CTA200H1 – Project Report
N-Body Simulation
Karandeep Basi
May 19, 2017

For this project I implemented an N-Body simulation for gravitationally attracted particles.
Source code: https://github.com/singhkaran/basi_karandeep_cta200_project.git

Approach

I used a tree-algorithm in order to implement the simulation.

The algorithm is outlined in section III.2.2.4 of Ott and Benson's paper.

Essentially the algorithm repeatedly partitions the system into smaller partitions until the smallest partitions are particles, storing the partitions in a tree structure. In order to calculate the force applied to an individual particle we recursively search through the tree, if the current node meets the opening angle criterion ((size of cell)/ (distance to center of mass of cell) < theta) then we calculate the force on the particle, otherwise we continue recursively searching through the tree.

This algorithm takes advantage of the fact that distant groups of particles apply a similar force regardless of minor angle differences. Thus reducing the number of calculations we make.

In order to make the more algorithm more tractable (and manageable for my laptop) I made an accuracy/speed tradeoff where I enforced a minimum partition size (i.e a partition must have at least 32 particles). I then calculated the force on every leaf node using the same method described above and applied this force to all particles within that particular leaf node. This had a dramatic speedup (an order of magnitude faster). This tradeoff can of course be modified by changing the minimum partition size (a minimum size of 1 would be the original tree algorithm).

An additional speedup I could have taken advantage of in the future was using parallel processing. Due to the recursive nature of this algorithm it is a prime candidate to be optimized via parallel processing.

Problems

One issue I faced was the "slingshot issue". This happened as two particles accelerate at an increasing rate as they approached each other. Since the timesteps weren't small enough to effectively accelerate the particles back towards each other they would simply fly past each other.

I attempted to manage this via softening. I applied an epsilon to the divisor in the force calculation which would minimize the acceleration when two particles were nearby so they did not accelerate excessively.

This approach had some success, however, as the inputs changed (i.e a 10,000 particle simulation vs a 1000 particle simulation) the softening epsilon had to be changed in order to adapt to the new rates of acceleration which are now far greater. I also managed to eliminate the issue when particles had no initial velocity, but when I introduced them again it completely changed the demands of the system.

Another approach I could have taken was to reduce the timestep for particles which were in danger of accelerating too rapidly. The paper outlines an approach in which the local particle density is used to identify these particles. Since I used the tree algorithm, it would be cheap computationally to calculate the local density of individual particles so this seems promising as a feasible approach.

Two other methods which I have not looked into deeply at this time are collision handling and adaptive smoothing (changing the epsilon based on the input) which would help solve this problem.

Example output

(Green = high mass, Blue = lower mass)

https://gfycat.com/AntiqueMammothGangesdolphin?speed=4

1000 particle simulation, the particles slingshot back and forth between each other but lean towards the left. I believed this was simply the slingshot issue, however Professor Norm pointed out that the initial velocities might not sum to zero which would explain why they do not collapse in the center of mass. I found that this was the case and adjusted the velocity of every particle to add to zero, but I unfortunately the slingshot issue still dominated the issue.

https://giphy.com/gifs/3ohzdKFLbJB4tcyGUU

10,000 particle simulation (same issues)

60,000 particle simulation
https://gfycat.com/GranularFortunateCow?speed=2

Animation

The method I used to create these animations was fairly crude, I could not find a suitable library to animate 3d systems (matplotlib seemed to only support 2d animations?). So, I simply plotted

each step of the simulation, saved the image, and then created a gif of the series of images and used a library to auto upload them to gfycat.