buffer` to the `Compute Shader` and set G and $numBodies$ variables in the Shader. We can then compute the enumber of thread groups we need on the GPU as $numBodies/numThreads$ and dispatch the shader. Finally, we read result that is computed on the GPU and apply the forces to the bodies.

```
// Create compute buffer for passing data to GPU
bodiesBuffer = new ComputeBuffer(bodies.Length, totalSize);
bodiesBuffer.SetData(bodies);

// Pass data to Compute Shader
gravityComputeShader.SetBuffer(0, "bodies", bodiesBuffer);
gravityComputeShader.SetFloat("G", G);
gravityComputeShader.SetInt("numBodies", bodies.Length);
gravityComputeShader.Dispatch(0, bodiesObj.Length /
    numShaderThreads, 1, 1);

// Read results
bodiesBuffer.GetData(bodies);
bodiesBuffer.Dispose();

// ... apply forces ...
```

3 Data generation

The process of data generation is a very hard task. Simply randomly generating data is of course easy, but due to laws of probability their combined momentum will be roughly 0 and the angular momentum of the system will also be zero. To generate data that has any sort of interesting behaviour, we had to intentionally construct an interesting starting point.

The way we decided to tackle this problem is by deciding, we are going to build two spinning galaxies that will collide with each other. We started by creating two large bodies of masses `m1` and `m2` at a certain distance `dist`. Then we populated the galaxies with bodies with lower masses on the interval `m-in-1 / 3` to `m-in-1` for the first galaxy, and `m-in-2 / 3` to `m-in-2` for the second galaxy. These bodies were on a radius from their initial heavy body by somewhere between 0 and `radius1` for the first galaxy and between 0 and `radius2` for the second galaxy. Another variable that needed to be assigned was the value of G.

From this data we had to somehow construct velocities, so that the behaviour of the system would be something interesting. We sent the two large bodies on a collision course towards one another with a small offset velocity, so they wouldn't be going directly towards eachother but rather just passing eachother on a very

close distance.

The velocity of the smaller bodies would be calculated as such: their base velocity would be the same velocity that their large body has - this ensures that the galaxy is moving together. Then a velocity for rotation would be added the following way:

We would create a unit vector that was perpendicular to the radius vector from the large body to this body. Then we would scale that vector by the magnitude, that was calculated in such a way:

$$\sqrt{\frac{G * m2}{specificRadius^3}}$$

m2 being the mass of the large body, and specific radius being the radius from the large body to this smaller body. This is the equation that describes the magnitude that ensures, that if only the large body and small body would be present in the system, the smaller body would have a circular orbit around the large body, This of course gets messed up due to the effects of all the other small bodies in the galaxy, but it is a good start for creating a consistent rotation. This had to be run a lot of times with different parameters to achieve any interesting examples. If the G was too big, everything just went together and then dispersed. If it was too small, the bodies just went on their own way. All of this made even harder by the fact, that the script had randomness in it, and so even the same parameters would sometimes yield an interesting example and sometimes a completely uninteresting one.

4 Results

4.1 Performace comparison between implementations

Up to this point we only qualitatively assessed how the simulation performs in different implementations. To paint a clear picture of how much we managed to improve performance, we timed how long a single iteration of the simulation takes in Python and in Unity.

In Unity, we timed 10 iterations for random initial conditions of bodies of 20 up to 5000, with a step of 20. We then computed the average of times for each number of bodies and plotted it on a line chart. In Python this was not possible as it performed much slower and such analysis would take too long. Instead we timed only 1 iteration for each number of bodies, and we did not time it for all numbers of bodies timed in Unity.

Figure 2 shows comparison of computation time between all real-time implementations. We can clearly see that our initial implementation in Python is much slower than the one in Unity. It requires around 10 seconds to compute a single iteration at only 1000 bodies, and around 3 minutes when simulating 5000 bodies. Even the most naive implementation in Unity that runs on CPU takes "only" 1.5 seconds for a single iteration when simulating a system of 5000 bodies, which is a huge omprovement over the Python implementation.

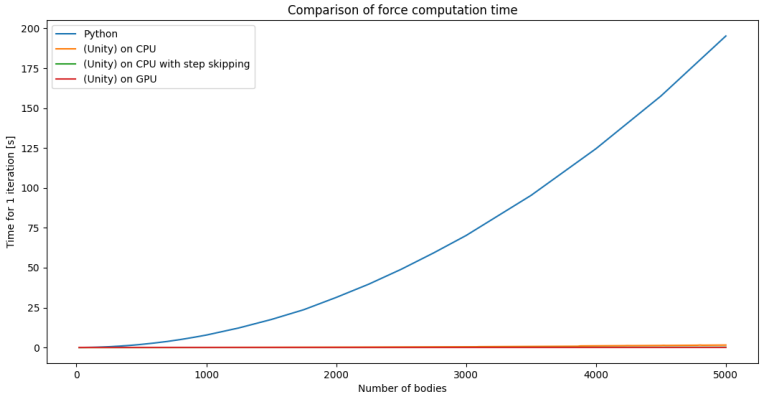

Figure 2: All computation times

We then plotted only Unity comparison times on a chart, since we can not clearly see how they compare in Figure 2. Figure 3 shows comparison of computation times for a single iteration between different computation modes in Unity - computation on CPU, computation on CPU with step skipping and computation on GPU.

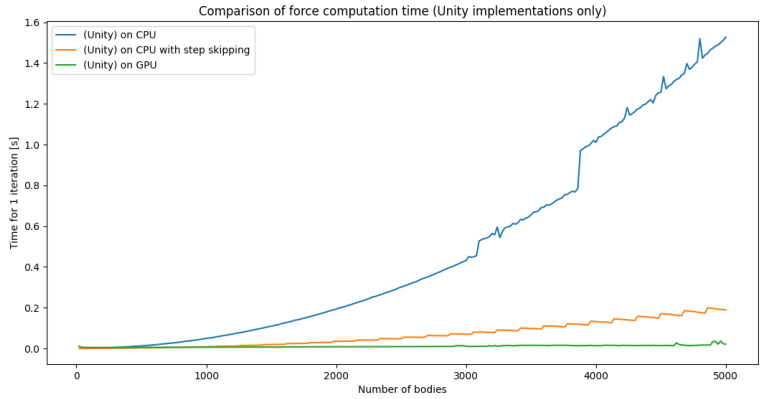

Figure 3: Unity computation times

We can see that the implementation with step skipping is roughly 9 times faster than the one without step skipping, which makes sense since only one ninth of forces are computed each step. The best performing implementation is, as expected, the computation on GPU in Unity. It only takes 0.02 seconds to evaluate and apply forces between 5000 bodies, allowing us to run 50 steps per second in real-time.

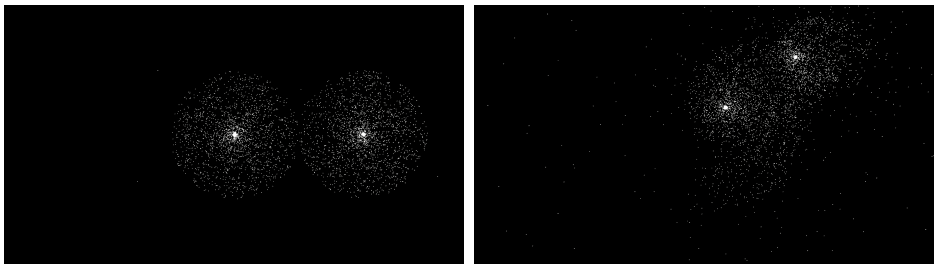

Figure 5: Simulation of two galaxies

4.2 Simulations

As described above, we generated two main types of initial conditions. The first one is just a single galaxy in which lighter bodies circle around a heavier body in the center of the galaxy. The initial conditions of this system and its state after many iterations can be seen in Figure 4.

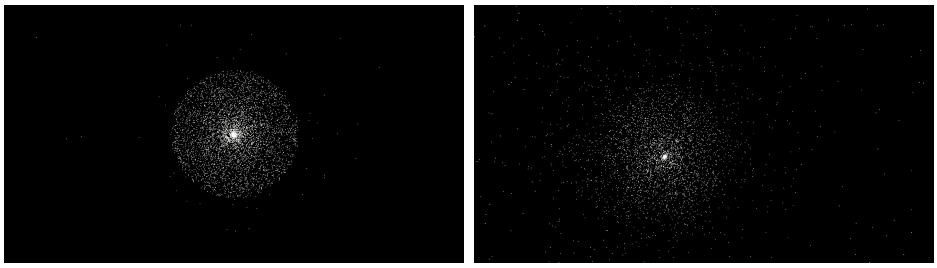

Figure 4: Simulation of one galaxy

A more interesting simulation of two galaxies is shown in Figure 5.

5 Extra Features

5.1 Export simulation to video

We implemented a feature that allows us to run the simulation in non-real-time. If export to video option is enabled, then a screenshot is captured each iteration of the simulation. These screenshots are saved in a designated folder and in the end merged into a video using `ffmpeg` - an open source tool for working with videos.

5.2 Coloring bodies

We decided to add color to the bodies, to visually represent their velocities. The faster the body is, the more red it contains and the slower it moves, the more

blue it's color is. During the development phase of our project, we discussed and implemented two different ways of coloring the bodies - based on a velocity of a body and based on a velocity of the body relative to other bodies. We chose the latter in the final version of our program.

We decided for coloring to happen every 50 time units (this was modifiable in the settings of a project). We did not want coloring to happen too frequently (it would only take resources from the program), nor did we want the coloring to happen too rarely (would appear slow and sluggish).

5.2.1 Coloring based on a velocity of a body

This was the first implementation. The colors in Unity are represented in the RGBA format. The colors were set in the following manner. `red = min(len (velocity vector), 255), green = 0, blue = 255 - red, alpha = 255`.

The issue with this approach is that once the system would be up and running for some time, all of the bodies would be moving so fast they would be red and there would not have been a clear way for us to distinguish between their velocities.

5.2.2 Coloring based on a velocity of a body relative to other bodies

We decided to use this approach as it would allow dynamic changes to the colors of the bodies. Instead of having the magnitude of the velocity vector compared to 255, we saved the minimal and the maximal magnitudes of the system upon every 50 time units and then compared the magnitude of each body to those values, to see how large it was compared to them. If the magnitude of the velocity vector would be closer to the minimal magnitude, the body would be more blue and if it were closer to the maximal magnitude it would be more red.

5.3 Collisions

We decided that the collisions should follow the law of conservation of momentum. We were dealing with a closed system, meaning the total momentum in the system should remain constant. As we wanted the spheres to visually represent planets in space rather that pool balls, we decided to implement inelastic collisions between the bodies.

Due to conserving total momentum and inelastic collisions, we can use the following equation

$$m_1 v_1 + m_2 v_2 = (m_1 + m_2) v$$
$$v = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2}$$

We assigned the new velocity and joined mass to one of the bodies and set the other other ones' to zero, effectively deleting the body from the simulation. From that point on, the body with the mass of zero was skipped entirely, allowing our simulation to speed up over time, having to do less comparisons between the bodies upon each collision.

5.4 Sizes of bodies

The size of the body is the sixth root of it's mass. This was done, so we can visually distinguish between bodies of various masses. The root of the mass had to be large enough to shrink the centers of a galaxy to appropriate size (due to their enormous mass) and also small enough to ensure not all bodies would look the same.

6 Standalone app functionality

Up to this point we performed all simulations directly in the Unity editor environment. To finish our project we decided to build an executable file from the Unity source code. By itself this is not a problem since Unity offers support for compiling their code to a executable for a chosen platform, but we had to make some changes if we wanted the simulation to be runnable as a standalone application.

We added a title screen along with a settings menu where you can tweak simulation parameters. We also had to add a way for selecting a file containing initial conditions. We created a folder in which we put all of initial conditions csv files that we have generated. The application then reads files in this folder and prompts the user to select one. After a file is selected, the simulation starts.

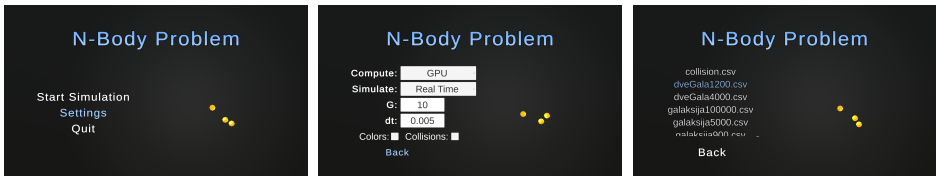

Figure 6: Title menu

The title menu is shown in Figure 6 in the following order: main menu, settings menu and csv file selection menu. Additionally, we also added camera movement using WASD keys and mouse to look around.

7 Work division in group

Since we implemented the simulation of the system in three different environments, it made sense that each one of us implemented the basic simulation

once to get more familiar with the problem. Janez implemented the first basic simulation in Python, Matevž in Julia and Nace programmed the initial Unity implementation.

After we had the basic implementation working correctly, Matevž experimented with step skipping in the Unity implementation to improve performance. After that he started working on data generation, mainly focusing on finding interesting initial conditions.

Janez worked mainly on improving the performance in Unity by making rendering faster using a custom sphere mesh and batching render calls and by implementing the computation of forces on the GPU. Then he added the "Export to video" feature to allow non-real-time simualtions. When the project was finished, he implemented the title screen and camera movement.

Meanwhile, Nace worked on extra simulation features. He implemented coloring of bodies based on their velocity and resizing them according to their mass. He also implemented collisions that follow the law of momentum conservation.

We wrote the report together, each one of us writing the chapters on which he mainly worked throughout the project.

8 Sources

- The n-body problem Wikipedia page
- Newton's second law
- Newton's law of gravity
- Introduciton to compute shaders in Unity (video)
- High-level shader language (HLSL) documentation
- Conservation of linear momentum Wikipedia page