N-BODY SIMULATION: MODELING MONOLITHIC COLLAPSE

Trevor "*BOOYAH*" Picard
Department of Astronomy, New Mexico State University, Las Cruces, NM 88003, USA

ABSTRACT

We test the monolithic collapse scenario of galaxy formation using collisionless n-body methods implemented in C++. Assuming a star formation timescale much shorter than a dynamical time, we approximate the collapse of a primordial gas cloud as an initially randomly distributed, spherically symmetric system of 30,000 particles with $M_{tot} \sim 10^{12} M_{\odot}$. However, due to time constraints the model was downsized to a mere 300 particles, which significantly reduced its accuracy in most respects. The initial configuration has $R_i = 4$ kpc, and the particles were given velocity dispersions of $v_{rms} = 10 \pm 1$ km s^{-1}. We let the system evolve and relax, monitoring the physical configuration and *rms* velocity & mass density profiles of the resulting "elliptical galaxy". Comparing to observations and previous studies involving collapse simulations, our model holds up well in some respects, and mediocre in others. The density profile doesn't even come close to following the broken power law we expect from observations, with $\rho(r) \propto r^{-1}$ in the inner few kpc and $\rho(r) \propto r^{-4}$ in the outer regions. Our broken power law shows the opposite trend. In addition, the system expands by several orders of magnitude during the integration, which is in direct conflict with previous simulations. The general qualitative features of monolithic collapse reflected in the results, although we conclude that these features don't necessarily reflect reality across the board. Improvements should be made on the number of particles, the integration time step, and the time allotted for project completion.

Keywords: n-body, simulations, numerical models, galaxy formation

1. INTRODUCTION

In 1962 the idea of galaxy formation via the monolithic collapse of a gas cloud was proposed by Eggen, Lynden-Bell & Sandage. Monolithic collapse was a popular theory amongst astronomers, and it continued to be a plausible and commonly cited potential galaxy formation process for many years. However, it's known today that the predictions of monolithic collapse don't match observations within the currently accepted ΛCDM paradigm. In spite of being incorrect, monolithic collapse is physically simple enough to model in a relatively straightforward manner, thus serving as an interesting case study regarding the numerical methods of elliptical galaxy formation.

Monolithic collapse in this particular case is based on the assumption that the timescale of star formation in the initial gas cloud is much less than a dynamical time. In other words, an elliptical galaxy will form if the gas cloud fragments and forms individual particles long before the mass has time to collapse and orbit about the center of mass. The result is trivially simple initial conditions for the system; models of the monolithic collapse scenario start as a spherical, uniformly dense distribution of N particles. We can estimate the dynamical time for the resulting particle distribution using

$$t_{dyn} \sim \frac{R_i}{v_i}, \tag{1}$$

where R_i is the initial radius of the spherical distribution and v_i is the initial velocity of a particle. Based on the dynamical timescale we can roughly approximate the time it would take for a system of N particles to relax, which is

$$t_{rel} \simeq \frac{N}{8 \ln N} t_{dyn} \tag{2}$$

Studying the properties of a simulated elliptical galaxy post-relaxation can potentially shed light on the validity of the monolithic collapse scenario, the accuracy and thoroughness of the underlying physical assumptions, and the basic concepts of n-body simulations.

Table 1
Model Parameters

Parameter	Model Value
Number of Particles	30,000
M_{tot}	$10^{12} M_{\odot}$
Initial Radius (kpc)	4
Initial Velocity (km s^{-1})	10

Observations of elliptical galaxies are quite extensive, and from previous studies we know roughly what results any accurate numerical simulation should produce. We expect the central velocity dispersion of the galaxy to be fairly high post-relaxation, roughly ~ 200km s^{-1}. The central region of an elliptical galaxy should be relatively compact ($\sim 2-5$ kpc), with a density profile following a shallow power law given by $\rho(r) \propto r^{-1}$. The surrounding stellar halo should have a steeply declining density profile, best approximated as $\rho(r) \propto r^{-4}$.

2. METHODS & TESTS

In order to test the monolithic collapse scenario, we start with a uniform spherical distribution of $N = 30,000$ particles. The actual number of particles needed to realistically model an elliptical galaxy is far higher, so for our simulation $N \ll N_*$, making this a collisionless system with a theoretically smooth density distribution.

The physical parameters of the simulation are summarized in Table 1. Typical elliptical galaxies have masses $\sim 10^{12} M_{\odot}$, so this is the mass we assign to the entire system, evenly distributing the mass between the 30,000 particles in the simulation. Radii of elliptical galaxies fall within 500 pc $\lesssim R \lesssim$ 100 kpc, so choosing an initial

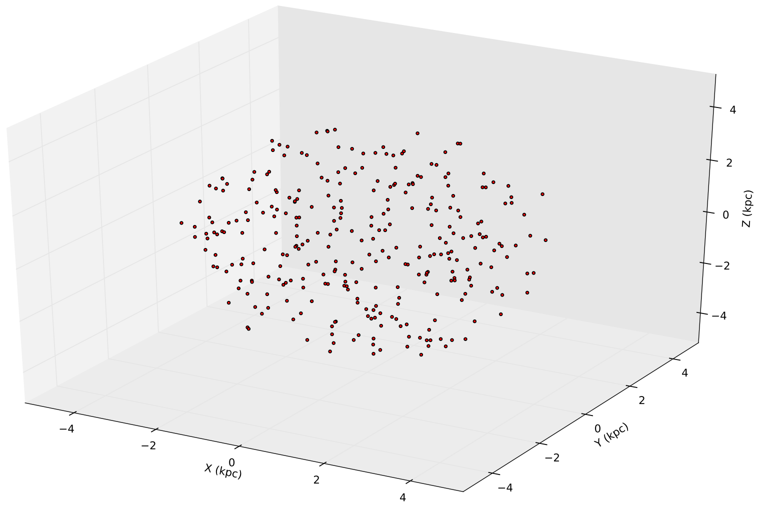

Figure 1. The initial configuration of the 300 particle simulation.

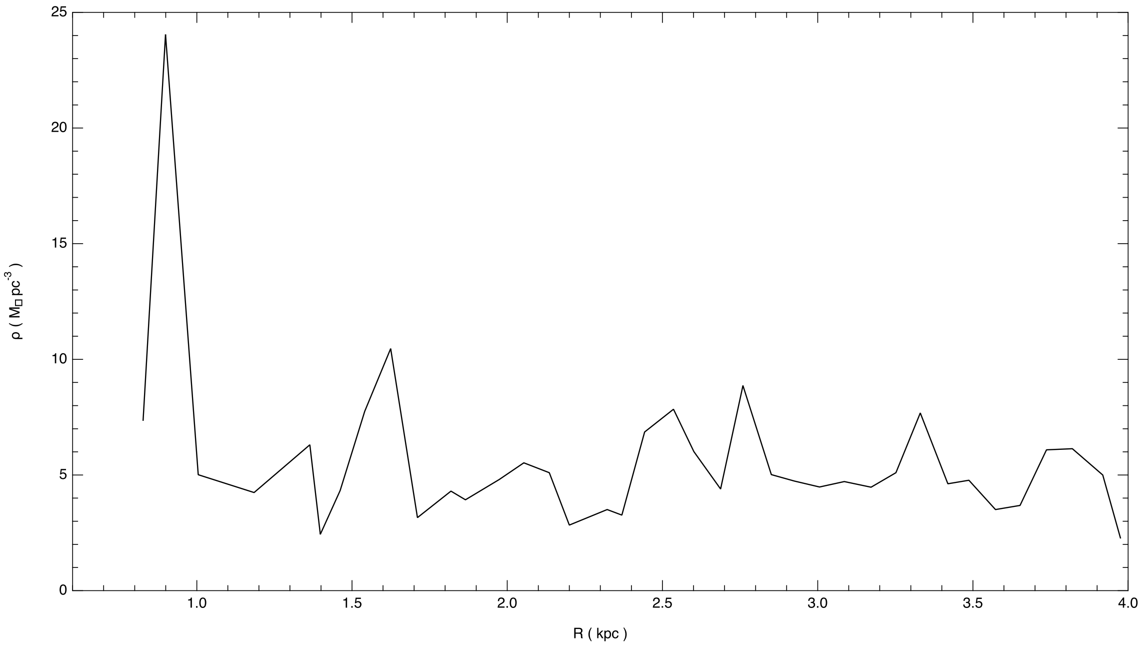

Figure 2. The initial density profile of the $N = 300$ simulated galaxy. The profile is split into 50 radial bins.

radius of 4 kpc seems to be reasonable.

Modeling monolithic collapse comes with benefits and tradeoffs in the time domain. Only a half dozen or so dynamical times are required for the system to relax, allowing the total simulated time to be relatively short compared to many other dynamical systems. However, the temporal resolution required for an accurate simulation is much higher, so we choose a timestep of $dt \sim 85,000$ years.

In our simulation we use the leap-frog method of integration, which has been shown to conserve the total system energy well over a large number of dynamical times. We also use Plummer force softening to stabilize the numerical integration, hopefully avoiding enormous accelerations between particles that come very close to one another during the simulation. To first order, the leap-frog method is simple and likely effective for the purposes of our simulation. However, this integration method comes with an annoying numerical artifact: the system's center of mass drifts significantly away from its starting point as it evolves. We re-calculate the center of mass at each time step and re-center the particle distribution to counteract this effect, although unfortunately this comes with the cost of being computationally inten-

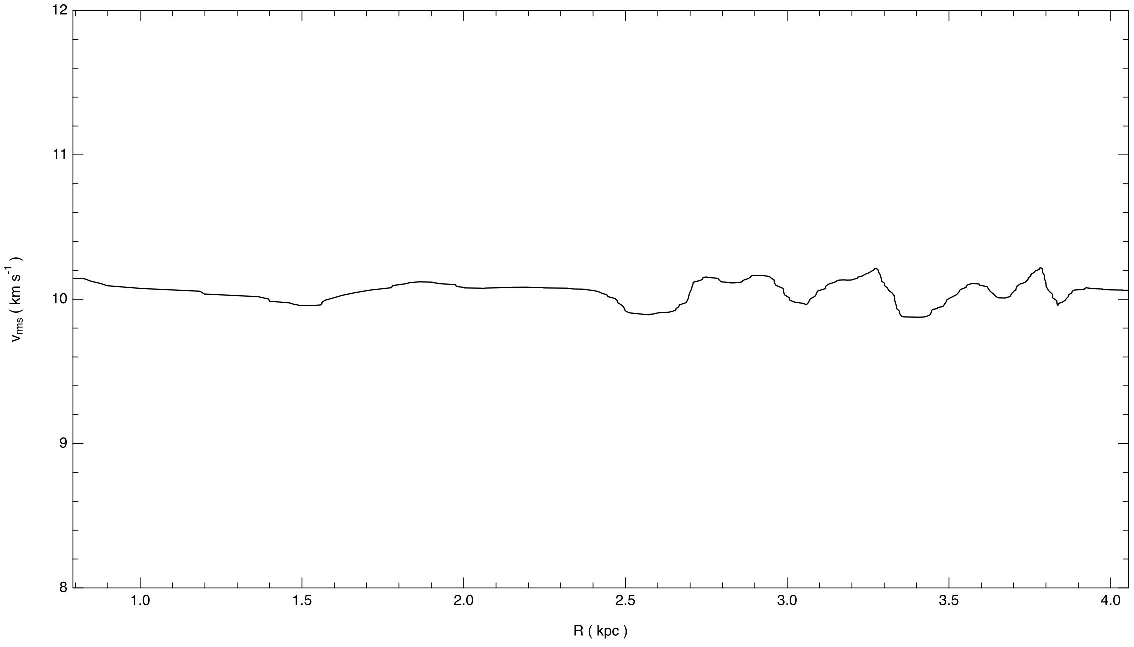

Figure 3. The initialv_{rms} profile of the $N = 300$ simulated galaxy. The profile is a result of a binomial smoothing algorithm with smoothing parameter 100.

sive.

Thankfully C++ is compatible with Open MP, so we take advantage of the multi-core machines available to the department to run many of the computationally intensive loops in parallel. In particular, we use the *milkyway* machine, which boasts a 64-core CPU that is capable of running the simulation in a reasonable amount of time (or is it?). Re-calculating and correcting the system's center of mass, calculating the accelerations between all 30,000 particles, determining the rms velocity profiles, and updating the velocities and positions each time step are all examples of loops that were run in parallel using *milkyway.* Loops that depend on the order of execution or write data to a file for subsequent analysis could not be paralellized.

Modeling monolithic collapse accurately with 30,000 particles is extremely computationally intensive. At the time of this writing the simulation has been running for nearly 15 hours in parallel using all 64 cores of *milkyway* without completing a single dynamical timescale, and thus we leave the results of the full simulation for a follow up paper. Instead, for this work we ran a scaled-down model of 300 particles, which has the potential to yield at least some results by the due date for submission. In case hogging *milkyway* wasn't obnoxious enough, we simultaneously used all 64 cores of *virgo* to run the scaled-down simulation.

3. RESULTS

The initial configuration of the 300 particles is shown in Figure 1. The initial conditions seem visually satisfactory; the distribution of particles in spherical and homogeneous with $R = 4$ kpc. The density profile of the initial configuration is shown in Figure 2, where $\rho(r)$ is split into 50 density bins. While this is not fantastic resolution, it does show the general shape of the profile well. As expected from a uniform spherical distribution of equal mass particles, the density is roughly constant at all radii. Of course, there is some statistical spread owing to the fact that the particles are spaced randomly. Lastly, the initial v_{rms} profile of the distribution is shown in Figure 3. Since every particle in the simulation is represented by its own velocity dispersion on this plot, the profile's realization ended up being initially noisy enough that it was unreadable. Because of this, we applied IGOR Pro's binomial smoothing algorithm to the profile, somewhat arbitrarily adopting a smoothing parameter of 100. Figure 3 is the resulting smoothed distribution, which resembles the expected initial profile. By construction, the profile is a Gaussian random velocity distribution centered on $v_{rms} = 10$km s^{-1} with $\sigma = 10\%$.

After more than six dynamical timescales the system met the criteria for relaxation, which is given by Eq. 2. The relaxed configuration is shown in Figure 4. A simulation with only 300 particles is a very crude representation of an elliptical galaxy, and this is reflected in the configuration of particles. However, the galaxy does roughly coincide with what we'd expect in a qualitative sense. The change in scale from Figure 1, on the other hand, is particularly disturbing. We expect the system to expand by roughly a factor of five or so, but somehow the expansion is several orders of magnitude. Things get even worse in this respect by the final configuration, which is shown in Figure 5.

It's obvious that the system has developed a denser core and a more diffuse outer region, with some particles being ejected from the system. This can be seen more quantitatively in Figure 6, which shows the density profile of the galaxy at the time of relaxation and every five dynamical times afterward. The profile does not seem to change significantly as the system evolves, although the distribution does continue to extent. Furthermore, to reiterate the point made previously, the x-axis scale is $\sim 10^3$ times larger than in Figure 2, which suggests a serious problem with the model. In addition, the den-

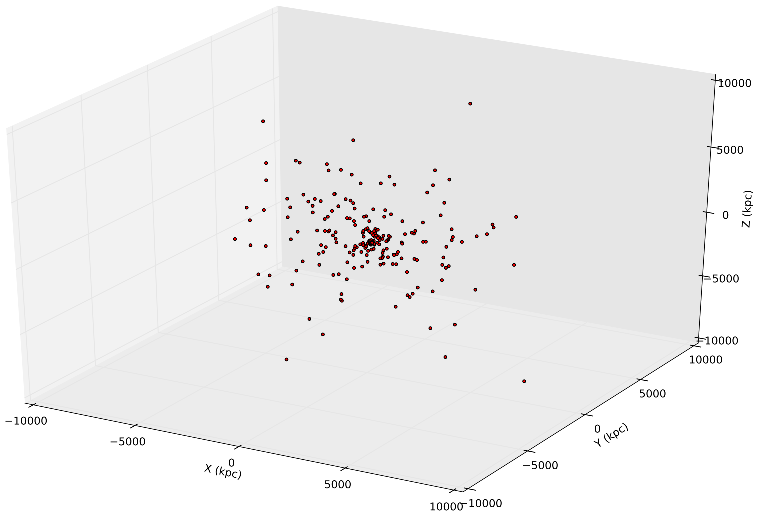

Figure 4. The configuration of the system at $t = t_{rel}$. The particles have undergone more than six dynamical times at the moment of this snapshot, and are beginning to resemble something like an elliptical galaxy.

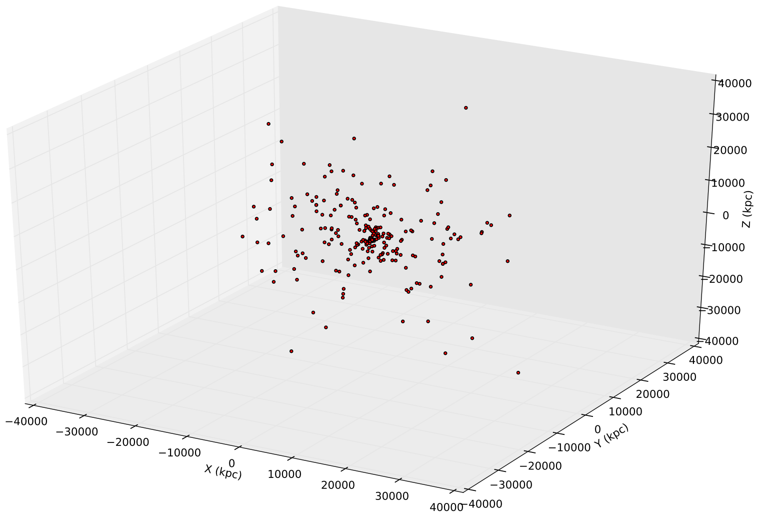

Figure 5. The final configuration of the system at $t = 25t_{dyn}$. The appearance of the particle distribution has not changed significantly since the system relaxed; it still loosely resembles an elliptical galaxy.

sity profiles post-relaxation do not resemble other simulations and elliptical galaxy observations; in fact, they appear to have a sort of reverse broken power law, with density steeply declining near the center and flattening in the outskirts. This obviously does not match what we'd expect ($\rho(r) \propto r^{-1}$ in the inner regions and $\rho(r) \propto r^{-4}$ in the outer regions).

Thankfully the corresponding *rms* velocity profiles are better behaved. Figure 7 shows the smoothed v_{rms} profiles of the galaxy at the time of relaxation and every five dynamical times afterward. Ignoring the same problem from before about the scale of the simulation, as time marches forward the velocity dispersions in the central region of the galaxy actually approach the expected value of $v_{rms} \sim 200$ km s^{-1}! With each successive dynamical time the distribution flattens and settles. The qualitative

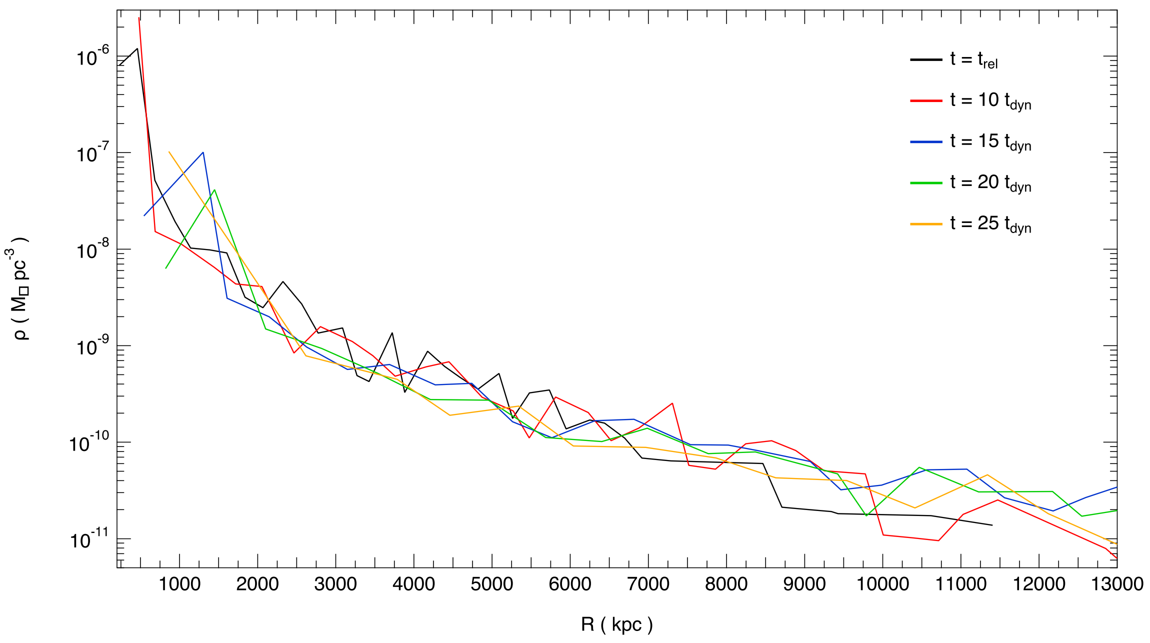

Figure 6. The density profiles of the simulated galaxy at the time of relaxation and every five dynamical times afterward. The profiles are split into 50 radial bins.

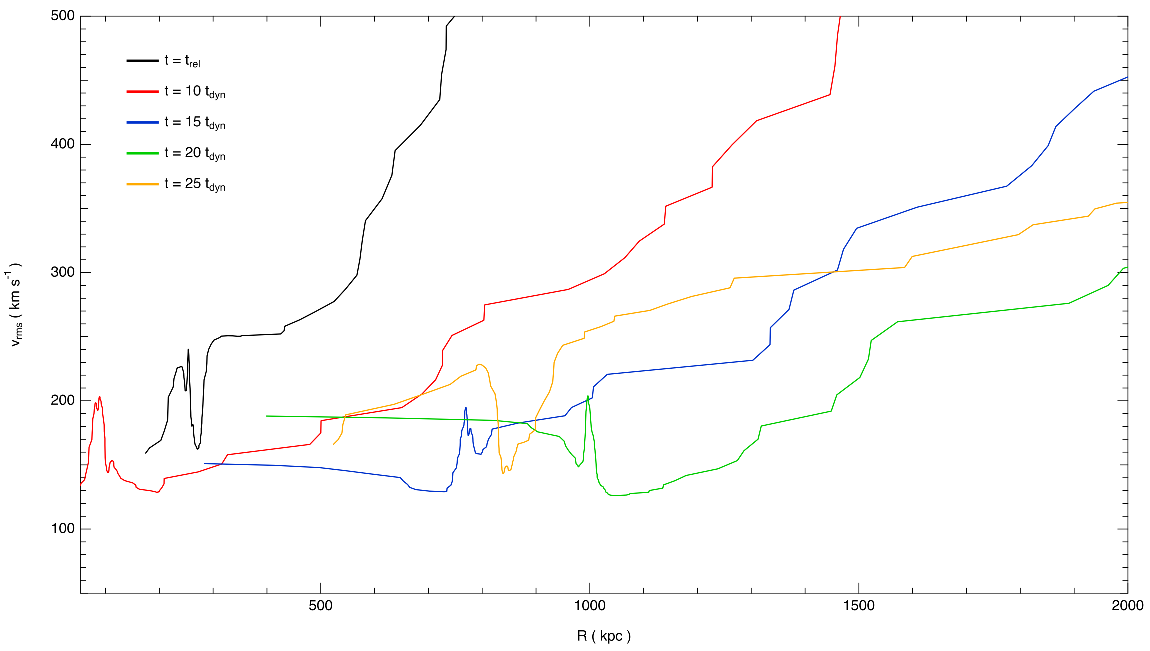

Figure 7. The v_{rms} profiles of the simulated galaxy at the time of relaxation and every five dynamical times afterward. The profiles are a result of a binomial smoothing algorithm with smoothing parameter 100.

behavior with galactocentric radius is also realistic, since velocity dispersion tends to increase at larger distances.

4. CONCLUSIONS

Collisionless simulations are more successful modeling dynamics when the integration time is much less than the relaxation time. In our simulation the system is allowed to relax, which detracts from the authenticity of the model. In reality particles collide, binary systems form, stars have a large dynamic range of masses, and these systems have numerous other intricacies ignored by our model. We also lack sufficient resolution to conduct this simulation. For example, elliptical galaxies are much better approximated by a trillion $1M_{\odot}$ particles than by 30,000 much more massive particles, let alone the 300 used in this simulation. Similarly, this simulation does a very poor job of approximating the smooth density profiles that can be measured for real elliptical galaxies. This is also likely a result of the small number of particles.

Another issue with this model is the enormous change in scale as the system evolves and relaxes. Other simulations show enlarged clouds of particles due to violent relaxation of the system. This simulation also shows this effect, but it is orders of magnitude more dramatic. From our thoughts and analysis, this is likely an emphasis on the importance of initial conditions of the model, particularly in a monolithic collapse simulation. Just as an example, the failure of our model could be a result of a poor choice of initial velocity dispersion with respect to the system's initial radius. Choosing a value of v_{rms} that is too small could cause violent relaxation to spin out of control, greatly expanding the initial cloud of particles.

Lastly, the choice of time step could also be a factor in the less-than-impressive results we see from this simulation. Improving temporal resolution would do wonders for energy conservation and the accuracy of the collapse dynamics; however, the computational cost makes decreasing the time step any further an impossibility for the purposes of this project. I would love to decrease the time step by an order of magnitude, increase the number of particles by a few orders of magnitude, and run the simulations on the best supercomputer on earth. Oh well, I guess I was stuck with *milkyway* and *virgo*.

Important Notes

My code containing 30,000 particles is currently using all 64 cores on *milkyway*, and likely will be for weeks unless JH forces me to control-C. Being gone for the APOGEE meeting for a week did me in. There was no way I'd have time to run the full simulation. Also, many of the comments in the attached code are inaccurate. You can just ignore them; I didn't have time to edit anything.

APPENDIX

```
/*
 *  Monolithic Collapse: N-body code
 *
 *  Copyright  2017 Trevor Picard. All rights reserved.
 *
 * general TO DO: parallelize with OpenMP, check out HDF5, learn how to run in
 *                parallel without crashing the server
 *
 */

//////////////////////header/////////////////////////////////
#include <iostream>
#include <cmath>
#include <math.h>
#include <random>
#include <stdlib.h>
#include <fstream>
#include <algorithm>
#include <omp.h>
using namespace std;



//////////////////////global variables/////////////////////////////////
const int n_dim = 3;                 // spatial dimensions (x, y, z)
const double G = 6.67408e-11;        // gravitational constant
const double pc_m = 3.085677581e+16; // pc to m conversion
const double sol_kg = 1.989e+30;     // solar masses to kg conversion

// physical parameters of the simulation
const int N = 30000;                       // number of particles
const double M = sol_kg * 1e+12;           // total mass of the system [kg]
const double Ri0 = 4000 * pc_m;            // initial radius of the system [m]
double vol = (4 / 3) * M_PI * pow(Ri0, 3); // initial volume of the system [m^3]
const double vi0 = 10000 / sqrt(n_dim);    // initial velocity [m/s]
const double epsilon = 0.001;              // force softening parameter

// times and time-dependent parameters
double t = 0;                                     // initial time
const double t0 = pow(G * M / pow(Ri0, 3), -0.5);  // time scale
const int n_dyn = 25;                             // number of crossing times
const int steps_dyn = 30000;                      // steps per crossing time
const double t_dyn = Ri0 / (vi0 * t0);            // dynamical timescale
const double t_relax = (N * t_dyn) / (8 * log(N)); // relaxation timescale
const double dt = t_dyn / steps_dyn;              // time step
const double t_max = n_dyn * t_dyn;               // end time


/////////////////////////////////////////////////////////////////////
/*
 * Requires: a mean velocity and a standard deviation.
 * Modifies: nothing.
 * Effects : returns a Gaussian random velocity given the mean velocity and
 *           standard deviation of a velocity distribution.
 * Used In : generate_sphere()
 */
double randn(double mu, double sigma) {

    double U1, U2, W, mult;
    static double X1, X2;
    static int call = 0;

    if (call == 1) {

        call = !call;
        return (mu + sigma * (double) X2);
    }

    do {
        U1 = -1 + ((double) rand () / RAND_MAX) * 2;
        U2 = -1 + ((double) rand () / RAND_MAX) * 2;
        W = pow (U1, 2) + pow (U2, 2);
    }

    while (W >= 1 || W == 0);

    mult = sqrt ((-2 * log (W)) / W);
    X1 = U1 * mult;
    X2 = U2 * mult;

    call = !call;

    return (mu + sigma * (double) X1);
}



//////////////////////////////////////////////////////////////////////////
/*
 * Requires: nothing.
 * Modifies: nothing.
 * Effects : defines a particle, which is a data structure with a mass
 *           and with 3D position, velocity, and acceleration components.
 */
struct particle {
    double m;
    double r[n_dim];
    double v[n_dim];
    double g[n_dim];
} p[N];



//////////////////////////////////////////////////////////////////////////
/*
 * Requires: an array of particles.
 * Modifies: the members of each particle in array p.
 * Effects : assigns masses and 3D initial coniditions to the particles in the
 *           simulation.
 * Used In : main()
 */
void initial_conditions(double m,
                        double rx,
                        double ry,
                        double rz,
                        double vx,
                        double vy,
                        double vz,
                        int index) {

    p[index].m    = m;
    p[index].r[0] = rx;
    p[index].r[1] = ry;
    p[index].r[2] = rz;
    p[index].v[0] = vx;
    p[index].v[1] = vy;
    p[index].v[2] = vz;
}



///////////////////////////////////////////////////////////////////////////
/*
 * Requires: a particle array.
 * Modifies: particle accelerations.
 * Effects : resets all acceleration components to 0.
 * Used In : acceleration()
 */
void reset_g() {

    // loops through particles
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {

        // sets all components to 0
        for (int j = 0; j < n_dim; ++j) {
            p[i].g[j] = 0;
        }
    }
}



///////////////////////////////////////////////////////////////////////////
/*
 * Requires: specified initial positions of the N particles in array p.
 * Modifies: particle accelerations.
 * Effects : updates the total accelerations of each particle due to every
 *           other particle in the simulation.
 * Used In : leap_frog()
 */
void acceleration() {

    // sets the acceleration components to 0
    reset_g();

    //cout << "finding the center of mass..." << endl << endl;

    // loops through each particle and finds the center of mass of the system
    double cm[n_dim] = {0, 0, 0};
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < n_dim; ++j) {
            cm[j] += p[i].r[j] / N;
        }
    }


    //cout << "updating positions..." << endl << endl;

    // update the positions relative to the center of mass
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < n_dim; ++j) {
            p[i].r[j] -= cm[j];
        }
    }
    //cout << "calculating accelerations..." << endl << endl;

    // for each particle i...
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {


        // sum accelerations from every other particle j...
        //#pragma omp parallel for num_threads(16)
        for (int j = 0; j < N; ++j) {

            // but not due to the particle itself.
            if (i != j) {

                // distance between particles i & j
                double d  = pow(sqrt(pow(p[j].r[0] - p[i].r[0], 2) +
                                     pow(p[j].r[1] - p[i].r[1], 2) +
                                     pow(p[j].r[2] - p[i].r[2], 2) +
                                     pow(epsilon, 2)), 3);

                // acceleration components of particle i due to particle j
                double gx = p[j].m * (p[j].r[0] - p[i].r[0]) / d;
                double gy = p[j].m * (p[j].r[1] - p[i].r[1]) / d;
                double gz = p[j].m * (p[j].r[2] - p[i].r[2]) / d;

                // sets NaN accelerations to 0
                if (std::isnan(gx)) {
                    gx = 0;
                }
                if (std::isnan(gy)) {
                    gy = 0;
                }
                if (std::isnan(gz)) {
                    gz = 0;
                }

                // adds accelerations to the total for each component
                p[i].g[0] += gx;
                p[i].g[1] += gy;
                p[i].g[2] += gz;
            }
        }
    }
}




////////////////////////////////////////////////////////////////////////////
/*
 * Requires: N particles in array p that have specified masses, positions, &
 *           velocities, and which type of energy to return.
 * Modifies: the density & velocity output file streams
 * Effects : calculates and returns the kinetic, potiential, ratio of kinetic
 *           to potential, and/or total energy of the system.
 * Used In : leap_frog()
 */
void density_v_rms(ofstream &dfile, ofstream &vfile) {

    // defines a volume density array
    struct pair {
        double r;
        double v;
    };

    pair r_vrms[N] = {};
    double rad[N]  = {};

    cout << "determining density & rms velocity profiles...";

    // calculates the corresponding spherical volume at each particle's radius
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        r_vrms[i].r = sqrt(pow(p[i].r[0], 2) +
                           pow(p[i].r[1], 2) +
                           pow(p[i].r[2], 2));

rad[i] = r_vrms[i].r;

        r_vrms[i].v = sqrt(pow(p[i].v[0], 2) +
                           pow(p[i].v[1], 2) +
                           pow(p[i].v[2], 2));
    }


    // sorts the array by radius
    sort(rad, rad + N);

    for (int i = 0; i < N; ++i) {
        vfile << r_vrms[i].v * Ri0 / t0 / 1000 << " "
              << r_vrms[i].r * Ri0 / pc_m / 1000 << endl;
    }
    vfile << endl;


    // calculates the density of particles starting at the center
    const int n_bins = 100;
    int bin_count = 0;
    double  rho[n_bins] = {};
    double r_step = rad[N - 1] / n_bins;
    double r_in = 0;
    double r_out = r_step;
    double V_in = 0;
    double V_out = 0;
    int i = 0;
    while (bin_count < n_bins) {

        // calculates the total volume enclosed within the shell's outer radius
        V_out = (4 / 3) * M_PI * pow(r_out, 3);

        // sums the masses within the shell volume
        double m_tot = 0;
        while ((rad[i] <= r_out) && (i < N)) {
            m_tot += p[i].m;
            ++i;
        }

        rho[bin_count] = m_tot / (V_out - V_in);

        // moves to the next volume shell and writes the density
        r_in = r_out;
        r_out += r_step;
        V_in = V_out;
if (m_tot != 0) {
    dfile << (rho[bin_count] * M * pow(pc_m / Ri0, 3)) / sol_kg << " "
                 << rad[i] * Ri0 / (pc_m * 1000) << endl;
}
```

```
        ++bin_count;
    }
    dfile << endl;
}




////////////////////////////////////////////////////////////////////////////
/*
 * Requires: an output file stream & N particles in array p.
 * Modifies: the output file stream and the initial conditions of the system.
 * Effects : assigns random positions to N particles within a sphere of radius
 *           1, calculates the mean velocity required for an energy ratio of
 *           K/U = -0.1, then generates random velocities using randn().
 * Used In : main()
 */
void generate_sphere() {


    default_random_engine generator;
    uniform_real_distribution<double> distribution(-1, 1);


    for (int i = 0; i < N; ++i) {

        bool done = false;

        while (!done) {

            double x0 = distribution(generator);
            double y0 = distribution(generator);
            double z0 = distribution(generator);

            bool in_sphere = (sqrt(pow(x0, 2) + pow(y0, 2) + pow(z0, 2)) <= 1);

            if (in_sphere) {
                initial_conditions(1. / N,                          // m
                                   x0,                              // x_0
                                   y0,                              // y_0
                                   z0,                              // z_0
                                   0,                               // v_x,0
                                   0,                               // v_y,0
                                   0,                               // v_z,0
                                   i);
                done = true;
            }
        }

        // assigns Gaussian random velocities
        double sig = 0.1 * vi0 * t0 / Ri0;
        for (int j = 0; j < n_dim; ++j) {
            p[i].v[j] = randn(vi0 * t0 / Ri0, sig);
        }
    }

    cout << "initial configuration set." << endl << endl;
}



////////////////////////////////////////////////////////////////////////////
/*
```

```
 * Requires: an open output file stream & specified initial conditions
 * Modifies: the output file stream
 * Effects : Evolves the system of particles in time using leap-frog
 *           integration, keeping track of particle properties and the total
 *           energy of the system.
 * Used In : check_and_run()
 */
void leap_frog(ofstream &dfile, ofstream &vfile, ofstream &pfile) {

    // counts the steps, # of crossing times, and half-mass radii
    int step_count = 0;
    int dyn_count  = 0;
    int rho_v_step = 5;

    bool pending_relaxed = true;
    bool print_one = true;
    bool relax_print = false;

    // loops through each time step
    while (t <= t_max) {

      //cout << step_count << endl;

        if (pending_relaxed && (t >= t_relax)) {
            cout << endl << "aaaand the system is relaxed." << endl << endl;
            pending_relaxed = false;
        print_one = true;
        relax_print = true;
        }

        // finds accelerations of all particles
        acceleration();


        // keeps track of when to record position data
        bool snapshot = (print_one && ((t == 0) || ((!pending_relaxed &&
      (dyn_count % rho_v_step == 0)))));

        // loops through each particle
        for (int i = 0; i < N; ++i) {

            // writes requested snapshot data to the file
            if (snapshot || relax_print) {

                // loops through and writes x, y, z dimensions to the file
                for (int j = 0; j < n_dim; ++j) {
                    pfile << (p[i].r[j] * Ri0) / (pc_m * 1000) << " ";

                    // starts a new line once all components are written
                    if (j == n_dim - 1) {
                        pfile << endl;
                    }
                }
            }

            // updates positions/velocities for the next time step
    #pragma omp parallel for
            for (int j = 0; j < n_dim; ++j) {

                // updates velocity components only after t_0
                if (t > 0) {
                    p[i].v[j] += p[i].g[j] * dt;
                }
                p[i].r[j] += p[i].v[j] * dt;
            }
        }

        // skips a line after each recorded snapshot
        if (snapshot || relax_print) {

            // writes the density & v_rms profiles to files
            density_v_rms(dfile, vfile);
            cout << "done." << endl << endl;
            pfile << endl;

            print_one = false;
        relax_print = false;
        }

        // moves to the next time step
        t += dt;
        ++step_count;

        // counts # of crossing times
        if (step_count % steps_dyn == 0) {
            ++dyn_count;
            print_one = true;
            cout << endl << dyn_count << " dynamical times complete."
                 << endl << endl;
        }
    }
}



//////////////////////////////////////////////////////////////////////////
/*
 * Requires: an output file stream & an integration method
 * Modifies: the output file stream
 * Effects : Checks if a file was created and opened properly, then runs the
 *           simulation using the specified integration method.
 * Used In : main()
 */
void check_and_run(ofstream &dfile, ofstream &vfile, ofstream &pfile) {


    // checks if the file was opened correctly
    if (dfile.is_open() && vfile.is_open() && pfile.is_open()) {

        leap_frog(dfile, vfile, pfile);

        // closes the file and resets the time
        dfile.close();
        vfile.close();
        pfile.close();
        t = 0;
    }
    else {
        cout << "something's wrong boi..." << endl;
    }
}



//////////////////////////////////////////////////////////////////////////
/*
 *
 *
 * BEGIN MAIN PROGRAM
 *
 *
 */
int main() {


/*
 * random spherical distribution
 *
 */
    int N_threads = 64;
    omp_set_num_threads(N_threads);

    generate_sphere();

    // runs the euler method and stores position data
    ofstream density;
    ofstream v_rms;
    ofstream particles;
    density.open("density.txt");
    v_rms.open("v_rms.txt");
    particles.open("particles.txt");

    check_and_run(density, v_rms, particles);

}
```