GRAVITATIONAL N-BODY SIMULATION

JOE LANDERS

"A computer will do what you tell it to do, but that may be much different from what you had in mind." – Joseph Weizenbaum

1. Introduction

Gravitational N-Body simulations are used to model systems of particles in space. The system can be anything from a solar system with a few planets to a model of the universe with tens of billions of "particles" each representing a billion solar masses. The simulations are useful, because general analytical solutions are not known for these systems. Numerical integration methods introduce challenges of their own, however, and our interpretations of the simulation must take into account its chaotic and inexact nature.

2. The Physics and Math

The magnitude of the gravitational force exerted on one body by another is given by the familiar equation, $F = G\frac{m_1 m_2}{r^2}$. Since I wasn't interested in simulating an actual, physical system, I'm free to ignore the factor of G. Furthermore, I restricted myself to the special case where all of the masses are equal. I also introduced a softening factor (discussed in more detail later) to limit the magnitude of close-range forces. Putting these together yields an equation for the magnitude of the acceleration of one mass due to another, $a = \frac{1}{r^2+s^2}$, where I've expressed the softening factor as s^2 to invite the comparison to r^2. It is interesting to note at this point that the force we are simulating is no longer necessarily the gravitational force, but any inverse-square force.

So far, I've ignored that the force and acceleration are vector quantities. It will help to find a form of the acceleration equation which gives us each of the components of the acceleration separately. With a bit of algebra,[1] we can find the following vector equation for the acceleration of a body, i, due to another body, j: $\vec{a_{ij}} = \frac{\vec{r_{ij}}}{(r_{ij}^2+s^2)^{\frac{3}{2}}}$. This form suits us quite well because we can solve for each of the acceleration components independently. This will let us easily extend the code to any number of dimensions (though in practice, we will probably concern ourselves mostly with the two- and three-dimensional cases!). To get the *total* acceleration of a body, i, we simply sum the previous equation over each of the N bodies:[2]

$$\vec{a_i} = \sum_{1 \le j \le N} \frac{\vec{r_{ij}}}{(r_{ij}^2 + s^2)^{\frac{3}{2}}} \tag{1}$$

[1] See *GPU Gems 3* by NVIDIA.

The numerical integration is handled by the Verlet method, which takes the form

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n\, dt^2. \tag{2}$$

This is a convenient method because the code doesn't need to worry about velocity (after the initial velocity). The only slightly tricky bit is shuffling around the previous, current, and next positions.

3. The Softening Factor

If we omitted the softening factor, the acceleration would get asymptotically large as the separation between two bodies approaches zero. This is undesirable for at least two reasons: first, because the masses we are simulating are not actually point-particles, they can't get vanishingly close to each other, and second, the error in our integration step gets larger as the velocity gets larger. Examining the first reason, we can reason that the closest that two identical spherical masses can get to each other is $2r$, where r is the radius of the masses. So if we think of our softening factor, s, as it appears in Eq. 1, as the diameter of our masses, we can (loosely) think of the model as representing (non-collisional) spheres of constant density. This is a decent model if we want our bodies to represent galaxies, because galaxies are mostly empty space and largely pass through each other when they collide. The second reason that the softening factor is important is apparent if we run the simulation with a small or zero softening factor. When two bodies pass very close to each other, the acceleration gets very large, and they shoot off in opposite directions, never to return. This happens because the large acceleration says that the two bodies will be very far apart during the next timestep, and the next acceleration will be too small to pull the bodies back together. If we do want to simulate point-like masses, we have to use a smaller timestep to avoid the case of the masses shooting away from each other to infinity.

4. The Timestep

Choosing a "suitable" timestep is a nebulous endeavor. This is largely because the error of the system is hard to determine. Even at quite small timesteps (where the visualization is perceptibly quite sluggish), the systems with two slightly different timesteps diverge from each other very quickly. Of course, when we are simulating a chaotic system, we do not expect to get exact positions at distant points in time. For our purposes, it is acceptable that the simulation be "qualitatively" correct—that is, the pattern of trails that is produced on the screen is a physically plausible case. To check the quantitative accuracy of such a simulation, one could keep track of the kinetic and potential energies at each timestep, and verify that the total energy stays constant over time. This would require some modification to my code, as I discarded the gravitational constant and the masses in my equations, so

[2]It is apparent from this equation that $\vec{a_{ij}}$ is equal in magnitude and opposite in direction to $\vec{a_{ji}}$, so a slight modification of the code could take advantage of this fact and reduce the number of calculations by a factor of about two. However, some quick testing showed that the slowest part of the code, by quite a bit, was the pixel plotting method, so I chose to not bother optimizing the physics calculations. Furthermore, the current version is probably more readable and understandable than the optimized version would be.

I can't simply add up the kinetic and potential energies. It is already known that Verlet integration doesn't conserve energy, but it would be of some interest to measure how the energy changes with time. However, it suffices in my simulation to choose a timestep small enough that the energy doesn't increase perceptibly.

In playing with the simulation, it is quickly apparent that a suitable timestep is dependent on the value of the softening factor. The larger the softening factor, the larger the timestep can be, and the quicker the simulation will be. I attribute this to the effect that the softening factor has on the maximum acceleration of a body. A larger softening factor reduces the maximum acceleration, and so a larger timestep is sufficient. Similarly, as the number of bodies increases (meaning the total gravitational force increases), a smaller timestep is required. Some more complex simulations use a variable timstep so that fast-moving particles get a smaller timstep than slow-moving particles.

5. Initialization

One of the biggest challenges was finding a good way to give initial positions and velocities to some number of particles in some number of dimensions. I wanted the initial conditions to be random, so that we get a different system each time the code is run, but I also wanted the initial conditions to be interesting. My code first generates initial positions in the "middle" of the screen (from $\frac{1}{4}w$ to $\frac{3}{4}w$, for example). Then it calculates the initial velocities by multiplying the corresponding position component by a scaling factor and an alternating sign. The scaling factor is randomized within some range that I found appropriate for keeping the particles from flying off the screen. The alternating sign is to add a bit of "swirliness" to the initial velocities. For example, in two dimensions a particle with position (x, y) gets a velocity in the (x, -y) direction. This is most useful in the two body case, where an absence of "swirliness" would result in the two bodies remaining on a straight line. With this method, they have some angular momentum, and the resulting orbits are more interesting.

Another important part of the initialization procedure involves getting in the center of mass reference frame. While assigning initial velocities and positions, the code adds up the positions and velocities in each component, and at the end divides by the number of bodies. It then steps through all of the positions and velocities again and subtracts the center of mass component from each one. This has the effect of putting the window in the center of mass reference frame, with the center of mass position in the middle of the window.

6. Repeatability

It is usually desirable to be able to repeat a simulation with the same initial conditions at a later time. In my code, the pseudorandom number generator is seeded (with the 1970 epoch time) just before the initialization function is called. So in order to generate the same set of initial conditions at a later time, it is only necessary to save the seed time and reseed the PRNG with that value later. I wrote my code so that, during a simulation, I

can hit the "s" key, and the program writes out to a file the command that will reproduce the current simulation. It also writes out a screenshot for reference.

I made use of this functionality when I wanted to play with the values of timestep and softening factor. I could keep the same initial conditions, make a slight change to the timestep or softening factor, and compare to the screenshot to see approximately when the simulations diverged.

7. Implementation

The program was written in C, and the graphics were produced with the Simple DirectMedia Layer (SDL) library. I chose C because of my familiarity with it, and because the organization of the program was simple enough that I could forgo the Object Oriented facilities of C++. Of course, the program got more and more complex, and I ended up passing quite a lot of arguments between functions when it might have been better to encapsulate some of the mess! However, the program is short enough that I think it's still pretty comprehendible. This was my first project involving graphics in C, so I chose SDL based on a few references which recommended it. The library is available on the major operating systems, and it was pretty straightforward to use. It would probably have been more fun to have the graphics in 3D with OpenGL, but that looked like quite a bit more work.

The program itself is just one file of under 300 lines of reasonably commented code, partitioned (I think) into a digestible number of functions. The only dependency is SDL, so it's pretty easy to compile across platforms.

Naturally, there are a few things that could be made better, but I didn't think were worth the time. Keeping the bodies gravitationally bound within the window is a bit dependent on the number of bodies, the size of the window, and the softening factor. It's a bit kludgy that the bodies' positions are doubles initialized within the range of the window's pixels, and then cast as integers when it comes time to plot them. It would be better to make a physical-to-pixel conversion so the physical positions aren't limited in this weird way. This would let me get rid of the resolution-dependent behavior. In practice, it isn't a big problem as I keep the window size within a small enough range that the weird behavior isn't very noticeable.

I mentioned in a footnote earlier that the number of acceleration calculations could be halved, since the acceleration of particle i due to particle j is equal and opposite to the acceleration of particle j due to particle i. I chose not to bother with this optimization because the pixel plotting was by far the slowest bit of the code. An optimization to the pixel plotting part of code would be to plot more pixels at a time. Instead of advancing each body by one timestep and plotting one more pixel for each body, I could advance each body by several timesteps, and plot the resulting pixels all at once. A good number of timsteps per plotting step would depend on how fast the code was running. If the code is running quickly, I could do more timesteps between plot steps. If the code is running slowly, there would have to be fewer timesteps between plot steps to avoid the visualization getting too sluggish.

8. Interface and Design

It was a goal of mine to make this program nice enough to use and maintain that I could come back to it later and play with it or expand it. I wanted it to be easy to use, while still being informational. I think I achieved these goals, as I spent quite a few hours just playing with the program. The program is pretty easy to use once you come up with some nice-looking parameters. Once it's running, it's easy to re-initialize with a single keypress, or to save the current simulation with another single keypress. It can be addictive.

9. Conclusion

The final program is certainly entertaining, and I think it provides some insight into the physics of N-body systems. Namely, it is apparent that the systems are chaotic, and very sensitive to initial conditions and parameters such as timestep and softening factor. One of the more striking realizations that came with building the program and playing with it is how such a complex system can arise out of a relatively simple equation.