# Project 5, FYS 3150 / 4150, fall 2013

Student number 555070 (in collaboration with student number 531008 )

December 9, 2013

# Contents

# 1 Introduction

Open clusters are groups of up to a few thousands stars, held together by mutual gravitational attraction. Open clusters generally last for a few hundred milions years. In it also important to underline that the parameters of the cluster's stars are kind of constant, since they are made from the same material. We want to build a model for such a cluster, and study its "cold collapse" (where "cold" comes from the fact that the particles have no initial velocity). To do so, we will begin with the approximation that stars are particles, point masses. Then, we will use a simple smoothing algorithm to increase the numerical stability of our system, for which we are able to derive a potential energy. We have also outlined a method to find an optimal smoothing parameter if one begins from a well-tested value. This value can then be scaled to fit other simulations with a suprisingly simple formula. However, it has some important limitations that we will discuss.

Numerous algorithms could have been used to simulate the behavior of the cluster. We chose to focus on the Leapfrog and on the fourth-order Runge-Kutta methods. Each one of the numeric methods has its pros and cons; we seek to find the one which gives priority to the stability instead of the short-term accuracy, as well as a reasonable running time. Due to the number of particles in our cluster, we will be interested in the statistical properties of our system instead of in the specific characteristics of each particle.

Our goal in this report is to simulate and to study the behavior of a cluster, after having paid attention to the algorithm used to derive the results. The first part of the report will focus on the analytical properties of the model (including algorithms). We then proceed to give an overview of the code structure, and finally present and discuss our results.

# 2 Theory

## 2.1 Derivation of the expression for $\tau_{crunch}$

We start with the parametric form of the Friedmann equations for a closed universe containing only dust in addition to curvature[1]:

$$R(\psi) = a(\psi)R_0 = \frac{R_0\Omega_{m0}}{2(\Omega_{m0}-1)}(1-\cos\psi) \tag{1}$$

where $a(\psi)$ is the dimensionless scale factor and

$$t(\psi) = \frac{\Omega_{m0}}{2H_0(\Omega_{m0}-1)^{3/2}}(\psi-\sin\psi). \tag{2}$$

We will use $t = 0$ at the Big Bang singularity for our sub-universe. From these results we see that $t_{max} = t(\psi = \pi)$ and $t_{crunch} = t(\psi = 2\pi)$, thus the elapsed time between these events is

$$\tau_{crunch} = t_{crunch} - t_{max} = \frac{\pi\Omega_{m0}}{2H_0(\Omega_{m0} - 1)^{3/2}}.$$

The mass parameter is defined by

$$\Omega_{m0} = \frac{8\pi G\rho_0}{3H_0^2}$$

and for readability we will make the substitution

$$u^2 = 8\pi G\rho_0 - 3H_0^2$$

thus

$$(\Omega_{m0} - 1)^{-3/2} = (\frac{u^2}{3H_0^2})^{-3/2} = 3\sqrt{3}H_0^3u^{-3}$$

$$\frac{\Omega_{m0}}{(\Omega_{m0} - 1)^{3/2}} = \frac{8\pi G\rho_0}{3H_0^2}3\sqrt{3}H_0^3u^{-3} = 8\sqrt{3}\pi G\rho_0 H_0 u^{-3}$$

$$\tau_{crunch} = \frac{\pi}{2H_0}8\sqrt{3}\pi G\rho_0 H_0 u^{-3} = 4\sqrt{3}\pi^2 G\rho_0 u^{-3}$$

Now we remember that at the time when $\rho = \rho_0$, everything is at rest, so we have $H_0 = (\frac{\dot{a}}{a})_{\tau=0} = 0$. Inserting this, we get $u^2 = 8\pi G\rho_0$, and

$$\tau_{crunch} = 4\sqrt{3}\pi^2 G\rho_0(8\pi G\rho_0)^{-3/2} = \sqrt{\frac{4^2 3\pi^4 G^2\rho_0^2}{8^3\pi^3 G^3\rho_0^3}} = \sqrt{\frac{3\pi}{32G\rho_0}} \qquad (3)$$

which is what we wanted to show.

## 2.2 Lack of a singularity in our model

The reason we do not see a singularity in our model is that we have assumed pressureless (i.e. collisionless) matter that is homogenously distributed in the Friedmann equations. In our simulation we will see internal "pressure" as kinetic energy from the collapse is turned into random motions by near-collisions

between the point mass particles. These motions will halt the collapse, causing the gravitationally bound particles to form more or less stable orbits around the center of mass.[2, 3] Keep in mind that the kinetic energy is not evenly distributed, so occasionally particles that receive more than their fair share will become unbound and may escape from the system before their energy surplus is removed. It is predicted that this should happen to a certain percentage of the particles, depending on $N$, with a lower bound of 50%.[4]

We say that the system is stable when it satisfies the virial theorem:

$$2\langle K\rangle = -\langle P\rangle$$

where $\langle P\rangle$ and $\langle K\rangle$ are the averages of the potential and kinetic energies (actually, these are supposed to be time averages for the total energies of the entire system, but it turns out that the averages of a point in time of the energies per particle is a good approximation to this[5]). It can be shown that this happens at time $\tau_{vir} = 0.81\tau_{crunch}$ when the sphere has collapsed to half its initial size, so we see that $\tau_{crunch}$ is a natural time scale for virialization to occur (some sources do in fact use $\tau_{crunch}$ to mark the point when the system is virialized).[2]

## 2.3 $G$ in units of ly, $M_{\odot}$ and $\tau_{crunch}$

With $\tau_{crunch}$ given in years, we can rewrite equation 3 as

$$G_{yr} = \frac{3\pi}{32\tau_{crunch}^2\rho_0}.$$

Switching time units to $\tau_{crunch}$, we get that $\tau_{crunch} = 1$ in these units, hence

$$G = \frac{3\pi}{32\rho_0} = \frac{\pi^2 R_0^3}{8\mu N} \tag{4}$$

where for the latter equality we have used the definitions of average mass $\mu = \frac{M}{N}$ and initial mass density for a sphere $\rho_0 = \frac{M}{V_0} = \frac{\mu N}{V_n(R_0)}$, where $V_n(R_o)$ is the volume of the n-dimensional ball with radius $R_0$ (see equation 7).

This means our gravitational constant and our time unit both depend on $N$, $R_o$ and $\mu$. With $R_0$ given in light years and $\mu$ given in solar masses, we get the desired units for $G$.

## 2.4 Gravitational potential with modified gravity

In many of our simulation we will use a smoothing term in our gravitational potential to compensate for the fact that we use time steps on a certain length.

With point masses (that do not have a collisional cross section), the forces will approach infinity as $r \to 0$ with the classical Newtonian approach. We therefore modify the forces using a length scale $\epsilon$, at which point the forces will converge toward the finite magnitude $\frac{GMm}{\epsilon^2}$ instead of infinity. This modification will have minimal effect when $r \gg \epsilon$.

However, this also slightly changes the way we calculate the potential energy. Starting with the magnitude of the force

$$F = \frac{GMm}{r^2 + \epsilon^2} = \frac{GMm}{\epsilon^2} \frac{1}{(\frac{r}{\epsilon})^2 + 1} = \frac{GMm}{\epsilon^2} \frac{1}{u^2 + 1}$$

where we have used the subsitution $u = \frac{r}{\epsilon}$ which gives $\frac{du}{dr} = \frac{1}{\epsilon}$, hence $dr = \epsilon \cdot du$, and so

$$E_p = m\Phi = \int F dr = \frac{GMm}{\epsilon^2} \int \frac{1}{u^2 + 1} \epsilon du = \frac{GMm}{\epsilon} (\arctan(u) + C)$$

We want $E_p \to 0$ as $r \to \infty$, and since $\arctan(u) \to \frac{\pi}{2}$ as $u \to \infty$, we achieve this by choosing $C = -\frac{\pi}{2}$:

$$E_p = \frac{GMm}{\epsilon} (\arctan(\frac{r}{\epsilon}) - \frac{\pi}{2}) \qquad (\epsilon > 0) \tag{5}$$

## 2.5 Scaling the gravitational smoothing parameter $\epsilon$

The challenge with the $\epsilon$ values is that we want it to be as small as possible (to give more realistic results), yet make the number of ejected particles as small as possible, conserving as much of the total energy as we can. We chose to determine a good fit for $\epsilon$ experimentally (see figures 16 and 17), and then we determined how this value would scale to different simulations in the following way:

The basic premise is that when $r = \epsilon$ (i.e. when the $\epsilon$ term starts dominating the gravitational potential), we want to set an upper limit on how much a particle's velocity can change during one time step due to the gravitational attraction from one other particle. We want this upper limit to be invariant across simulations:

$$\Delta v_1 = \Delta v_2$$

$$a_1 \Delta t_1 = a_2 \Delta t_2$$

We will assume all particles have the average mass $\mu$ and set $r = \epsilon$:

$$\frac{G_1\mu_1\Delta t_1}{2\epsilon_1^2} = \frac{G_2\mu_2\Delta t_2}{2\epsilon_2^2}$$

Now, inserting the value of $G$ from 4 with initial radii $R_1$ and $R_2$, we get:

$$\frac{\pi^2 R_1^3\mu_1\Delta t_1}{2\epsilon_1^2 \cdot 8\mu_1 N_1} = \frac{\pi^2 R_2^3\mu_2\Delta t_2}{2\epsilon_2^2 \cdot 8\mu_2 N_2}$$

$$\frac{R_1^3\Delta t_1}{\epsilon_1^2 N_1} = \frac{R_2^3\Delta t_2}{\epsilon_2^2 N_2}$$

giving us this handy formula for $\epsilon_2$ if we have a good match for $\epsilon_1$:

$$\epsilon_2 = \sqrt{\frac{N_1}{N_2}\left(\frac{R_2}{R_1}\right)^3 \frac{\Delta t_2}{\Delta t_1}} \cdot \epsilon_1 \tag{6}$$

## 2.6 Volume of the n-ball

To calculate the gravitational constant with $\tau_{crunch}$ as the time unit in any dimension, we need the volume of the sphere in $n$ dimensions to calculate the initial mean density $\rho_0$. This is accomplished by the following formula[6]:

$$V_n = \frac{\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2}+1)} \tag{7}$$

## 2.7 Uniform distribution in the n-ball

Our dimension-independent algorithm for generating uniformly distributed points inside the n-ball is[7]:

1. Generate random points on the surface of the unit n-ball (i.e. randomize the directions of the unit vectors):
   (a) Generate normally distributed n-dimensional vector $\mathbf{x}_n = [x_1, x_2, \ldots, x_n]$ where the normal distribution has $\mu = 0$ and $\sigma = 1$.
   (b) Calculate the n-dimensional norm of the vector $|\mathbf{x}_n|$. We chose to let Armadillo handle this, but another easy way to do it is using the n-dimensional dot product and taking the square root of this: $|\mathbf{x}_n| = \sqrt{\mathbf{x}_n \cdot \mathbf{x}_n} = \sqrt{\sum_i x_i^2}$.

(c) Turn it into a unit vector: $\mathbf{u}_n = \frac{\mathbf{x}_n}{|\mathbf{x}_n|}$. We refer to (source) for the proof that this is uniformly distriuted in terms of direction.

2. Generate a radius that results in a uniform distribution within maximum radius $R_0$:

   (a) Generate a uniformly distributed value $r \in [0, 1]$.

   (b) The desired radius that takes into account that the outer spherical shells have a larger surface area is $R = \sqrt[n]{r} \cdot R_0$, where $n$ is the number of dimensions.

## 2.8 Algorithms

For both of our algorithms, the step interval is $[t_0; t_{max}]$. The number of steps we call $n_{Steps}$, and the resulting step length is $h = \frac{t_{max} - t_0}{n_{Steps} - 1}$.

### 2.8.1 Runge-Kutta

We have $t_i = t_0 + i \cdot h$ for each i belonging to $[0, n_{Steps}]$. We can now define $x_i = x(t_i)$.

The general formula for Simpson's rule is: $\int_{t_i}^{t_i+1} f(t, x)dt = \frac{1}{6}h(f(t_i, x_i) + 2f(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) + f(t_{i+1}, x_{i+1}))$.

Runge-Kutta's method defines four quantities, to predict and correct the value of $x_{i+1}$:

$$k_1 = f(t_i, x_i)$$

$$k_2 = f(t_{i+\frac{1}{2}}, x_{i+\frac{1}{2}}) \qquad x_{i+\frac{1}{2}} = x_i + \frac{h}{2}k_1$$

$$k_3 = f(t_{i+\frac{1}{2}}, x^*_{i+\frac{1}{2}}) \qquad y_{i+\frac{1}{2}} = x_i + \frac{h}{2}k_2$$

$$k4 = f(t_{i+1}, x_{i+1}) \qquad x_{i+1} = x_i + hk_3$$

This Runge-Kutta algorithm is especially interesting since we only need initial conditions to unfold it. Its approximation error runs like $\mathcal{O}(\Delta h^4)$ .[9]

### 2.8.2 Leapfrog algorithm

The Leapfrog algorithm is given by the trhee following steps:

$$x^{(1)}(t+\frac{h}{2}) = (x^{(1)}(t) + \frac{h}{2} \cdot x^{(2)}(t) + \mathcal{O}(\Delta h^2)$$

$$x(t+h) = x(t) + x^{(1)}(t+\frac{h}{2}) + \mathcal{O}(\Delta h^3)$$

$$x^{(1)}(t+h) = x^{(1)}(t+\frac{h}{2}) + \frac{h}{2} \cdot x^{(2)}(t+h) + \mathcal{O}(\Delta h^2)$$

Now, let's take a look at the approximation error for this algorithm. The first step to derive this algorithm is to use Taylor expansion on the position:

$$x(t+h) = x(t)+hx^{(1)}(t)+\frac{h^2}{2}x^{(2)}(t)+\mathcal{O}(\Delta h^3) = x(t)+h(x^{(1)}(t)+\frac{h}{2}\cdot x^{(2)}(t))+\mathcal{O}(\Delta h^3)$$

In the same way, we have for the velocity:

$$x^{(1)}(t+\frac{h}{2}) = x^{(1)}(t) + \frac{h}{2} \cdot x^{(2)}(t) + \mathcal{O}(\Delta h^2)$$

$$x^{(1)}(t-\frac{h}{2}) = x^{(1)}(t) - \frac{h}{2} \cdot x^{(2)}(t) + \mathcal{O}(\Delta h^2)$$

$$x^{(1)}(t+\frac{h}{2})-x^{(1)}(t-\frac{h}{2}) = x^{(1)}(t)+\frac{h}{2}\cdot x^{(2)}(t)+\mathcal{O}(\Delta h^2)-(x^{(1)}(t)-\frac{h}{2}\cdot x^{(2)}(t)+\mathcal{O}(\Delta h^2)) = h\cdot x^{(2)}(t)+\mathcal{O}(\Delta h^2)$$

So

$$x^{(1)}(t+h) = x^{(1)}(t+\frac{h}{2}) + \frac{h}{2} * x^{(2)}(t+h) + \mathcal{O}(\Delta h^2)$$

We can rewrite the first step thanks to the expression of $x^{(1)}(t+\frac{h}{2})$:

$$x(t+h) = x(t) + x^{(1)}(t+\frac{h}{2}) + \mathcal{O}(\Delta h^3)$$

In the end, we can see that the approximation error runs like $\mathcal{O}(\Delta h^2)$ for the Leapfrog algorithm.[9]

# 3 Implementation

## 3.1 Overview

The code is designed to be highly modular and independent of dimension to maximise reusability.


Figure 1: Class Diagram

### 3.2 Main program (Project5)

Sets up a series of simulations so we can plot certain results as a function of either $N$ (number of particles), $\epsilon$ (correction factor for gravitation) or $\Delta t$ (time step). This automated approach made the process of plotting considerably easier for us than setting up everything "by hand" each time. The main program also contains code to test the simulation against a 2D benchmark (Project 3) and contains helper methods for curve fitting.

### 3.3 SolarSystem class

Container class for the entire N-body system. This class is dimension independent (number of dimensions is given as a parameter), to make it as general and reusable as possible. Does not specify a method to solve the equations of motion, so any numerical method can be used on this class to update the positions every time step (by iterating over the celestial bodies contained in this class). A SolarSystem can make deep copies of itself (and all its CelestialBody and Gravity objects), which we make use of to be able to run different algorithms on identical copies of a system (for comparison of results and stability analysis). Some resource intensive properties, like potential energy, are only calculated when we need to plot the data - this can be set not to happen at every step to speed up the execution.

### 3.4 CelestialBody class

Particle class of our N-body simulation, with position and mass. Handles all calculations on the individual particle level (e.g. kinetic energy) and can be set to fixed if desired. Also stores properties calculated by the encompassing SolarSystem (like potential energy) so these will not have to be calculated on the fly every time they are needed. This class inherits dimensionality from the SolarSystem it belongs to, so we avoid creating a sub-dimensional celestial body by accident.

### 3.5 Gravity class

This class allows each SolarSystem to use a different gravity. One can change both the value of the gravitational constant to fit the time unit of choice and set an $\epsilon$ value to dampen collisions. For future use it is possible to extend this class to include any form of modified gravity (which is an object of interest in cosmology[8]). The only thing the SolarSystem class requires of this object is that it is able to return a force and potential energy when two CelestialBody objects are given as input. This way, a SolarSystem does not need to handle the specifics of gravitational forces itself, making the code more modular.

### 3.6 CelestialBodyInitializer class

Sets up a uniform position distribution given an initial maximum radius (generalized for any number of dimensions). Also generates normal distributed random masses. Doing this in a separate class allows the SolarSystem class to be as general as possible (we might not want a random distribution every time, but when we do, this class will provide it). Keeping this class dimension independent makes it more general and easier to use in other projects. Sets the values of $G$ and $\epsilon$ based on the initial state of the SolarSystem, using eqs. (4) and (8), respectively.

### 3.7 Solvers class

Contains code to iterate over a SolarSystem object over a number of time steps, using the Leapfrog, Runge-Kutta (4th order) and Euler-Cromer algorithms to solve the equations of motion. Creates copies of the system given as a parameter to do this, so we can keep the original system unchanged and also solve using several numerical methods simultaneously and compare the results. Keeping this in a separate class allows adaptation of this code to use other objects than the SolarSystems we employ here. Also times the execution of each algorithm used.

### 3.8 GaussPDF class (not included in appendix)

This is simply a static class wrapper we made for the code provided at:

https://www.uio.no/studier/emner/matnat/fys/FYS3150/h13/gaussiandeviate.cpp

It provides random numbers in the uniform and normal probability distributions.

## 4 Results and analysis

### 4.1 Benchmarks and validation

In a previous project, we worked on the implementation of the Runge-Kutta algorithm to simulate the behavior of the Solar System. To test our code, we also wrote the Euler-Cromer algorithm. In order to verify that our code works, we tested it against our previous results, in two dimensions, then we extended it to three dimensions.

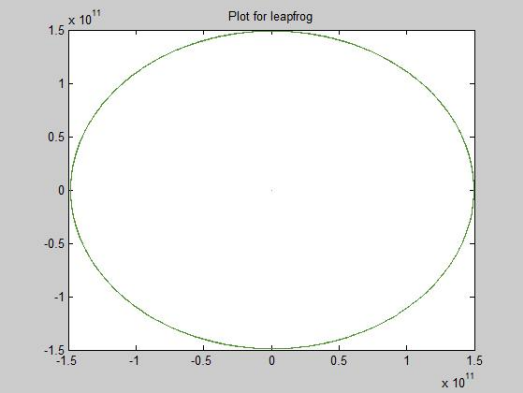


Figure 2: Simulation with our Leapfrog algorithm

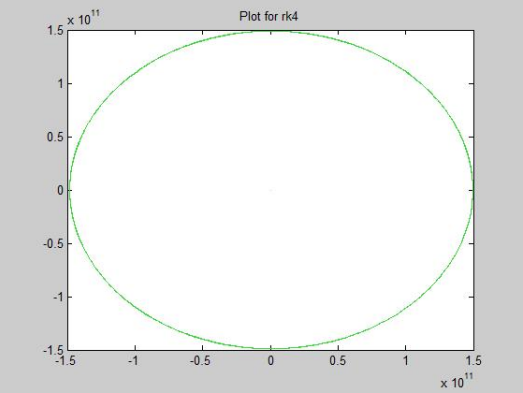


Figure 3: Simulation with our Runge-Kutta - 4 algorithm

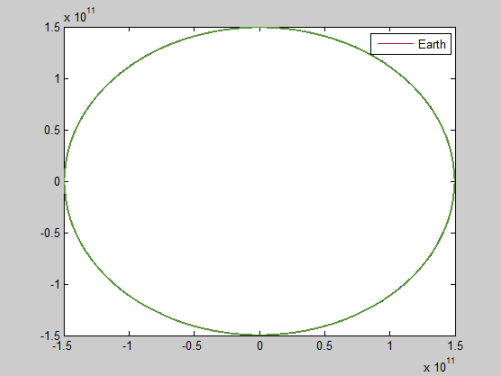


Figure 4: Simulation of our previous project

We found equivalent results between the two projects, for the same time step size (one point per day) and the same length of simulation (two years).

When we use them for very long times, the Leapfrog algorithm should be more well-behaved than the Runge-Kutta one[10].

We also did an analysis on runnning time of the two algorithms:

| Execution time (s) | 1000 steps, 2 bodies | 1000 steps, 100 bodies | 5000 steps, 100 bodies |
|---|---|---|---|
| Leapfrog | 19,18 | 86,35 | 405,73 |
| Runge-Kutta 4 | 30,85 | 171,25 | 1456,12 |

Table 1: Elapsed Time for different parameters

We can see here that the Runge-Kutta algorithm takes more time to process the same set of data than the Leapfrog. For small data set, the time spent by the forth order Runge-Kutta algorithm is twice as much as the time spent by the Leapfrog method. And when we increase the number of steps, or the number of bodies, the relative time spent grows even shorter for the Leapfrog algorithm. Thus, the difference in the execution time between Runge-Kutta and Leapfrog becomes more and more significant.

Looking at the energy conservation, we see significantly better results with Runge-Kutta compared to Leapfrog for the 2-body problem.

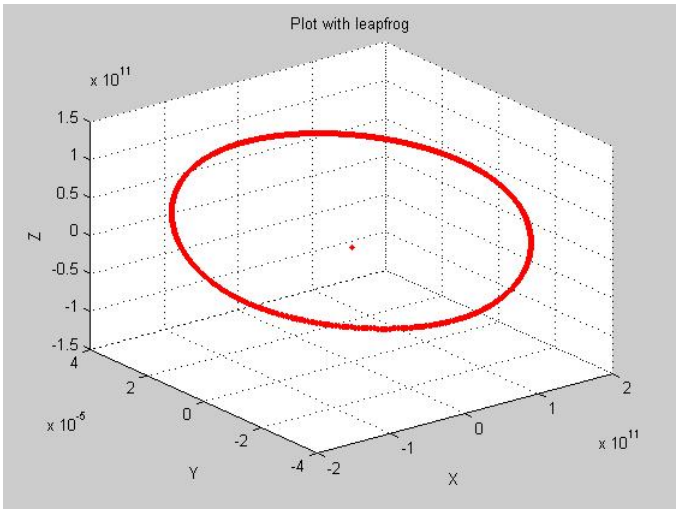


Figure 5: Conservation of energy – Leapfrog – Length: 2 yars

```
E_k before: 1.33206e+033
E_p before: 0
E_tot before: 1.33206e+033
Entering the Benchmark part
Using Leap-frog
Aaand ... We're done
E_tot after: -1.62098e+033
E_k after: 1.37681e+033
E_p after: -2.9978e+033
```

Figure 6: Conservation of energy – Leapfrog

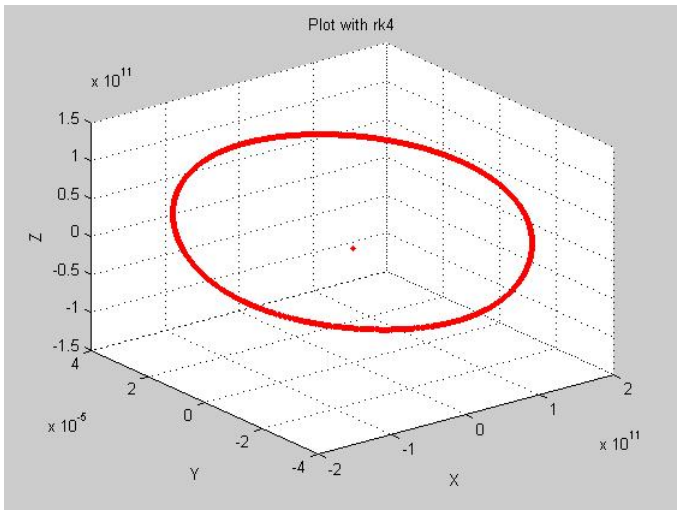


Figure 7: Conservation of eenergy – fourth-order Runge-Kutta – Length: 2 years

```
Aaand ... We're done
E_k before : 2.66412e+033
E_p before : 0
E_tot before : 2.66412e+033
E_tot after: -2.68958e+033
E_k after: 2.66412e+033
E_p after: -5.3537e+033
```

Figure 8: Conservation of energy – fourth-order Runge-Kutta

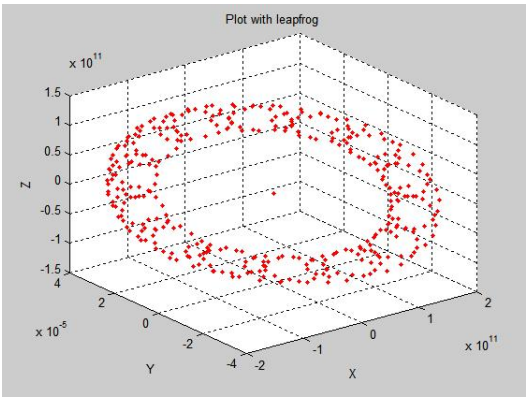


Figure 9: Behavior of the system with bigger time steps after 10 years with Leapfrog. 2-body problem with position plotted for all time steps.

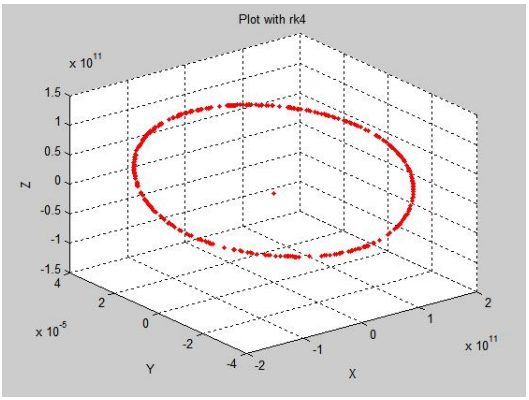


Figure 10: Behavior of the system with bigger time steps after 10 years with Runge-Kutta (4th order). 2-body problem with position plotted for all time steps.

We saw in the theory part that the forth-order Runge-Kutta has an approximation error smaller than the Leapfrog algorithm. It basically means that for small time steps, local data about each particles will be more accurate with the Runge-Kutta method than with the Leapfrog method, even though this one still gives us interesting results for reasonably small time steps. The Runge-Kutta method is also more accurate to compute the velocity, but here, it does not matter so much. As we can see on the above figures, with bigger time steps, we have a more stable system with the Runge-Kutta algorithm than with the Leapfrog algorithm as stated in [9].

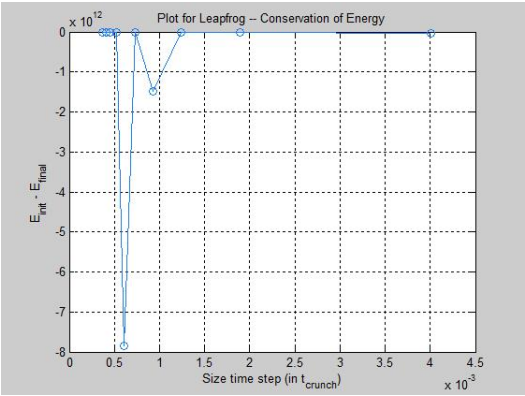


Figure 11: Energy conservation for different time steps after 4 $\tau_{crunch}$ (Leapfrog).

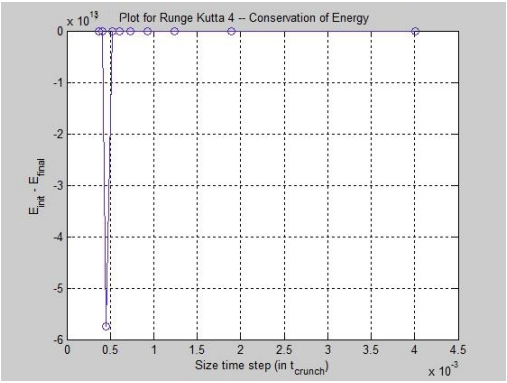


Figure 12: Energy conservation for different time steps after 4 $\tau_{crunch}$ (Runge-Kutta).

For the choice of the time step size, we simulate the system with two random particles in our 20 light year radius sphere. And we looked at the energy conservation. We can see here that for big time steps, the conservation of the energy becomes really messy. But for small enough one, the energy is well conserved.

But we should expect, as stated in [10], that at some point, the results given by the Runge-Kutta method will start to drift, while they are still rather stable with the Leapfrog algorithm. Curiously, we were unable to reproduce this effect in our 2-body simulations (we used energy conservation to look for this effect).

| Time steps | 1000000 | 100000 | 10000 |
|---|---|---|---|
| Leapfrog | 2,66412 + 33 \| - 2,68958 + 33 | 2,66412 + 33 \| - 2,68958 + 33 | 2,66412 + 33 \| - 2,68958 + |
| Runge-Kutta | 2,66412 + 33 \| - 2,66412 + 33 | 2,66412 + 33 \| - 2,68958 + 33 | 2,66412 + 33 \| - 2,68958 + |

Table 2: Initial and final total energy in function of the time step size

Since we want to simulate a system over a long time period, with at least 100 particles, we are more interested in the long term statistical accuracy of our system than in the local accuracy of the position or velocity of each particle. In the end, our choice of algorithm comes down to running time, long-term stability and the addition of the smoothing factor $\epsilon$. The latter gives us more leeway when it comes to time step length (a connection which is seen in equation 6), in which case we are more concerned about our system being stable in the long run. We therefore prefer the Leapfrog algorithm in our simulations.

## 4.2 Application to a multi-bodies system

### 4.2.1 Evaluation of $G$

```
T_CRUNCH = 7.97246e+006
G = 9.92352
G_YLS = 1.56128e-013
```

Figure 13: Computation of G

With a system of $N = 100$ particles, an initial radius of $R_0 = 20$ light years, we found a $\tau_{crunch}$ of nearly 8 millions of years. In the figure above, the variable called G_YLS is the gravitational constant, uses meters, seconds, and kg. G is computed in ly, $M_{\odot}$ and $\tau_{crunch}$.

A natural time unit for $\tau_{crunch}$ is thus Myrs ($10^6$ yrs).

### 4.2.2 Applying the algorithms to a many-body simulation

We now extend our code to include more than two particles. To find an appropriate time step length, we will look at the total energy conservation over different time steps, for our two algorithms. Our system is the following: 100 particles, and an initial radius of 20 light years.

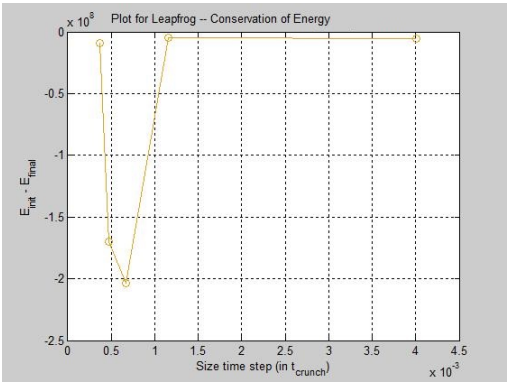


Figure 14: Energy conservation for different time steps - 4 $\tau_{crunch}$– Leapfrog

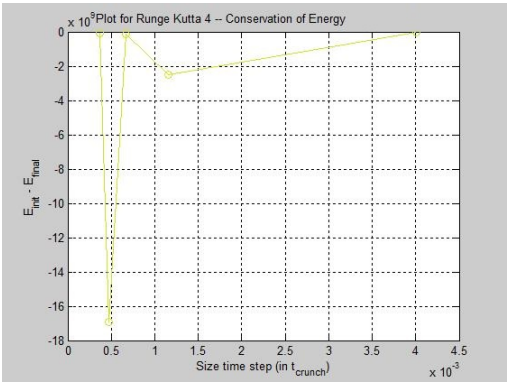


Figure 15: Energy conservation for different time steps - 4 $\tau_{crunch}$ – Runge-Kutta 4

With more particles than before, we can see that the conservation of the energy did not change so much. The difference between the initial energy and the final energy is mainly due to the fact that without using the smoothing function, we

lose up to 45% of our initial particles. So there has to be a difference between the initial energy and the final one. We can see on the above figures that Runge-Kutta is less accurate than Leapfrog, which is more stable. As stated before, we are more in a long-term stable algorithm, rather than in locally accurate one. When using the Leapfrog method, we initially chose a time step length of $2,5.10^{-3}\tau_{crunch}$ to be able to see the effect of the gravitational smoothing parameter.

## 4.3 Energy conservation

### 4.3.1 Bound particles

Wher we refer to bound particles, we refer to those particles that are not gravitationally bound. Since potential energy is negative (at most 0), and kinetic energy is positive (at least 0), we say that a particle is bound when its total energy $E_{tot} = E_k + E_p < 0$.

### 4.3.2 Finding a reference value for $\epsilon$

With a system of $N = 100$ particles, an initial radius of $R_0 = 20$ light years and a time step of $\frac{1}{250}\tau_{crunch}$, we did a series of simulations where we varied the gravitational correction parameter in the interval $\epsilon \in [0,0.15]$. According to equation 6, the values are supposed to be independent of mass, but the masses used were a normal distribution with mean $\mu = 10$ and $\sigma = 1$ (both in solar masses).

The criteria for accepting an $\epsilon$ value are that the number of bound particles are as high as possible, while the total energy, at least for the bound particles, is as conserved as possible. In the process, we discovered that these criteria are actually mutually exclusive (see figure 16): The energy conservation for bound particles improves if we eject many particles. Our interpretation of this result is that particles that are bound but close to being unbound will impact the energy bound conservation negatively: If these particles are actually ejected, the remaining particles' total energy is better conserved. Thus, we need to look at both criteria simultaneously when deciding on a value for $\epsilon$. Additionally, we want $\epsilon$ to be as close to 0 (and the Newtonian limit) as possible, so we get more realistic results.

We also see that the number of ejected particles will fluctuate considerably between simulations with the same starting parameters.[4] Ideally, we would run many simulations with the same epsilon value to eliminate statistical errors - this is something that could be looked at in a future study.

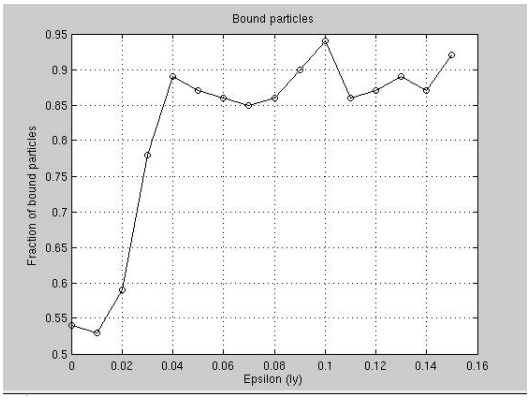


Figure 16: Fraction of bound particles at end of simulation for a range of $\epsilon$ values.

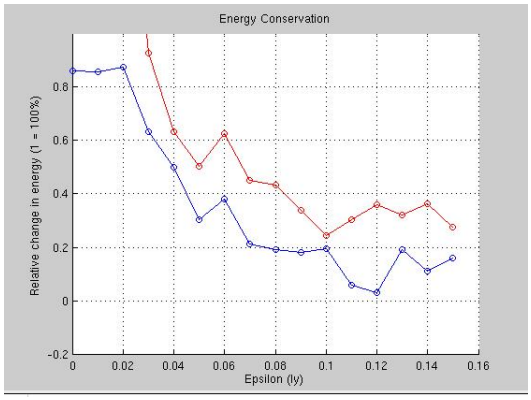


Figure 17: Energy conservation (relative to starting potential energy) for a range of $\epsilon$ values.

The relative energy conservation in figure 17 was calculated using the following formula, using the total energy before the simulation as our scale:

$$\Delta E_{rel} = \frac{E_{after} - E_{before}}{|E_{before}|}$$

where $E_{after}$ can be the total energy of all the particles and the bound particles, respectively.

The results show that the bound particle fraction reaches a value close to the extrapolated value for $N = 100$ (see fig. 3 in [4]) with as low a value as $\epsilon \approx 0.04$. The energy conservation takes a higher value to approach desired levels. Around $\epsilon \approx 0.10$ we notice that the two curves start behaving opposite of one another: When one value of $\Delta E_{rel}$ goes up, the other goes down for the same $\epsilon$. Note also that the curve for the bound particles starts mirroring the fraction in figure 16. This is a result of the low number of particles: We expect the bound particle fraction to fluctuate considerably between simulations with the same starting parameters (again, see fig. 3 in [4]). Thus, when the energy difference for bound particles starts mirroring the number of bound particles, we interpret this as the curve having stabilized (the relative energy diference for all the particles seems to reach a plateau at this point). On the other hand, when $\epsilon < 0.10$, we do not see this behaviour: Both curves are in decline. Hence our (admittedly shaky) basis for choosing $\epsilon = 0.10$ is that after this point, we can no longer distinguish improvements in energy conservation from random fluctuactions caused by variations in the fraction of bound particles.

Using this result, we propose inserting the following values into equation 6: $\epsilon_1 = 0.1$, $\Delta t_1 = \frac{1}{250}$, $N_1 = 100$, $R_1 = 20$. We then get the following formula

that we have used in the remaining simulations for automating the process of choosing an acceptable value for $\epsilon$:

$$\epsilon = \sqrt{\frac{100}{N}\left(\frac{R_0}{20}\right)^3 \frac{\Delta t}{\frac{1}{250}}} \cdot 0.1 = \frac{1}{4}\sqrt{\frac{R_0^3 \Delta t}{2N}} \tag{8}$$

which we expect could be of use in future versions of this project. This formula gave us consistently acceptable results with regards to energy conservation and particle rejection. However, for future projects, we would look at running many more simulations like the one we used to pick the reference value $\epsilon = 0.1$, to ensure that the process is statistically sound (and preferrably with a higher number of particles): While the value we chose has been good enough in practice, we do not claim to have found the optimal value for the case we were studying, and expect future projects to be able to achieve much more accurate results using the method we have outlined above. Another interesting prospect is to seek a closed form solution for a reference value for $\epsilon$ (before scaling it), instead of taking the brute force approach and iterate over a set of possible values like we did here.

One should also keep in mind that the way we scale $\epsilon$ is linear in the sense that we only consider one pair of particles at a time, not looking at "crosstalk" from multi-particle collisions. Furthermore, we have assumed that all the particles have the same mass: If the standard deviation is significantly greater than the 10% of the mean we used to arrive at this result, the above formula may well not be applicable. That said, the particles that receive a change in velocity that is too high for a timestep would tend to be the lighter particles, so we would expect to see the fraction of bound particles decrease while the energy conservation should feel the impact of a greater mass variance less in this case.

## 4.4 Particles ejection and equilibrium

As discussed in the theory part, we observe from animations that the system collapses and reaches a turnaround point at about $1\tau_{crunch}$, which is close to the teoretical value of $0.81\tau_{crunch}$ (i.e. after 6.46 Myrs).

After this time, we can observe on the following plots displaying the number of bound particles that there is a massive increase of unbound particles at this time. This happens when the outermost particles fall onto the very dense core and are accelerated by the strong central gravitational potential. With many near-collisions in a short time, we then reproduce the results in [4]. And we can see, as expected, that without using the smoothing function, we lose a lot more of particles in just a few $\tau_{crunch}$. The results are stabilized when we add epsilon, and the quantity of lost particles decreased. Without the smoothed potential, we keep losing particles even after the remaining particle reach equilibrium. With an $\epsilon$ value that is sufficientlt high, however, this does not happen (compare for instance fig. 21 and 24).

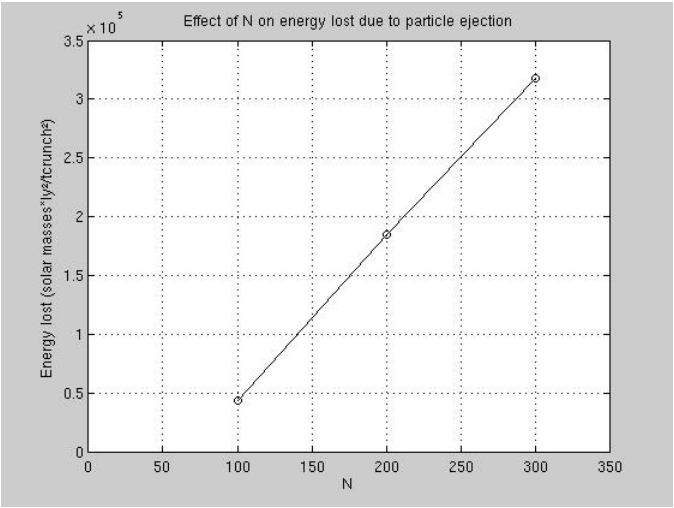


Figure 18: Lost energy as a function of N, using the $\epsilon$ scaling in (6).

When we increase N, the number of initial particles, we increase the percentage of lost particles too. But we keep the total mass constant, which means that as N increases, the particles become lighter. What is then the net effect of this in terms of energy? Since the energies, kinetic and potential, are both a function of the mass of the processed particles, we can see that the quantity of lost energy increases sharply with N and is nearly linear (fig. 18).

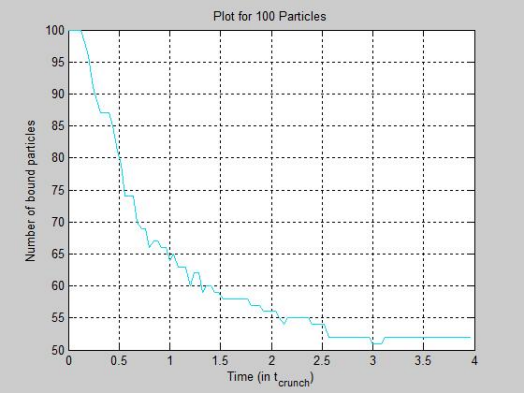


Figure 19: Number of bound particles for N initial = 100 bodies, $\varepsilon = 0$

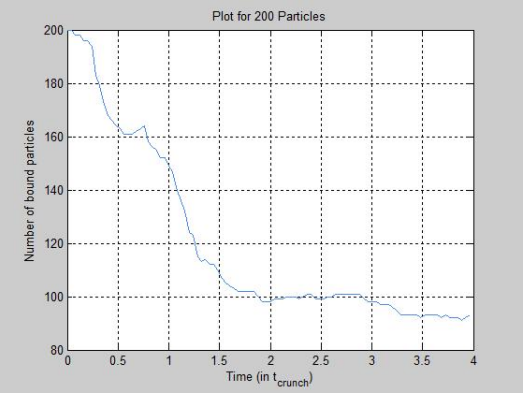


Figure 20: Number of bound particles for N initial = 200 bodies, $\varepsilon = 0$

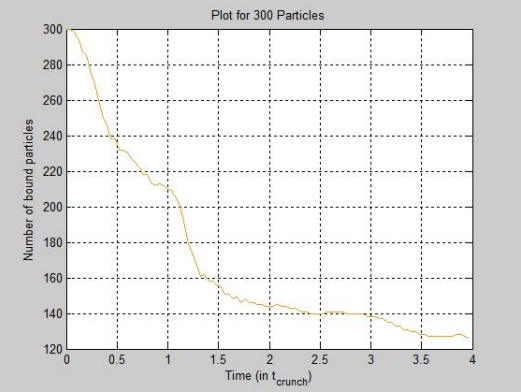


Figure 21: Number of bound particles for N initial = 300 bodies, $\varepsilon = 0$

Without using the modified gravitational potential, the energy is clearly not conserved: a lot of particles are ejected out of the system. For these particles, the kinetic energy increases substantially, and the total energy of theses particles becomes positive.

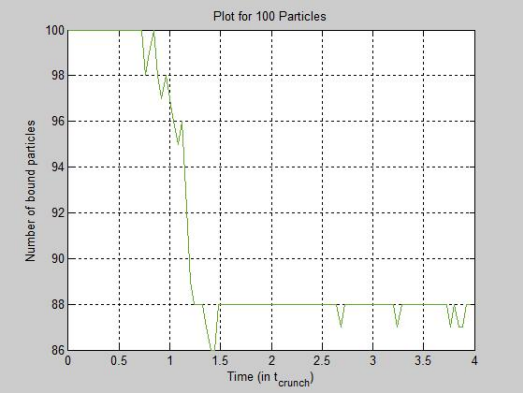


Figure 22: Number of bound particles for N initial = 100, ε = 0.10005

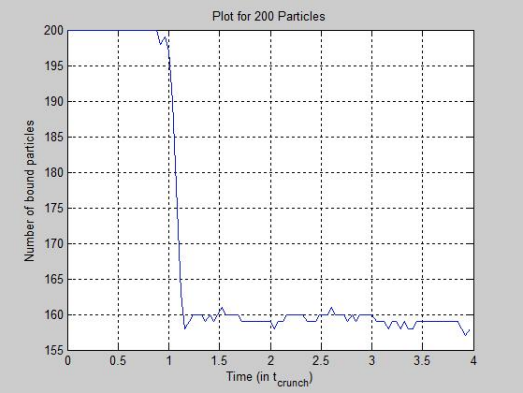


Figure 23: Number of bound particles for N initial = 200, ε = 0,071

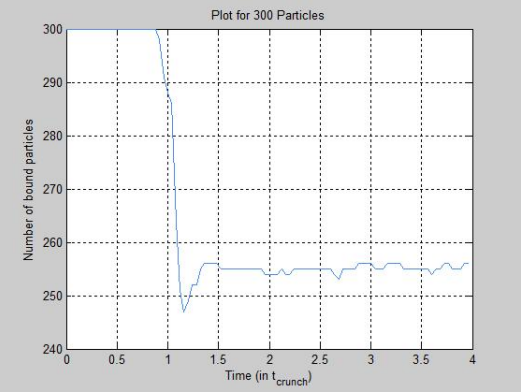


Figure 24: Number of bound particles for N initial =300 bodies, ε = 0,06

After adding the smoothing part in our gravitational potential, we can see that the kinetic energy increases less violently, and the quantity of energy lost is a lot smaller than before: this was expected, since we reduced the numerical instability by adding this ε factor in the calculation of our Newtonian force. Thus, the behavior of particles when they come closer to each other is more stable: we lose less particles than before.

The energy loss due to particle ejection also drops by a considerable amount when we increase $\epsilon$, until $\epsilon \approx 0.4$, at which point random fluctuations take over. This is the same behaviour that was found for the fraction of bound particles with the same $N$.

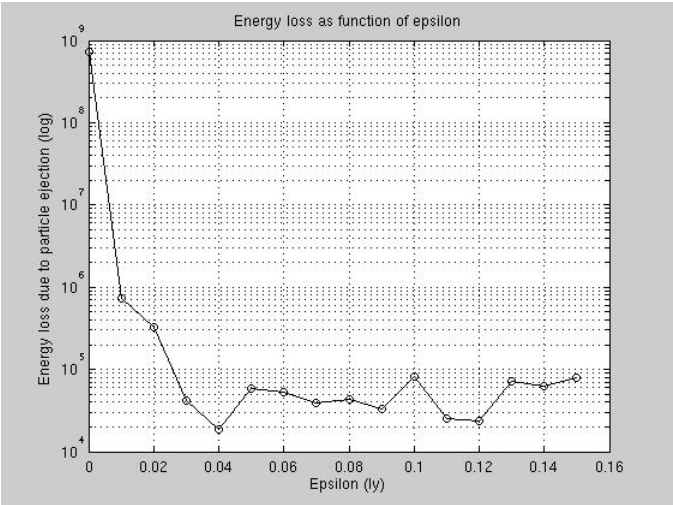


Figure 25: Energy loss ($\Delta E = \sum E_{all} - \sum E_{bound}$) for different values of $\epsilon$, made using $N = 100$.

## 4.5 Virial Theorem

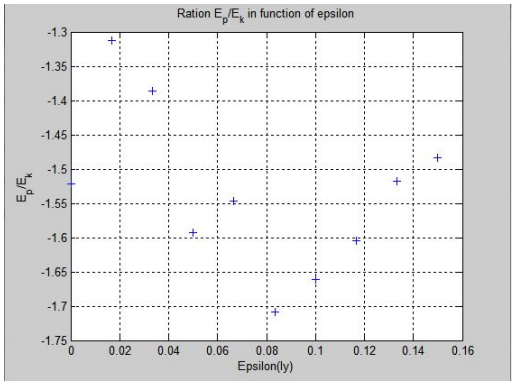


Figure 26: Ratio $\frac{\langle P \rangle}{\langle K \rangle}$ for different epsilons, N = 100

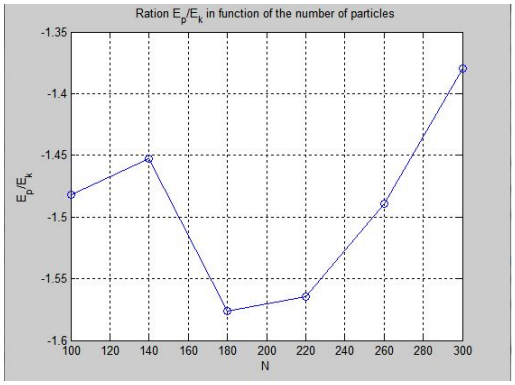


Figure 27: Ratio $\frac{\langle P \rangle}{\langle K \rangle}$ for different N, epsilon $\approx 0.03$

After running the system for a few $\tau_{crunch}$, we evaluate the kinetic and the potential energies for every particles of our bound system. What is displayed here is the value of the average kinetic energy, $\langle K \rangle$ , and the average of the potential energy,$\langle P \rangle$, for our bound system. We can see that the virial theorem is almost met (with a factor of 1.5 instead of 2) in the case where the smoothing

function is used, with an $\epsilon$ of 0.03, and is not met in our initial case (i.e. without using the smoothing function). The smoothing function helps us to take care of the numerical instability generated when two particles are too close. Indeed, with the standard gravitational potential, we lose too many particles. Thus, the introduction of $\varepsilon$ solves this problem.

Why is the virial theorem not satisfied exactly? It turns out that the kinetic energy of the particles is a little too high, as a side effect of the time steps we use (even with smoothing). Thus, even with only considering bound particles, some will have a slightly higher energy than they should due to a near-collision in the particle's near past. These will occur frequently in the dense core of our cluster, preventing us from reaching the exact proportion the virial theorem specifies.

## 4.6 Radial distribution and density in equilibrium

The largest simulation we found it practical to make was a series of four simulations $N \in [500, 2000]$ using the auto-generated $\epsilon$ values. The total running time in this case was about 12.5 hours on a single processor core as expected for $\mathcal{O}(n^2)$ algorithms when the $N = 100$ case takes about a minute to run (depending on processor load and frequency).

For the largest simulation ($N = 2000$), we got the following radial distribution of particles:

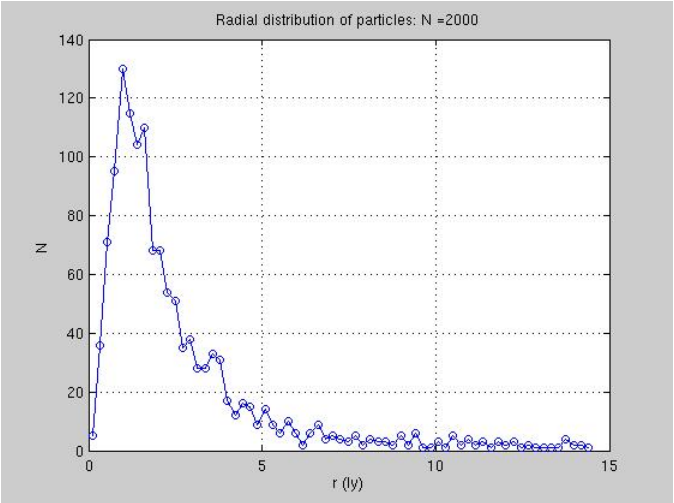


Figure 28: Radial distribution (histogram) for $N = 2000$, with $r$ in the range $[0, \mu_{bound} + 1\sigma_{bound}]$. For the bound particles we got $\mu_{bound} = 4.83$ ly and $\sigma_{bound} = 9.68$ ly (measured from the bound particles' center of mass).

In other words, we get a standard deviation that is about twice as large as the

average. Looking at these values for all four simulations, we see that this is a general trend:

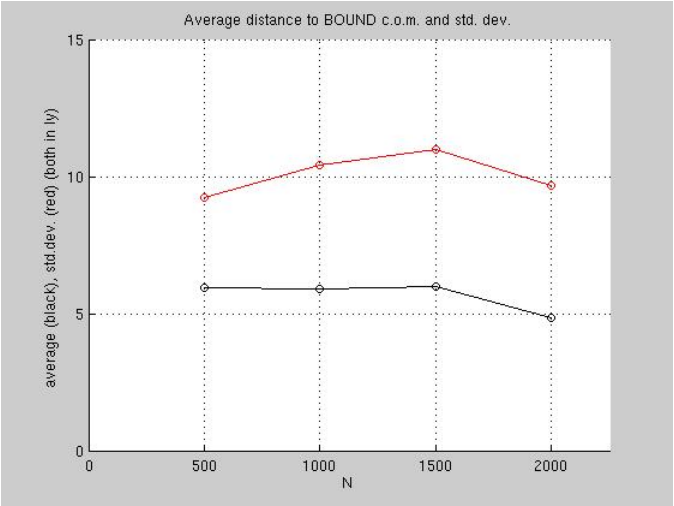


Figure 29: $\mu_{bound}$ (black) and $\sigma_{bound}$ (red), plotted against $N$.

We attribute the large standard deviation to the fact that some particles, while still bound, have been excited to such high kinetic energy levels that they are close to not being bound anymore (these would be the same particles responsible for the virial theorem not holding exactly even for the bound particles). Thus, we expect the standard deviation to vary roughly as the number of bound particles does - more bound particles means a higher chance of high-energy bound particles (less energy lost to the unbound particles). It is clear, however, that we do not have sufficient data to find an expression for $\mu_{bound}(N)$, however we note that the radius does indeed shrink with increased $N$ as noted in [4].

We then proceeded to plot the number density $n(r) = \frac{N(r)}{V(r)}$ for each spherical shell in our histogram and used the method of least squares to fit these data to a curve of the form

$$n(r) = \frac{n_0}{1 + \left(\frac{r}{r_0}\right)^4} \tag{9}$$

where we seek to determine the value of the constant $n_0 = n(0)$ (the core number density) and $r_0$. The latter serves as a scale length for $n(r)$, as seen by the fact that $n(r_0) = \frac{n_0}{2}$, thus $r_0$ is the radius at which we find that the core density has initially halved (it is easily seen that this trend does not continue as we move further out, however).

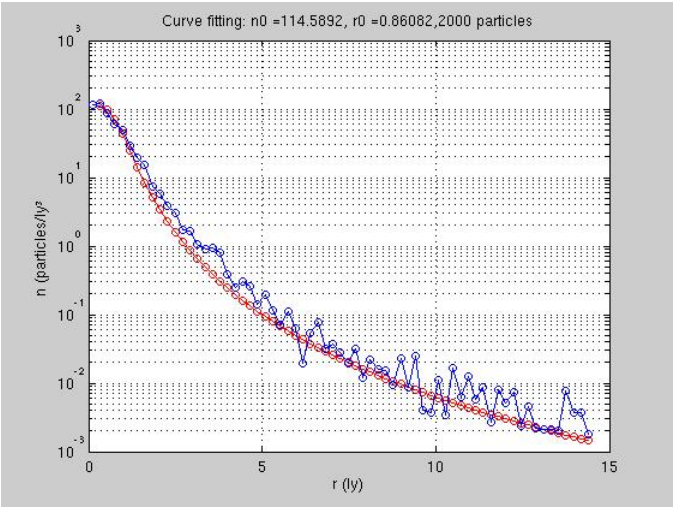


Figure 30: $n(r)$ (logarithmic) plotted against $r$ for $N = 2000$. The blue curve is from our histogram, the red curve is given by (9) with $n_0 = 114.59$ ly$^{-3}$, $r_0 = 0.86$ ly.

We conclude that this gives a nice fit to our radial distribution.

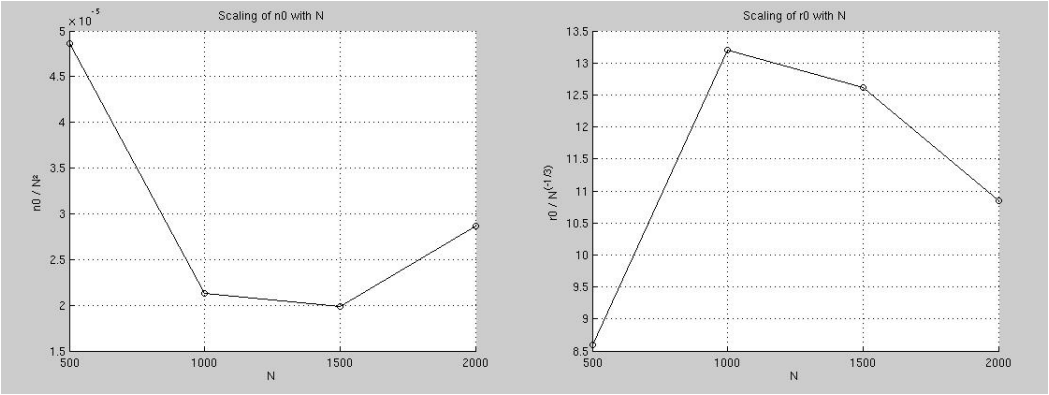


Figure 31: Comparison to the scaling behaviour found for $n(r)$ in [4]. We scaled the axes to match these predictions, but did not encounter the expected scaling behavior, where $n_0 \propto N^2$ and $r_0 \propto N^{-1/3}$: In that case, we would expect the values to remain more or less constant. However, it may well be that we are simply simulating too few particles to be able to reproduce this behavior. Another possible explanation is that we have not kept our $\epsilon$ values constant, but used the dynamic scaling of this parameter from (8).

# 5 Conclusion

The running time is severely affected because we simulate every particle. A better approach might be to divide the particles into cells and then study only the interactions within each cell and between the cells (treating where cells as entities for this purpose).

One could also consider adaptive time steps, however, this is unlikely to scale well, since the amount of near-collisions that demand a smaller time step, will increase sharply with higher $N$. There will always be a particle colliding with another somewhere.

Paralellizing the code would indeed be beneficial (due to time constraints, ironically, we were forced not to implement this as we would then need to find out how to do this in Visual C++). It is especially beneficial when doing series of many simulations that should result in a plottable data set over all the values of $N$ we are interested in. One could then aim to distribute the load evenly over each processor. Furthermore, even with just one simulation, the $\mathcal{O}(n^2)$ parts of the code (gravitational forces, potential energy), could be divided among processors (each core being given a subset of the particle pairs/vertices to work on).

# References


[1] Ø. Elgarøy: Lecture notes in AST 4220, University of Oslo, fall 2009, pp. 31-33: https://www.uio.no/studier/emner/matnat/astro/AST4220/h09/course-material/lectures.pdf

[2] Ø. Elgarøy: The Spherical Collapse Model: https://www.uio.no/studier/emner/matnat/astro/AST4320/h12/ undervisningsmateriale/spherecollapse.pdf

[3] J.A. Peacock: Large scale surveys and cosmic structure (Lectures delivered at the 2002 Tenerife Winter School, "Dark matter and dark energy in the universe"), pp. 15 - 19: http://arxiv.org/abs/astro-ph/0309240

[4] M. Joyce, B. Marcos, F. Sylos Labini: "Cold uniform spherical collapse revisited", 2010: http://arxiv.org/abs/1011.0614

[5] F.K. Hansen: "The Virial Theorem", lecture notes in AST 1100, University of Oslo, fall 2013: https://www.uio.no/studier/emner/matnat/astro/AST1100/h13/ undervisningsmateriale/lecture5.pdf

[6] R. Feres: "Math 350 Fall 2012 - Homework 8 Solutions", Washington University in St. Louis: http://www.math.wustl.edu/∼feres/Math350Fall2012/Math350F12HW08Sol.pdf

[7] Rubinstein: <title unknown>, photocopy from University of Colorado at Boulder: http://ecee.colorado.edu/~fmeyer/class/ecen5322/rubinstein.pdf

[8] T. Clifton, P. G. Ferreira, A. Padilla, C. Skordis: "Modified Gravity and Cosmology", 2011: http://arxiv.org/abs/1106.2476

[9] A. Clement, A. Clement, G. Fenchel, J. Fenchel, J. Lycnh: "Numerical Intregration Methods", 2008: http://www.challenge.nm.org/archive/07-08/finalreports/82.pdf

[10] "The Leapfrog Integrator", photocopy from the University of Drexel: http://einstein.drexel.edu/courses/Comp_Phys/Integrators/leapfrog/