The Fast Multipole Method Applied to the N-Body Problem

Erkin Verbeek

May 11, 2024

Introduction/Motivation

The n-body problem is a fundamental challenge in physics that aims to calculate the interactions between n objects under various forces. Example forces include fluid dynamics and electrostatic forces. For the purposes of this report we will be focusing on the gravitational forces that all astronomical objects exert on one another.

Knowing the gravitational forces being exerted on any given body in space allows us to model the movement of the body over time. If our system has two bodies, then we can predict the future positions of each body exactly. With two bodies, the gravitational forces between them are equal in magnitude and opposite in direction, pulling each other towards a mutual center. With any preexisting lateral movement, the bodies rotate around their mutual center, resulting in what we call an orbit. Any system with two bodies has already been solved exactly, that is, we can calculate the positions of the bodies at any time in the future with exact precision. An example of a two-body system can be seen in figure 1.

However, if you've ever glanced up at the night sky you might have noticed more than two objects. Unfortunately, any system with more than two bodies (excluding a few specific scenarios) is not known to be solvable exactly. Instead of solving for positions exactly, we turn to simulations to get approximate positions. We can exactly calculate the gravitational forces between any two bodies, and then sum over all the pairs in a given system to get the total gravitational force being exerted. With the gravitational force being known, we can use time-stepping methods such as the Euler Method to approximate any future state of the system. An example of a three-body system can be seen in figure 2.

The issue with the method described so far is its scalability. Iterating over every single pair of objects in a system requires $O(n^2)$ iterations. Celestial phenomena such as galaxies can include upwards of a billion stars in them, immediately ruling out any $O(n^2)$ methods from being feasible. The answer to how to reduce this scaling issue is the Fast Multipole Method (FMM).

The Fast Multipole Method "arguably provid[es] the first numerically defensible method for reducing the N-body problem's computational complexity to $O(n)$" [1].

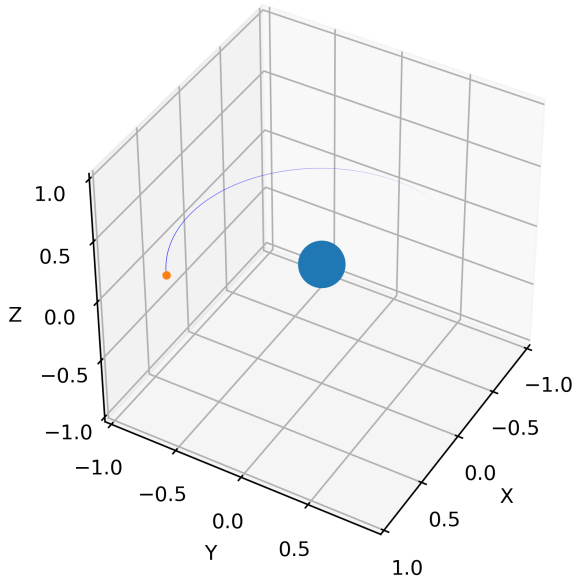

Figure 1: A simulation of Earth and the Sun in orbit (sizes not to scale).

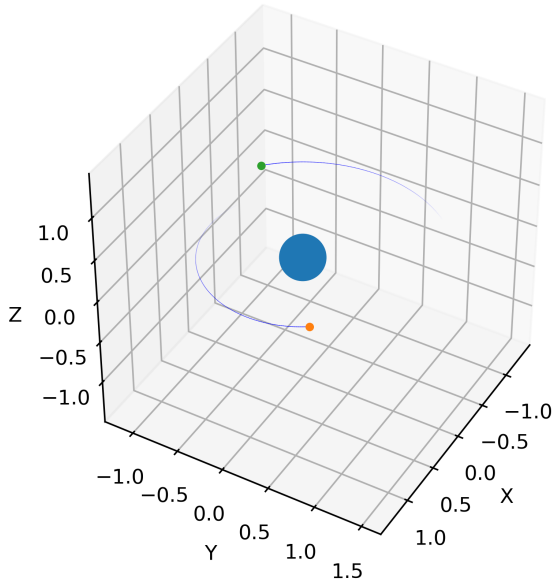

Figure 2: A simulation of three bodies in orbit. Each body is exerting a force on every other existing body.

1 Fast Multipole Method

In our n-body simulations, we use Newton's Gravity equation to calculate the forces between any two bodies in a given time step. The equation is as follows:

$$F = G\frac{m_1 m_2}{r^2}$$

where $G = 6.674 \times 10^{-11}$ is the Gravitational Constant, m_1, m_2 are the masses of the two participating bodies, and r is the distance between them.

The Fast Multipole Method takes advantage of the fact that the calculated gravitational force scales with $1/r^2$. This means that as two bodies get further and further away from each other, the gravitational force exerted between them becomes exponentially smaller. FMM takes advantage of this by performing *estimate* calculations when groups of bodies are far apart instead of calculating every single inter-body force exactly.

FMM requires that the entire 3D space be partitioned into smaller blocks. At every time step in my simulation I first get the max and min x, y, and z values so that I know how large of a space I need to partition. These max and min values can change per time step because objects are constantly moving closer or further apart. After knowing my 3D boundaries, I recursively partition the 3D block of space into smaller and smaller blocks. I partition the entire space into 8 smaller equally sized cubes, then each of those 8 cubes gets partitioned into 8 smaller cubes, and so on and so forth. The depth of this recursion is defined by a single parameter: NUM_LAYERS. Modifying this parameter allows for tuning how accurate I want the FMM simulation to be, at a trade off of longer run times. More partitions leads to higher accuracy, but more complexity in terms of calculations. In my simulations I kept NUM_LAYERS equal to 3.

Once we have partitioned the space into blocks, we then group together all objects that exist in the same block. The reason we partition and group the objects is so that we can easily identify all the objects that are close together (those in the same or neighboring blocks) and those that are far apart (everything not in the same or neighboring blocks). In the direct method, if two groups of bodies are far apart, then we would perform one force calculation for every unique pairing of bodies. In FMM, we average the masses of all the bodies in each block, and then perform only a *single* force calculation between two distant blocks, for each pair of blocks. Figure 3 demonstrates how FMM reduces multiple calculations down to one by grouping distant objects together.

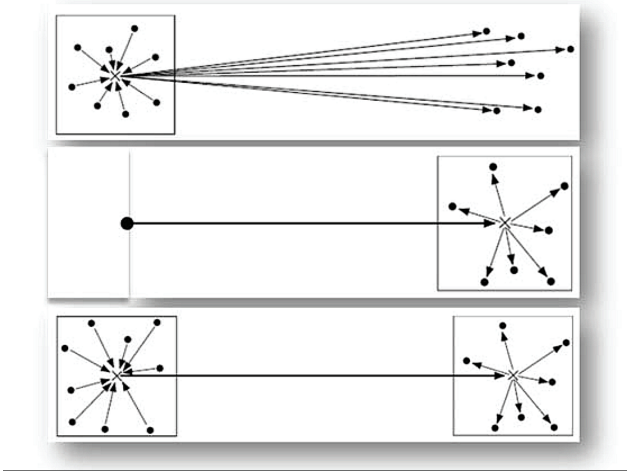

Figure 3: A representation of the Fast Multipole Method [1].

This approximation only works for objects that are far apart. So even in the FMM case, if two objects are in the same block or in neighboring blocks (i.e. close together), then we perform direct force calculations. When two objects are close together then trying to approximate their forces would lead to more noticeable errors. Therefore, to summarize, we perform the approximate force calculations for objects far apart, and the direct force calculation for objects close together.

2 Results

Using both the direct method and FMM we are able to successfully simulate any number of bodies for any given number of time steps. In Figure 4 you can see a simulation that was run with 16 bodies of varying mass, location, and velocity.

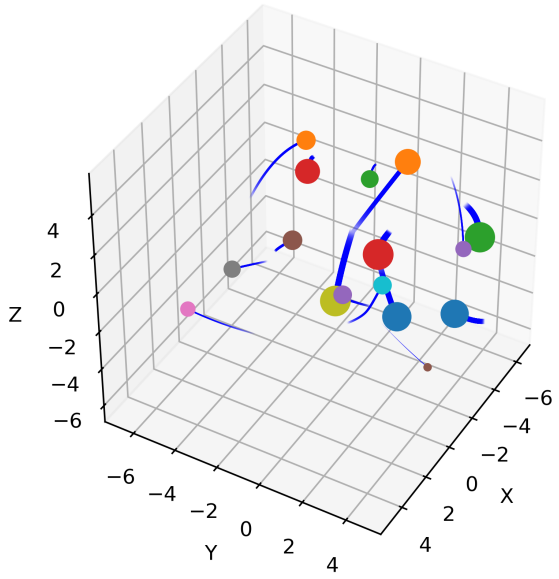

Figure 4: A simulation of 16 bodies of varying mass, location, and velocity.

2.1 Scaling

In order to observe how the direct method and the Fast Multipole Method scales with n, I created multiple scenarios with varying values of n and timed how long it took each method to run for 100 time steps.

For each method and each value of n I ran three simulations with random initial conditions. Each of the three simulation ran for 100 time steps. I then discarded the slowest of those three timings (to help account for CPU variations) and averaged the remaining two results. The results for these timing tests can be seen in Figure 5. Immediately it is clear to see that the direct method scales much worse than the Fast Multipole Method. This makes sense as the direct method is $O(n^2)$ while FMM is shown to be nearly linear.

Another interesting result to see is that the direct method is faster than FMM for small enough n (in our case, the tipping point appears to be around $n = 12$). The reason that the direct method is faster for small n is because at every time step FMM goes through the process of partitioning the 3D domain into blocks of equal sizes, and then assigning each object to a block in that partition. This partitioning is required for FMM to work, but has a slight overhead associated with it. Fortunately however, this overhead quickly becomes worth it as n increases to larger values.

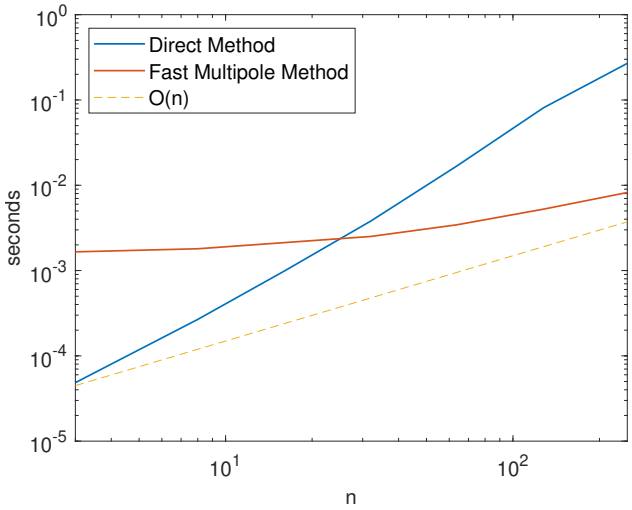

Figure 5: Scaling results as n increases. (linear line drawn for reference)

3 Conclusion

FMM provides significant performance improvements for n-body simulations when n is large enough. In complex astronomical simulations, n can be upwards of a billion. The approximation errors that come from using FMM instead of the direct method can be mitigated by tuning the layer-depth used when partitioning the domain. And any errors left over are on the same magnitude as those gathered from whichever time stepping scheme is used, implying that no meaningful errors are observed when simulating with FMM over the direct method.

References

[1] Bela Erdelyi. "The fast multipole method for N-body problems". In: *AIP Conference Proceedings*. Vol. 1507. 1. American Institute of Physics. 2012, pp. 387–392.

4 Appendix

4.1 Implementation

All the code used for this project can be found at this GitHub repository.

All the simulations were done using C++. All data was saved to HDF5 files. Python was used to plot visualizations. And Docker was used to containerize each runnable segment of code (containerized into images built from Alpine Linux).