Universiteit Leiden
The Netherlands

Opleiding Informatica

How many stars does it take to form a binary?

Automating analysis of binary formation during core collapse

Eva van Houten

Supervisors:
Aske Plaat & Simon Portegies Zwart

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden Observatory

Abstract

Tanikawa et al. were the first to simulate and examine binary formation during the core collapse of a globular cluster. We try to replicate their findings, and automate the analysis of the N-body simulation data. Our simulation starts with an initial run from a randomised cluster model, followed by a more detailed simulation with smaller time steps. In our analysis of the data we did not manage to find a binary formation event, only exchanges between a binary and a third star. We did gather some valuable insights into the challenges of the automation process. Future work will need to formulate a better binary finding algorithm, solve issues of accuracy due to the chaotic nature of the system, and refine the rest of the analysis in order to create a fully automated simulation and analysis program.

Contents

1 Introduction

A globular cluster is a slow creature. This ball of up to a million stars evolves over billions of years. Every globular cluster we see in the night sky gives us only one snapshot of the slow processes going on, so if we want to know what happens inside, we cannot simply observe it. Knowing more about these clusters could shed light on many astronomical topics of interest, such as star formation and exoplanets [HH03, p23-25]. To really understand these systems, where the density of stars is very high, it is vital to examine the gravitational interactions between the stars. If we want to study these star dynamics in detail, we have to turn to N-body simulations. The information in this section comes from the book *The Gravitational Million-Body Problem* by Heggie and Hut [HH03], which explores different aspects of N-body simulations as they relate to star clusters.

N-body simulations are, in their essence, rather simple. We usually assume each star is a size-less point mass in empty space, although there are N-body models that do give each particle a size. We divide the continuum of time and space up into discrete chunks, as small as is feasible. For each chunk of time, we then apply Newton's equations of gravity to calculate where the particles will move to, and what their new velocity will be. Modern N-body codes use some clever mathematical tricks to speed up this process, but no other physics needs to be involved. Using these N-body simulations, we can model astronomical objects like globular clusters very accurately. As long as collisions between stars are extremely rare, like they are in these clusters, we only need to take gravitational interactions into account.

One of the findings that result from both sophisticated theoretical models and N-body simulation, is the phenomenon of core collapse. We have also observed globular clusters in this state, so we know it occurs in real clusters as well. The core of the cluster gets much denser, and in this dense mass of stars, a binary can form. A binary star is nothing more than two stars orbiting each other. These two stars are important to the cluster as a whole, however, as their energy by itself can halt the collapse of the entire cluster. Interactions with this binary speed up the other stars, slinging them outward and reversing the collapse.

It used to be widely believed that this binary formed through an interaction between three stars. These stars would meet within the dense core, perform a dance, and out would come one ejected star and a binary bound together by gravity. Tanikawa, Hut and Makino performed the N-body simulations, and found there were often more stars involved in a more complicated process [THM12]. In a second paper they elaborated on this, and made a call for their tedious, manual analysis of the simulation data to be automated [THHM13].

So how *do* we automate the data analysis process used to describe interactions of multiple stars during binary formation? This is the question that we attempt to answer in this bachelor's research project, supervised by Aske Plaat from LIACS and Simon Portegies Zwart from Leiden Observatory.

First we will give an overview of the relevant astronomical theory and terminology in section 2. We have tried to replicate Tanikawa's results, summarised in section 3, but attempted to skip the manual analysis and immediately automate the process in our code. How we did this is detailed in section 4. The results from our simulations and their subsequent analysis can be found in section 5.

We discuss our findings, and the many challenges we came across in section 6, and conclude with recommendations for future work in section 7.

2 Background

Before we can discuss our simulations, we need to introduce some astronomical terminology and theory. We will take a look at the processes going on inside a globular cluster, look at its evolutionary stages, and define some formulas and quantities that will be referenced later in this thesis. We have used three books to source most of the information for this section: the aforementioned *The Gravitational Million-Body Problem* by Heggie and Hut [HH03], *Galactic Dynamics* by Binney and Tremaine [BT08], and *Astrophysical Recipes* by Portegies Zwart and McMillan [PZM18]. When some chapters or pages are particularly relevant, we will mention them in the corresponding subsection.

2.1 Globular clusters

A globular cluster is a collection of ten thousand to a million stars, held together by the gravity between the stars, which makes the cluster roughly spherical in shape. These clusters can be found in most galaxies, including around the edges of our own Milky Way. How these clusters form is still a bit of a mystery. Counter to what you might think, the stars in the cluster did not form at the same time from a molecular cloud, as is the case for smaller open clusters. The stars in a globular cluster are all different ages, though generally they are almost as old as the universe itself. The current theory is that these clusters are formed through collisions between clusters. We will be studying the gravitational dynamics of globular clusters long past their formation, however that may have happened. The age and composition of the individual stars is not our concern.

2.2 Two-body relaxation

The main process we are dealing with in studying cluster dynamics is *two-body relaxation*. This is what drives the relaxation process described in section 2.7.1. In the cluster, stars continuously pass each other at close enough distance that the gravity between them has a significant effect on both of their orbits. The effect can be quite small. In fact, it is the *weak interactions*, with a smaller effect on the stars' energy, rather than the rarer strong interactions that drive this process [HH03, p139].

2.3 Negative heat capacity

Gravitational systems show a strange, counter-intuitive property: negative heat capacity. Heat in any system could simply be taken to be the average velocity of the particles. The faster the particles in a gas, liquid, solid, or star cluster move, the higher its temperature. When gravity comes into play, something odd happens. When particles are cooled down (thus slowed down), they respond by speeding up! This can be explained by looking at a satellite in orbit. The speed at which it flies around the Earth counteracts the force trying to pull it down to the surface. If we slow down the satellite, it therefore drops into a lower orbit. This lower orbit requires a higher speed to be maintained, and so the satellite paradoxically speeds up. It might seem that this violates energy

conservation, but this is not the case. There is a net energy loss from the satellite, due to the initial force slowing it down. Its potential energy decreases more than its kinetic energy increases, and so there is no net increase in energy.

2.4 Cluster evolution

In the lifetime of a globular cluster, we can distinguish three different eras: relaxation, core collapse and post-collapse [LS78].

2.4.1 Relaxation

In simulations of clusters, we usually start with stars distributed spherically, with different velocities. This is probably somewhat similar to how the cluster is actually formed. But even if we throw some non-moving stars into space, they will behave similarly enough after a while. The stars attract each other, and so the cluster collapses. The stars are all on different paths, however, and so they do not all collide in the middle, but pass each other to form a variety of orbits. The cluster will contract and expand a few times, but soon enough the stars are all mixed together. The initial conditions have been forgotten, and a dynamical equilibrium, a quasi-steady state, has been reached [HH03, p9-10].

2.4.2 Core collapse

The quasi-steady state after relaxation does not last. Energy is transported from the inner regions to the outside, which causes the core to contract. Some stars are flung out and escape from the cluster. See figure 1 for a simulation of a cluster before and after core collapse.

We would expect core collapse to continue indefinitely, were it not for binaries. When the core gets denser, it becomes more likely for the stars to interact in such a way that *binary stars* form: two stars end up orbiting each other in a tight circle. Just one of these binaries is enough to halt the collapse of the entire cluster! The energy contained in the binary is on the order of the energy of the whole cluster.

2.4.3 Post-collapse

The newly formed binary star increases the kinetic energy of the stars it encounters, and so the core can re-expand. The core then contracts somewhat again, and re-expands and contracts a few more times. We call this phenomenon *gravothermal oscillations*.

2.5 Modelling clusters

Globular clusters, known in dynamical terms as *dense stellar systems*, have been modelled in four different ways. Each of these methods has a different accuracy. The higher the accuracy, the more difficult it is to compute. Heggie and Hut [HH03, ch9] discuss these models in much more detail. I have summarised their chapter on methods in this section. The subsection on N-body simulations uses additional information from chapter 2 of Astrophysical Recipes [PZM18, ch2].

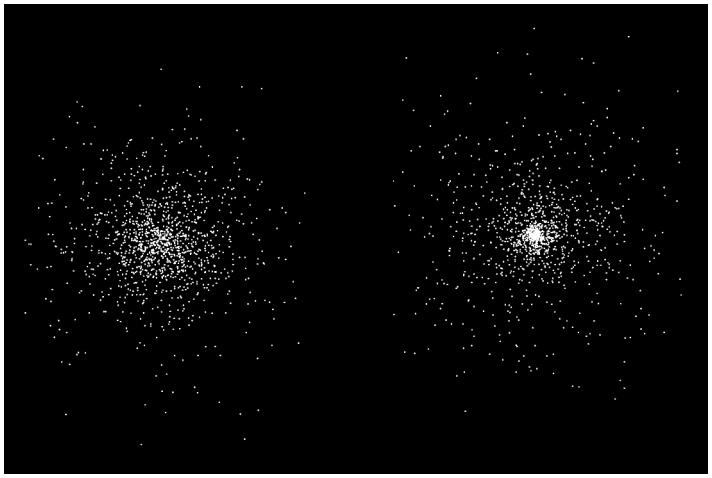


Figure 1: A simulation of a cluster, showing the cluster before and after core collapse [HH03, p187].

2.5.1 Theoretical models

Since N-body simulations were initially extremely labour-intensive, as they had to be computed by hand, for a long time dense stellar systems were modelled using formulas. There exist no formulas to describe any system of more than two bodies exactly. These theoretical models, therefore, only approximate the behaviour of a cluster.

The simplest models are the evaporative ones. They use only the mass, size and mean square speed of the stars as parameters. They can only be used to determine global measures such as energy equilibrium.

More complex are the models that apply gas physics to star clusters. We get a surprisingly accurate model if we treat the stars as molecules in a gas. It was a gas model that first showed core collapse. The main problem here is that the forces between molecules in a gas only act on short distances, while gravity (although it decreases fast, by the square of the distance) has enough of an effect even across the whole cluster.

Sophisticated formulas called Boltzmann and Focker-Planck equations describe clusters in quite high detail. They can be used for many applications where a simulation is too costly, although studying individual orbits is still out of the question.

2.5.2 N-body simulation

N-body simulations, such as we are exploring in our research, are the most accurate model of dynamical systems. The movement of every single star is calculated per tiny time step, which makes it possible to study the behaviour of the cluster at the level of individual stars. Note that this is technically a theoretical model as well, since we are not using telescopes to observe stars in the sky. It can still be said that one "observes" something in a simulation though, and so it is not quite as theoretical as the models named previously.

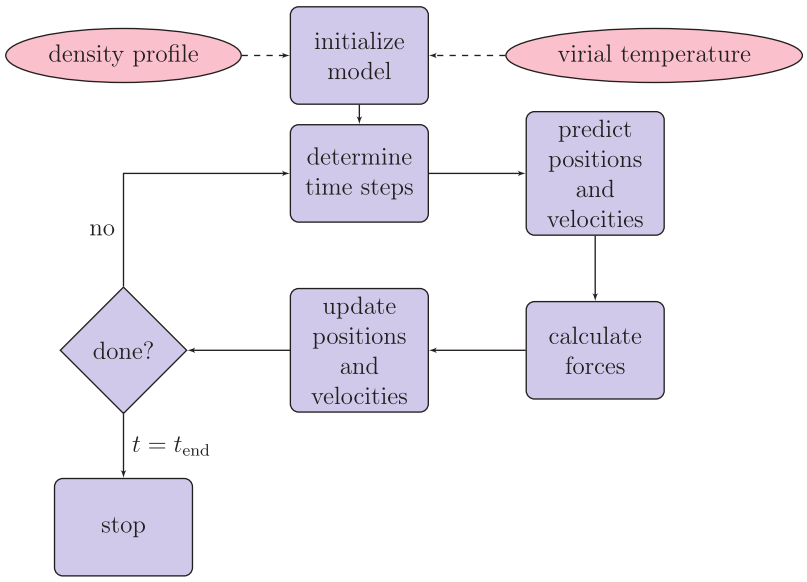


Figure 2: This flowchart shows the process that a *direct N-body code* uses to perform the simulation time step by time step [PZM18, ch2 p14].

We use numerical integrators to perform these simulations. Figure 2 shows an overview of how one of these integrators (specifically a *direct N-body code*) works. For each time step, the integrator predicts the positions and velocities of all the particles. Then it calculates the forces that the particles exert on each other. The positions and velocities are updated according to these forces, and the cycle begins again. The time step that is used for each iteration varies. We will be using a similar integrator called a *pure N-body code*. The direct N-body code has multiple parameters, while in a pure N-body code we can only adjust the time stepping criterion, also called the accuracy parameter.

These methods can get quite complex in order to achieve better performance, but all they really do is apply Newton's formula for gravitational force. This formula gives the gravitational acceleration of an object i due to a group of objects j [PZM18, ch2 p1]:

$$\boldsymbol{a}_i = G \sum_{j \neq i}^{N} m_j \frac{\boldsymbol{r}_j - \boldsymbol{r}_i}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3}$$

Here G is the gravitational constant, m_j is the mass of each of objects pulling on i, and $\boldsymbol{r}$ is the position vector of an object in space.

Between N-body simulation methods the accuracy can vary too. Most commonly used numerical integration methods can now be run on the order of thousands to tens of thousands of stars. More approximate methods like *tree codes* exist that can be used for up to a million stars in reasonable computation time. They manage this by calculating interactions between close by stars precisely, but approximating more distant forces.

Even the pure integrators, that strictly apply Newton's laws, have errors. There will always be an error in a simulation, simply because we have to choose a time step, and every variable has limited

precision. A double-precision floating point number, as often used in modern computers, has 64 bits to store its data. At some point we simply run out of digits. A cluster is a chaotic system, the consequence of which is that even a slight change in the state of the system at a given time, will result in a very different outcome after some time.

Recently a code has been developed that tries to minimise the error to such an extent that the simulation is reversible. Normally this is not possible, because every tiny error builds up due to the high amount of chaos in the system. Brutus [BPZ14] achieves this by using a Bulirsch-Stoer integrator and variable size of the floating point numbers, also called the word length. We will not be using Brutus, because it is too slow to use for more than a few stars.

2.6 Soft and hard binaries

Binary stars within a cluster can be divided into two categories, based on how tightly the two stars are bound together. This is measured by the binding energy [BT08, p617]:

$$E_b = \frac{Gm_1m_2}{|r_1 - r_2|} - \frac{1}{2}\mu(v_1 - v_2)^2$$

Where the reduced mass $\mu = \frac{m_1m_2}{m_1+m_2}$. Note that the first term on the right is the gravitational potential energy ($E_g = \frac{Gm}{r}$), and the second term is the kinetic energy ($E_{kin} = \frac{1}{2}mv^2$). We are subtracting how much the particles try to move away from each other from the gravitational pull between them. E_b is defined here as being more positive the more the particles are bound.

For an equal-mass system such as we are studying, $m_1 = m_2$, and thus this simplifies to:

$$E_b = \frac{Gm^2}{|r_1 - r_2|} - \frac{1}{4}m(v_1 - v_2)^2$$

Binaries with $E_b < m\sigma^2$ are called *soft binaries*, and when $E_b > m\sigma^2$ they are *hard binaries* [BT08, p617]. The velocity dispersion σ gives a measure of the average velocity of the stars in the cluster. Any star that encounters the binary will, on average, have a velocity σ. This makes $m\sigma^2$ the average kinetic energy of the stars in the region for which σ is the velocity dispersion. This will not be the same throughout the cluster.

To express this average kinetic energy, we will use a measure originating in gas physics, kT, which is related to the energy of the particles in a system. $\frac{3}{2}kT$ is the average kinetic energy of the stars in the cluster. $1kT = 1/(6N)$ at $t = 0$, and we will be using this value for kT in our work, like Tanikawa et al. do [THM12].

When a soft binary encounters another star, the star's kinetic energy will, by definition, usually be higher than the binary's binding energy. This gives it the power to knock one of the binary's stars out of its orbit. The binary will be no more. We call this phenomenon *ionisation*, since it is similar to an electron being knocked out of its orbit around the atom to create an ion. A soft binary can also *evaporate* due to many long-distance encounters that gradually decrease the binding energy.

For a hard binary, it is more likely that one of two other things happen. If the third star has not enough energy to significantly affect the binary, which is more likely than not, because of the high binding energy, it simply passes by: a *fly-by*. It is also possible that the third star hits the binary in such a way that it knocks one of the stars out, and takes its place in the binary. We call this an *exchange*.

Soft binaries tend to be disrupted by their encounters with other stars. Hard binaries, in contrast, are often bound closer together after an encounter. The third star that does a fly-by picks up energy from the hard binary, which then paradoxically increases its binding energy (as for why, see the explanation on negative heat capacity in section 2.3). This leads to *Heggie's law*: soft binaries tend to get softer, and hard binaries tend to get harder [BT08, p621]. It is of course still possible for a soft binary to "harden" into a hard binary, since these are not strict rules, but only averages. This will be important when we try to find the interaction that forms or hardens the first hard binary during core collapse.

2.7 Timescales

In the study of globular clusters and N-body simulations, there are several relevant timescales and time quantities that we will use.

2.7.1 Relaxation time

A useful measure for time when we are dealing with dynamical systems, is the *relaxation time*. The relaxation time is the time which it takes for the state of the cluster to become independent of its starting conditions. In other words, if you threw some stars into space, close enough together that they would form a cluster, after one relaxation time you cannot distinguish it from any naturally occurring cluster. The cluster reaches a *quasi-steady state* in this time.

We will specifically see the *half-mass relaxation time* play a role. The relaxation time varies greatly throughout the cluster, and so we use the relaxation time at the *half-mass radius* r_h (the radius which contains half the mass of the cluster) as a global measure.

An estimate of the relaxation time can be derived from a certain Fokker-Planck equation [HH03, p146]. There is no precise formula that can be derived. The definition that is used is rather a convention. Spitzer (1987) defines the half-mass relaxation time as [Spi87]:

$$t_{rh} \simeq 0.138 \frac{N^{1/2} r_h^{3/2}}{(Gm)^{1/2} \ln \Lambda}$$

Here G is the gravitational constant. We study a system with equal-mass stars, so $m = M/N$, where M is the total mass of the cluster and N is the number of stars.

$$t_{rh} \simeq 0.138 \frac{N r_h^{3/2}}{G^{1/2} M^{1/2} \ln \Lambda}$$

Here $\ln \Lambda = \ln \gamma N$ (we call this the Coulomb logarithm). γ is a constant, which has been determined through N-body simulations to be $\gamma \approx 0.11$ in equal-mass systems [GH94].

Tanikawa et al., the authors of the paper we are building upon, use this formula for the half-mass relaxation time:

$$t_{rh} = 0.138 \frac{N r_h^{3/2}}{G^{1/2} M^{1/2} \ln(0.4N)}$$

Note here that $\gamma \approx 0.4$ used to be the accepted value for γ, but this had been corrected to 0.11 by simulations years before they published their paper [HH03, p142].

2.7.2 Crossing time

Another timescale we will use is the *crossing time*. This is the time it would take a star with average velocity to cross from one side of the cluster to the other side. The *core crossing time* is the same, but then for a crossing of only the core. To calculate the crossing time, we can simply divide the relevant radius (in this case the core radius r_c) by the average speed of the stars in the region (v_c) [THM12]:

$$t_{cr,c} = \frac{r_c}{v_c}$$

2.7.3 Co-moving time τ

Tanikawa et al use a "co-moving time" τ for their graphs. It is calculated using the following formula:

$$\tau = \int \frac{dt}{t_{cr,c}}$$

Here $t_{cr,c}$ is the core crossing time defined in section 2.7.2 By scaling the time this way, we use more of the graph's x-axis for moments when the core crossing time is shorter. During core collapse, as the core reaches its minimal radius, we therefore zoom in on what is happening. This way we can show more detail during the time when the most interesting things happen.

3 Previous work

For this research project, we will try to replicate and expand upon the work of Tanikawa, Hut, Makino and Heggie. In this section we will introduce their two relevant papers.

3.1 The first simulations - Tanikawa, Hut & Makino 2011

It was always thought that binary star formation in globular clusters happened in the following way [HH03]: when core collapse happens, the core gets much denser. This makes it more likely that three stars meet in a single interaction. These three-body interactions are what form soft binaries. These binaries then gradually harden through interactions with other stars, until they are stable enough to not be separated anymore. These hard binaries then drive the post-collapse evolution of the cluster.

There had been some suspicions that this was not the whole story. Tanikawa et al were the first to really run the simulations and study the process in detail. They chose to use a 10kT threshold where they consider a binary to be hard enough. Compared to the hardness threshold given by the average kinetic energy, $\frac{3}{2}kT$, this excludes many binaries that could be considered hard.
They first ran the simulation with a larger Δt, and then increased the time resolution to $0.01t_{crc}$. They start the simulation from Plummer's model, a common approximation of a star cluster. We will do the same.
The analysis of the data was no easy task. The data from the simulation had to be combed through by hand to pinpoint the moment of binary formation, and the interaction had to be untangled with the help of complicated graphs such as the ones in figure 3.

These graphs can be plotted mostly automatically, but choosing which of the 1000 stars to plot had to be done manually. This is the crux of the analysis. Other automation factors are choosing how far back in time to calculate every metric, and representing the interaction in some way. There is no straight-forward cutoff for which star is "involved" in an interaction, since all stars in the cluster are involved with every interaction that happens (though the force they exert may be minute).

The manual nature of the data analysis limited the authors in the number of formation events they could study. This paper shows six of them, which is of course not enough to base sound statistical conclusions on. The results of the runs do suggest, however, that binary formation is usually more complicated than the theory had predicted. Only one of the formations, the N=4k case, is a three-body interaction. The binary that is formed is already a hard binary though, and so the slowly hardening story does not apply even here. In all other cases, the binary is formed through interactions between up to six stars, and often multiple encounters happen in rapid succession.

3.2 More elaborate analysis - Tanikawa, Heggie, Hut & Makino 2013

In a second paper [THHM13], the same authors expanded on their analysis of one of the formation events. They used a formula they call a *work function* to measure the influence of a star j on the binding energy of a tuple i of stars. A tuple is a group of stars bound together, orbiting around each other. A binary star is a tuple of two stars. The work function is the integral, from a certain t_0 to the variable time t, of the *power function*:

$$\dot{E}_{i,j}(t) = -\sum_{k}^{n_i}[\boldsymbol{f}_{i_k,j} \cdot (\boldsymbol{v}_{i_k} - \boldsymbol{v}_{cm})]$$

Here $\boldsymbol{v}$ is the vector velocity of a given component, and $\boldsymbol{v}_{cm}$ is the velocity of the centre of mass of the tuple. $\boldsymbol{f}_{i_k,j}$ is the force exerted by star j on component k of tuple i (given by Newton's law of gravitation):

$$\boldsymbol{f}_{i_k,j} = -Gm_j m_{i_k}(\boldsymbol{r}_{i_k} - \boldsymbol{r}_j)/\left|\boldsymbol{r}_{i_k} - \boldsymbol{r}_j\right|^3$$

Here $\boldsymbol{r}$ is the position vector for a certain star.

We are going to use this work function as a metric for how important a certain star is in a formation event. In figure 4 we see an example of work calculated for four stars on a binary star. Ideally, we would be able to produce such a plot automatically for the time around the binary formation, and

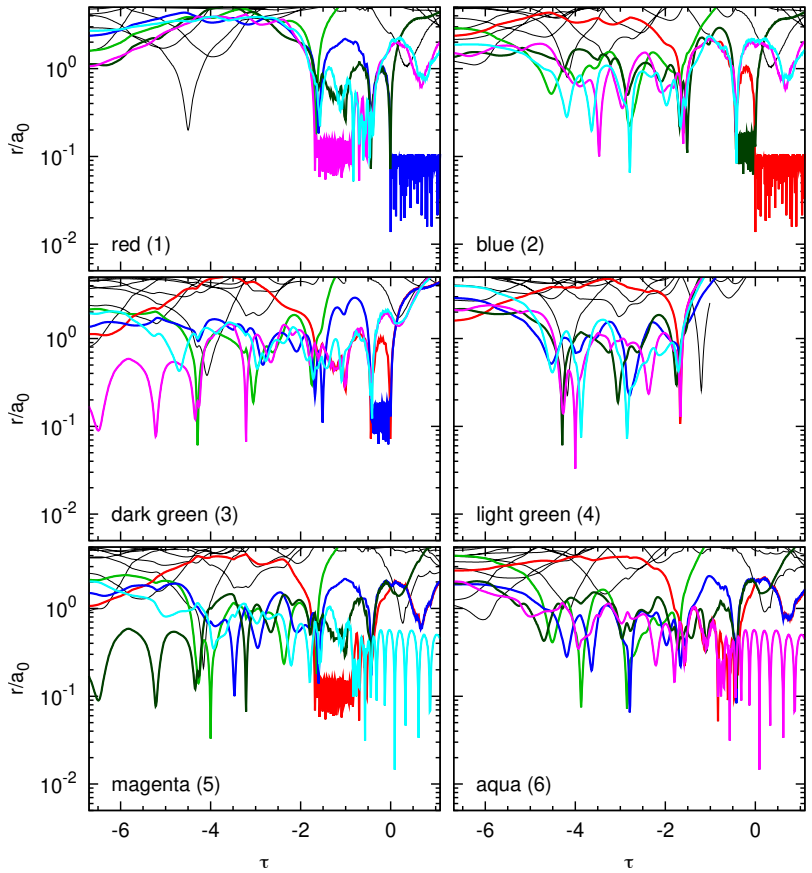


Figure 3: These graphs, from figure 7 in [THM12], show the distances between 6 stars involved in a binary star formation. The top left image, for example, shows the distance on the y-axis between star 1 and the five other stars. We can see that star 1 (red) and star 2 (blue) end up forming a binary, because they stay very close together in a rapidly oscillating pattern. Before then, 2 (blue) and 3 (dark green) form a temporary binary, just like 1 (red) and 5 (magenta).

the few stars that had the most influence on the process.

The authors eventually produced a figure resembling a Feynman diagram (see figure 5) to visualise the series of interactions that leads to the first hard binary.

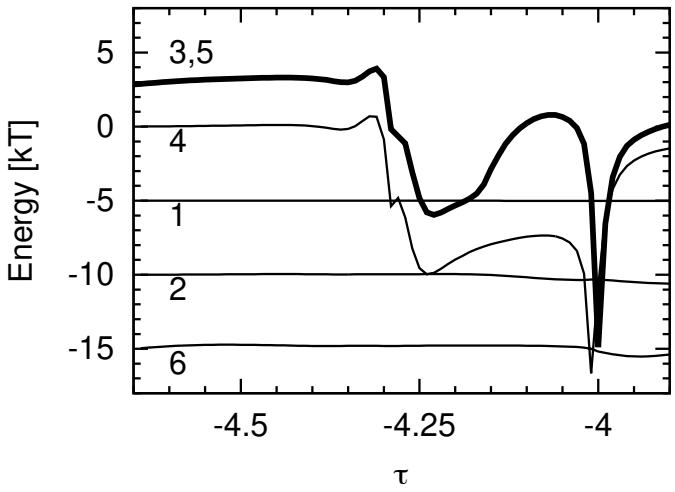


Figure 4: This graph shows the work function calculated for stars 1, 2, 4 and 6, on binary star 3,5. The binding energy of the binary is also shown in bold. Note that the work lines are shifted vertically in order to be able to distinguish them; at the start of the integration (here -4.65) we actually set the work to 0. It is obvious from this graph that star 4 is responsible for most of the change in binding energy of the binary.

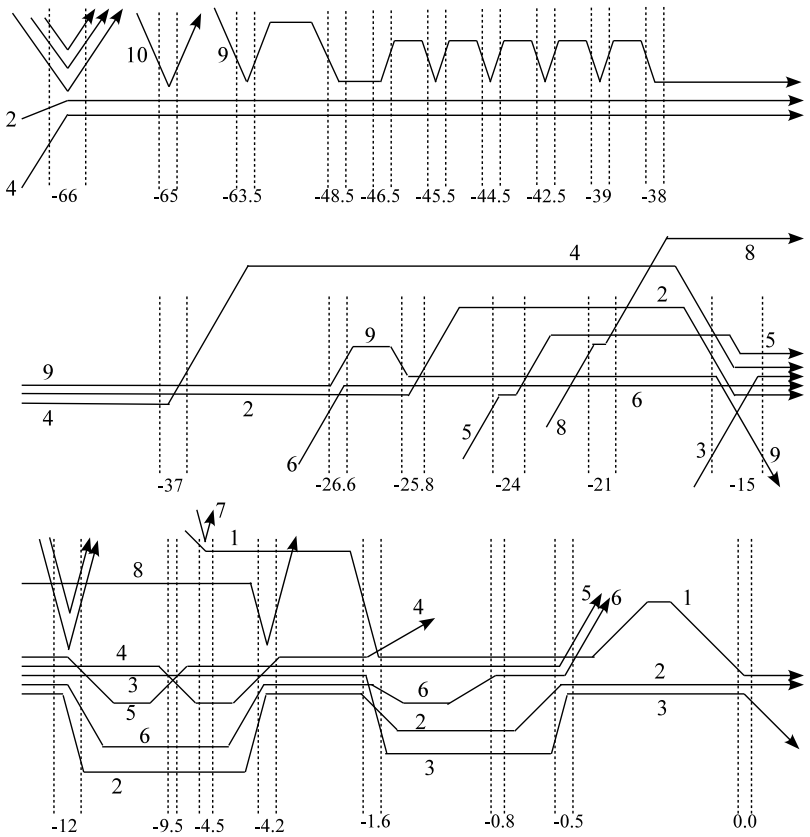


Figure 5: This is the Feynman-like diagram from figure 3 in [THHM13]. It represents a series of interactions between stars labelled 1 through 8. The time is scaled by τ (see section 2.7.3). When two or more lines run parallel close together, it means those stars are bound in a tuple. The lines "bouncing" off other lines are stars that interact, but do not become bound. Eventually stars 1 and 2 end up as a binary star at $\tau \equiv 0.0$.

4 Method

4.1 Software

4.1.1 Amuse and python

For the simulation, we use the Astrophysical Multi-purpose Software Environment, or AMUSE [PZvEP+21][PZM18][PZMvE+13][Pvd+13][PZMH+09]. This is a framework to integrate different so-called "community codes", written in languages like C, C++ and FORTRAN, into a user-friendly Python API. Most scientists prefer to use a garbage collected language like Python, because they do not have to concern themselves with things like memory management. Python is easy to write and read, and it has collected a huge ecosystem of scientific packages over the years. The downside, however, is performance. As an interpreted language which needs to run garbage collection regularly, Python simply cannot come near the performance of a compiled language like C. AMUSE aims to fix this dilemma, by providing high-performance codes that can easily be called from Python. Most codes provide effective optimisations such as multi-threading and GPU-acceleration.

For many data structures AMUSE builds on a popular package among scientists: Numpy. Numpy uses a similar approach, providing functions written in highly optimised C, that can be called from Python on Numpy-arrays. It is these arrays that AMUSE wraps with a unit system.

The data structure from AMUSE that forms the basis of the simulation is the `ParticleSet`. This set contains `Particle` objects, which contain all parameters of the particles in the simulation, such as position and velocity. The `ParticleSet` also provides functions to calculate certain cluster metrics, such as the core radius, virial radius and total energy of the cluster.

4.1.2 PH4: a Hermite integrator

The specific community code we are using is called PH4. It implements a fourth-order Hermite integrator. This integrator scheme is the preferred choice for N-body simulations with large N. It is easy to use and very fast, although it is not reversible and the error builds up rapidly over time [PZM18, ch2 p15-16].

For the integrator parameters, we simply mimic what Tanikawa et al. use, in order to come to similar results. Their smallest cluster consists of $N = 1000$ stars, all of equal mass. For the accuracy parameter, they use $\eta = 0.01$ and $\eta_s = 0.0025$ (its value at the start of the simulation). They use the standard N-body units, where G is scaled in such a way that $G = M = r_v = 1$. G is the gravitational constant, with units $Nm^2 * kg^{-2}$, M is the total mass of the cluster, and r_v is its virial radius.

4.2 Testing and validation

In scientific computing, validation of results is always a challenge. How do you know if your program is completely correct, and yield scientifically valid results? Some of the uncertainty can be taken away by testing.

4.2.1 Integration tests

When setting up the simulation, it was difficult to check whether the integration was working as it should. This made refactoring the code very problematic, as the simulation could break at any time in undetectable ways. The solution was to write an integration test. Integration tests are used to test a piece of software in a way similar to how it will be used in production. The most straight-forward method to do this is to run the program, with the same parameters every time, and compare the output to some known "correct" output.

We wrote a script that did exactly that, with the output being a png plot of the final state of the cluster after three integration steps. It soon became clear that this plot was not generated exactly the same way every time. Axes shifted slightly, due to some internal matplotlib logic. We solved this by saving the final state of the simulation as a simple csv file, and comparing those.

We also created a test for snapshot loading. This test loads a snapshot, calculates one time step, and then compares the result to the previous test where we started from scratch.

The tests had some issues, such as the results not quite matching up. The output varied past a few decimal points. This could be due to an inherent error in the simulation, or due to the limited accuracy of floating point numbers [Gol91]. We decided it was outside the scope of this research project to dive into this problem, although we will comment on accuracy in our discussion (section 6.1).

4.2.2 Energy conservation

A simple way to check the physics of a simulation, is to see whether energy is conserved. This is one of the fundamental laws of physics: energy is never created or destroyed, but only changes from one form into another. During a simulation, energy changes back and forth from kinetic into potential energy. If we add up all forms of energy, this number should be roughly constant. This means that the integrator, which always introduces some error into the calculation, is accurate enough to be able to trust the results. Over an interval of one crossing time, the energy error should be below 10^{-4} times the total energy, although some say this metric is too high [PZM18, ch2 p48]. The smallest possible error for a simulation with a Hermite scheme is 10^{-14}.

Most of our simulations showed an energy error at most 10^{-8} times total energy, although there was one run that reached $\sim 10^{-7}$.

4.2.3 Unit tests

We decided to not use test driven development for this research project, as the code will probably be only used once, and will not need to be maintained and developed for a long time. We did use some form of unit testing for the functions that calculate physical formulas. We assert at the end of each formula that the outcome has the correct units. This is a common practice when doing physics on paper as well, to catch many small omissions or errors during a calculation. AMUSE's unit system helps with this. Any calculation using ScalarQuantities (or VectorQuantities) keeps track of

the units. This makes it easy to validate results by checking whether the calculation outputs the correct unit with an assert statement.

4.3 Implementation

We have implemented the process in two separate phases. First, we *simulate* the cluster, gathering all relevant data. The state of the cluster is saved for every time step in a snapshot. We use Python's pickle functionality for this, which simply dumps the `ParticleSet` object in memory into a file. A more common approach would be to use a format like hdf5, which can store all snapshots in one file. We had some inconsistencies happen with this data format, and so switched to pickle instead. We also store a dictionary of cluster metrics, the command line parameters given to the program, and the constants we use.

The saved data can then be loaded into the *analysis* script. We have separated the process into two phases, because the simulation takes anywhere from an hour to a day to run. If we would not save the data it produces, we would have to re-run the simulation for every little tweak of the analysis algorithm.

The analysis script initially did not take very long to run, under a minute. With the addition of the calculation for the work function, however, the analysis can now take up to two hours, depending on the range of times we want to calculate work for. We now also save the calculated work, and some derived metrics that use the snapshots, so that we can load those in and skip a lot of the analysis time that would simply recalculate the same thing every time. This allows for a fast development loop, which makes it easy to explore the data in different ways.

4.3.1 Simulation

We will now give an overview of the simulation code, everything it does and why, and the command line parameters it can take as input. The code can be found and downloaded at https://github.com/evavh/bachelor-thesis/tree/main/simulation.

The script for the simulation needs to perform the following tasks:

- Generate an initial state.
  We create a Plummer model from a random seed if we start from scratch, otherwise we load a snapshot from a given directory.

- Set up the integrator.
  We create the AMUSE integrator object and wrap it in a class called `Gravity` to easily switch between different integrator codes. We did not end up using this functionality, but it is generally considered good practice to decouple implementation from the calling interface anyway. The simulation parameters are input into the integrator here.

- Start the main simulation loop.
  This loop repeats until we have found a suitable binary.

– Save data.
We save a snapshot of the cluster state and certain `cluster metrics`. These include the binaries we have found with binding energy larger than 10 kT, different radii of the cluster (half-mass radius, core radius), but also data on the simulation such as the current N-body time and how long it took to integrate this step.

– Check whether end time has been reached.
The data is saved at the start of the loop, so we also catch the initial state, and afterwards we check whether another step needs to be integrated. This way we also save the final state.

– Integrate a time step.
AMUSE makes it quite easy to perform the actual N-body integration:

```
while gravity.time() < time:
    gravity.evolve_model(time)

gravity.copy_from_worker()
```

The function `evolve_model` asks a separate worker thread to perform the integration. We then need to copy the data back to Python using an inter-process channel that AMUSE provides, which we wrap in the function `copy_from_worker`.

– Find binaries.
We use Numpy to calculate the binding energy per star for all the other stars in one step. Numpy performs calculations on arrays in C, and is highly optimised. The code shown below finds the star most bound to a certain star, and if its binding energy exceeds the threshold, it is added to the list of binaries.

```
def find_binaries(stars, minimum_Eb):
    G = nbody_system.G
    binding_energies = []
    binaries = []

    for star in stars:
        mu = star.mass*stars.mass/(star.mass+stars.mass)
        dr = (stars.position - star.position).lengths()
        dv = (stars.velocity - star.velocity).lengths()
        Eb = G*star.mass*stars.mass/dr - 0.5*mu*dv*dv

        # find index of second largest binding energy
        # (largest is binding energy to self, which is infinite)
        # .number removes units
        maxEb_index = numpy.argpartition(-Eb.number, 2)[1]
        maxEb = Eb.number[maxEb_index]
        partner = stars[maxEb_index]
        if maxEb > minimum_Eb and star.id < partner.id:
```

```
            binding_energies.append(maxEb)
            binaries.append((star, partner))

    return binaries, binding_energies
```

Once this binary has been found, we integrate another time step and then stop the simulation.

- Stop the worker code.
  Since this is a separate process, we need to explicitly stop it before our script ends.

- Save the final state for the integration test.

The simulation parameters allow variation in the number of stars, random seed, binding energy threshold and time step size (which can also be variable, in which case it will be $0.01t_{crc}$ like Tanikawa et al. use). Input snapshot folder and output for saved data can be configured here. A fixed time to stop the simulation can be set, rather than when a binary forms, and the start time will set the snapshot to be used as initial state.

For the exact command line parameters and their uses, run `main.py` with `--help`.

4.3.2 Analysis

When the simulation is done, it is time to analyse the data. The analysis code can be found here: https://github.com/evavh/bachelor-thesis/tree/main/analysis

The analysis script performs the following tasks:

- Load the simulation data.
  We load the snapshots and saved cluster metrics, but also the command line parameters and constants used in the simulation.

- Find when the binary reaches 10kT (`t_bin_10`).
  If possible, we simply go through the cluster metrics and find the time when the simulation found the binary. If we need to lower this threshold, because the simulation did not find a binary over 10kT, we need to re-calculate the binding energy for every time step.

- Find when it reached a positive binding energy (`t_bin_0`).
  This moment gives an approximation of when the interaction that hardened the binary started. If we look for this moment from the beginning of the data forward, however, we might be too early. The binary can have a positive binding energy for a short time, but become unbound again. Therefore we search for this moment back from the time it reaches the 10kT threshold. This way we find the last moment where the binding energy was below zero instead.

- Find the most important stars.
  To try to find the stars that most influence the binary formation, we need to do the following:

– Calculate the work function.
We calculate the work function (see section 3.2) for the binary with every other star in the cluster. The start and end times are somewhat arbitrary, with a margin around the time between when the binary reaches a positive binding energy and when it reaches 10kT.

```
def power_function(tuple, star):
    dEdt = 0 | nbody_system.energy / nbody_system.time
    for k in tuple:
        v_k = numpy.array([k.vx.number, k.vy.number, k.vz.number]) | k.
            ↪ vx.unit
        v_cm = tuple.center_of_mass_velocity()
        dEdt -= f_ik_j(k, star).dot(v_k - v_cm)

    assert (dEdt.unit == nbody_system.energy / nbody_system.time)

    return dEdt


def work_function(data, tuple_ids, star_id,
                  start_index, end_index):
    snapshots = data.snapshots
    power_functions = []

    for snapshot in snapshots[start_index:end_index]:
        tuple = helpers.ids_to_stars(snapshot, tuple_ids)
        star = helpers.ids_to_stars(snapshot, [star_id])[0]

        power = power_function(tuple, star) * (1.0 | nbody_system.time)
        power_functions.append(power.value_in(nbody_system.energy))

    power_functions = numpy.array(power_functions)

    dt = data.delta_ts()[start_index:end_index]
    work = numpy.cumsum(power_functions*dt)
    total_abs_work = numpy.sum(numpy.abs(power_functions*dt))

    return work, total_abs_work
```

The number `total_abs_work` will be used in the next step to determine which stars did the most work on the binary, whether by pulling it apart or making it more tightly bound.

The performance of this code is surprisingly good. We feared it would be unfeasible to calculate the work function for all stars for a sufficient number of time steps, but the calculation takes an acceptable few hours (depending on the number of snapshots). Integration of the function is Numpy accelerated using `cumsum`. We do save the work

function data, so that the rest of the analysis can be performed without those few hours of recalculating.

  - Find the top contributing stars
    We would like for the script to output an exact number of stars involved. Unfortunately there is no clear cut between involved and uninvolved stars, as every star in the cluster contributes some (albeit tiny) gravitational force to the interaction. We therefore take the top 5 or so stars that have done the most work on the binary.

- Plot graphs
  We use `matplotlib` to make graphs, some of which are similar to what Tanikawa et al. show in their papers. The fast plot command line option skips everything that is not absolutely necessary for the most important plots, and is very useful for tweaking the graphs.

There is some edge case handling in the analysis script, but this is not robust enough to deal with many cases. Some of the analysis can still be done when the cluster metrics are missing, like the plotting of the binding energy of a known binary. If no binary was detected in the simulation, a binary and new binding energy threshold can be manually input on the command line to be able to get some information as to what is going on.

4.4 Running the experiments

First we ran the simulation from a Plummer model with 1000 stars, seeded with `seed 1` until it found a binary with a binding energy of more than 10kT. This run uses $\Delta t = 1.0$ N-body time, a rather large time step. We will call this the initial run. This run gives an approximate time at which the first binary might have formed.

We then ran a more detailed simulation with variable $\Delta t = 0.01 t_{crc}$. We start from the snapshot before the binary reached 10kT, and run until the first snapshot where it had reached 10kT in the initial run. The binary was not detected again in this detailed run.

We decided to let the simulation keep running with these same small time steps until a binary with binding energy over 10kT was found again. This took around 16 hours of integration time.

Then we had a suspected binary formation event to analyse. We calculated the work function for this binary and the stars in the core. Because the results were not as expected (see section 6) we produced a video of the simulation to see what was going on. For this we used the command line tool ffmpeg to convert scatter plots of the simulation into a video.

To make sure our conclusions would be more reliable, we reran this process with two other seeds (creatively named `seed 2` and `seed 3`). For `seed 2`, the detailed run timed out after 23 hours of running time, which is an order of magnitude more than the other detailed runs.

5 Results

Here we present the results of our simulations and their subsequent analysis. There are three seeds, each with their initial and detailed runs.

5.1 Seed 1

This seed received the most attention, as it was the seed we initially worked with. We will show results for an initial run, a detailed, continued run, and the detailed run that we ran, which did not report a binary.

5.1.1 Initial run

The scatter plots in figure 6 show the initial state of the cluster and the state before and after the reported binary formation. We see that the core collapses, and that the binary forms in the core.

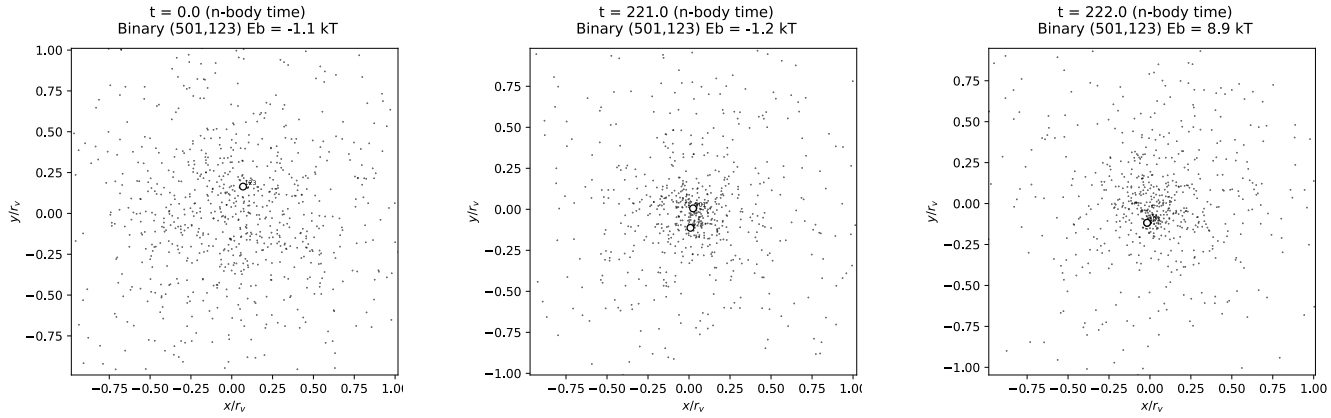


Figure 6: These scatter plots show the cluster for `seed 1` in different stages: at the start of the simulation, right before, and right after the binary forms. They were taken from the *initial run*, and show only two of the three dimensions. The stars that will form the binary reported by the script are marked with large circles, and every dot represents a star. The axes are scaled by the virial radius of the cluster. We see that the core of the cluster has collapsed by the middle snapshot, and that the binary formation happens in the core.

Figure 7 shows the binding energy for both the binary reported by the initial run, as well as the binary that is initially present in the detailed run. It compares this with the time the N-body integration took for each time step, which strongly correlates with the binary formations.

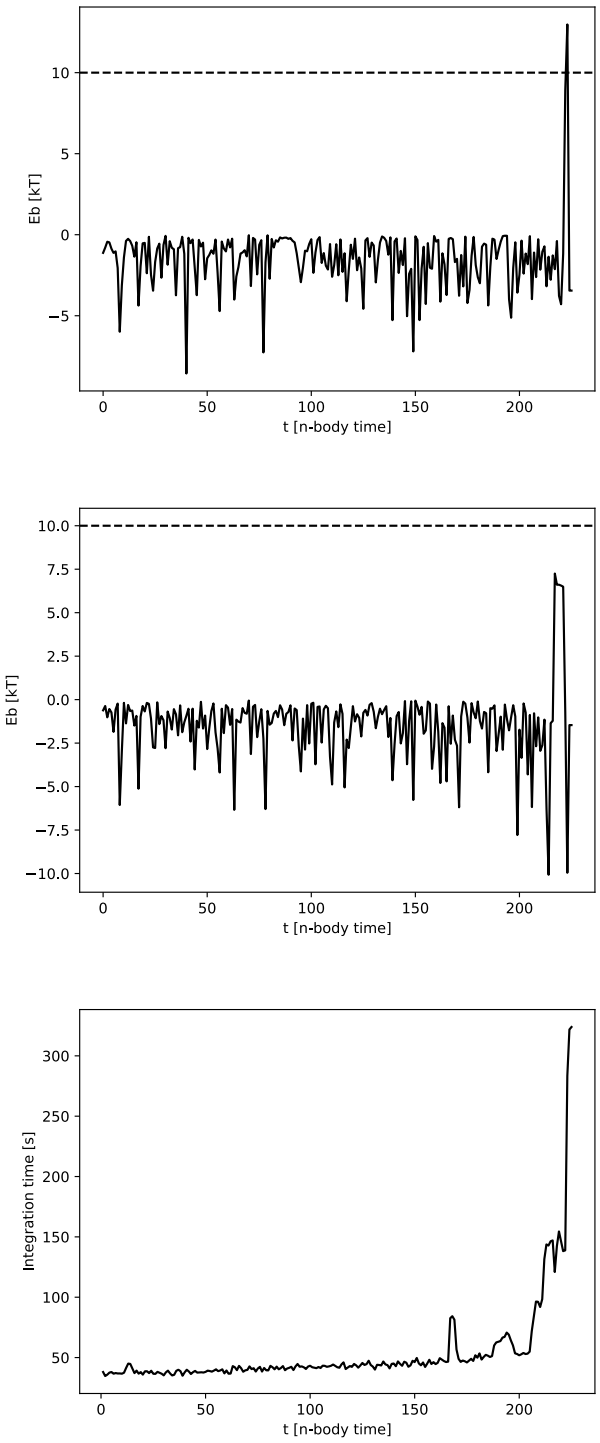


Figure 7: The top graph shows the binding energy of the reported binary in the *initial run* with `seed 1`. The 10kT threshold is marked. The middle graphs shows the same, but for the binary that is initially there in the detailed run. This did not reach the threshold. The bottom graphs shows the time each integration step took in seconds. Each graph is on the same time scale for comparison. The integration time increases explosively around the time that the binding energy of either binary peaks.

5.1.2 Detailed run, continued

The detailed, continued run for `seed 1` uses the same variable time step as the detailed run, but simply runs for longer. It was run until a binary reached 10kT. We then analysed this run, and found the work function graph in figure 8. This graph shows the stars that did the most work on the reported binary. Star 34 jumps out, because it does orders of magnitude more work than the binding energy of the binary itself.

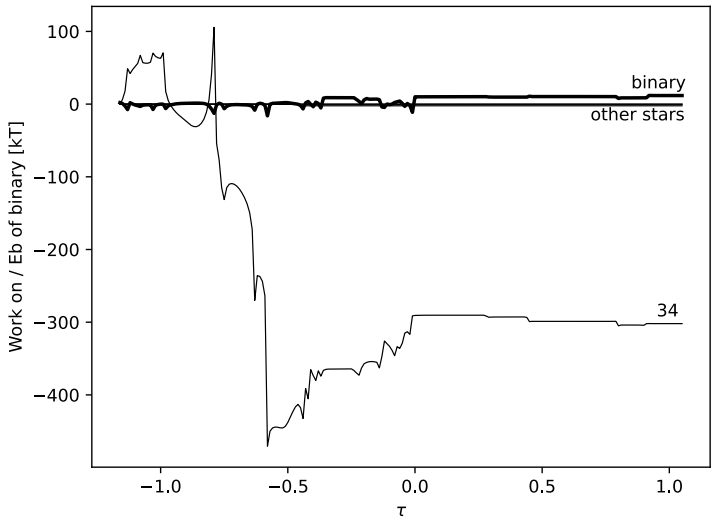
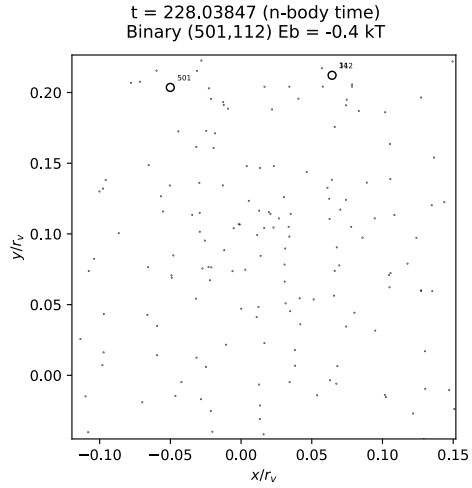
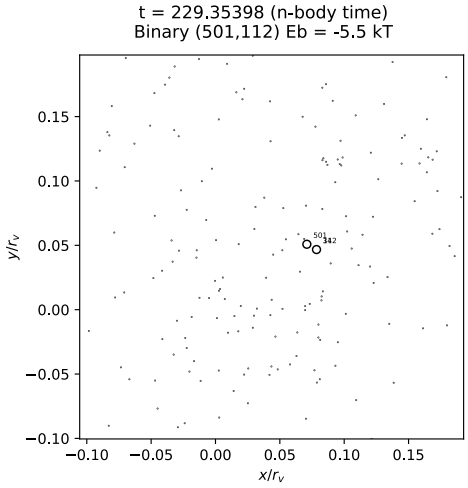


Figure 8: The work function for the stars that did the most work on the binary. It is calculated for the *detailed, continued run* for `seed 1`. The time is measured in τ (see section 2.7.3), with $\tau = 0$ being the binary formation time. Star 34 does orders of magnitude more work than the binding energy of the binary.

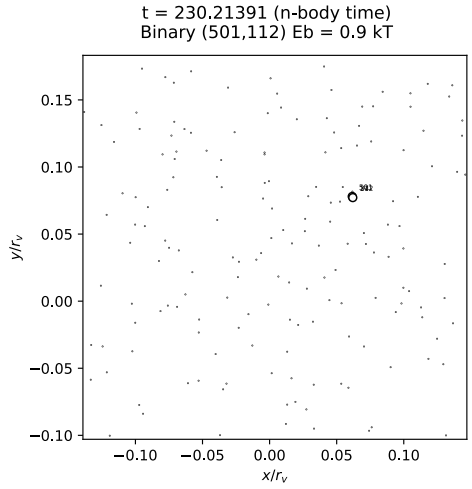
Figure 9 shows an exchange happening between binary (112, 34) and star 501. For a clearer view of this event, see the video at youtu.be/6MnqLimbFN8.



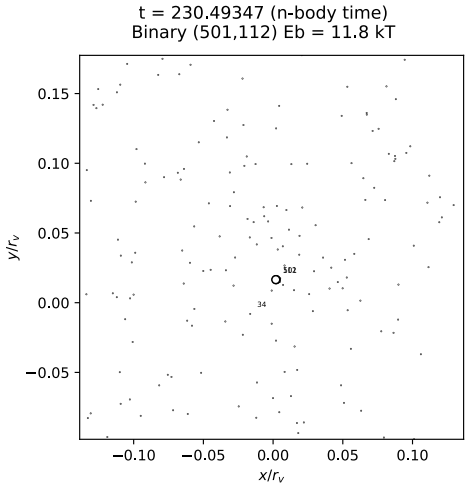
(a) The start of the run, (112, 34) is already a binary.



(b) A loose triple star (112, 501, 34) has formed.



(c) A tighter triple star has formed.



(d) Star 34 has been ejected from the triple.

Figure 9: This series of scatter plots shows an exchange happening in the *detailed, continued run* for `seed 1`. The marked stars are the reported binary (501, 112), which reaches a binding energy of 10kT during the run. In (b), (c) and (d) we see star 501 replace star 34 in its binary with 112.

The binary energy of the two binaries in the exchange is plotted, together with the integration time in the run, in figure 10. We see the integration time increase somewhat after the exchange.

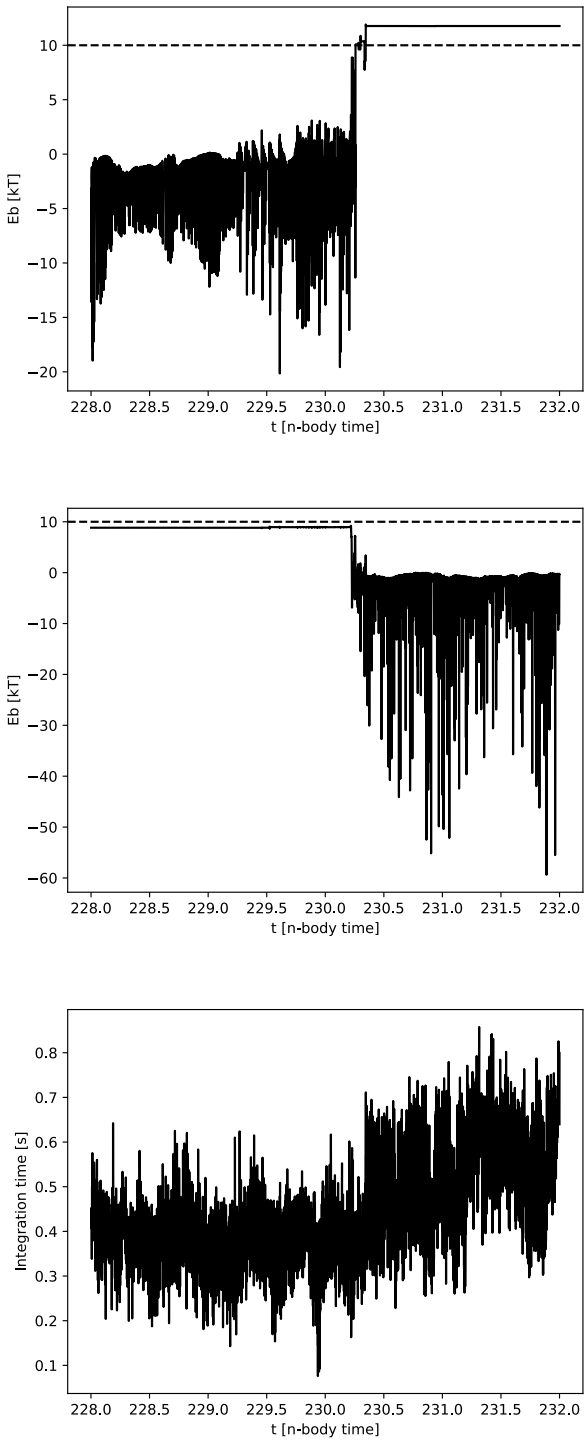
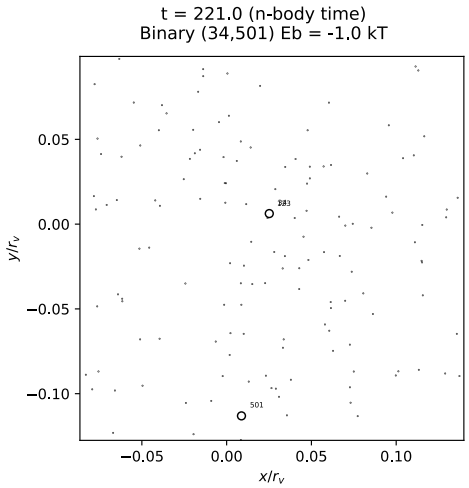


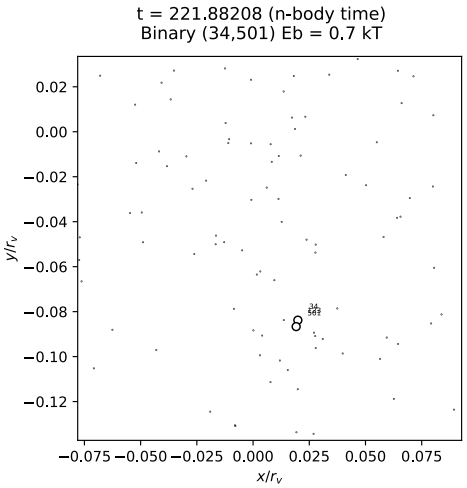
Figure 10: These figures belong to the *detailed, continued run* for `seed 1`. The top graph shows the binding energy of (501, 112), the binary that is formed in the exchange. The middle shows the binding energy of (501, 34), which is dissolved. The integration time, shown below, increases somewhat after the event.

5.1.3 Detailed run

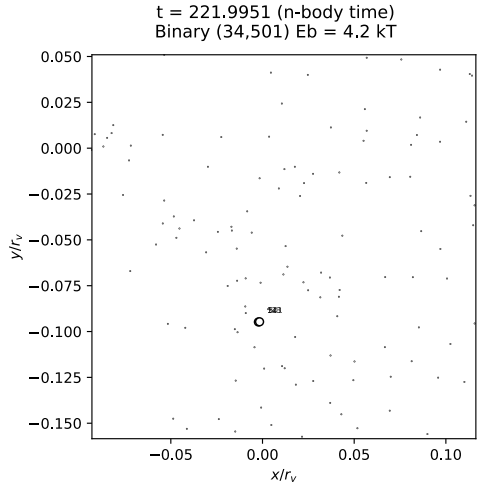
Figure 11 shows the binary that is already present at the start of the detailed, continued run (34, 501) emerging from an exchange between binary (123, 34) and star 501. See the video at youtu.be/CikI5wScOmY for a clearer view of the event.



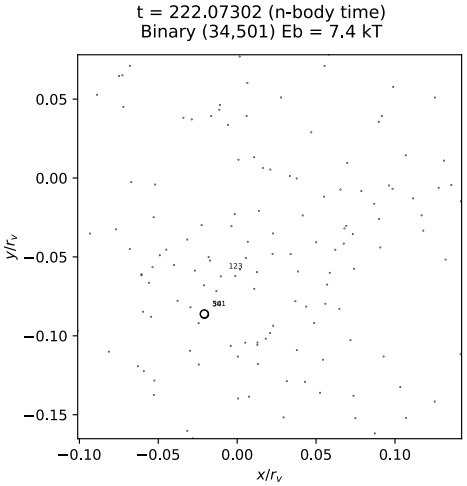
(a) The start of the run, (123, 34) is already a binary.



(b) A loose triple star (123, 34, 501) has formed.



(c) A tighter triple star has formed.



(d) Star 123 has been ejected from the triple.

Figure 11: The exchange in the *detailed run* for `seed 1` happens in much the same way as the later exchange in the detailed, continued run. The marked stars are 501 and 34, which will emerge from the exchange as a binary with binding energy under 10kT.

Figure 12 shows the binding energy for both binaries in the exchange, and the integration time. The integration time correlates with the binary dissolutions, formations, and tightenings, but not as strongly as in other runs.

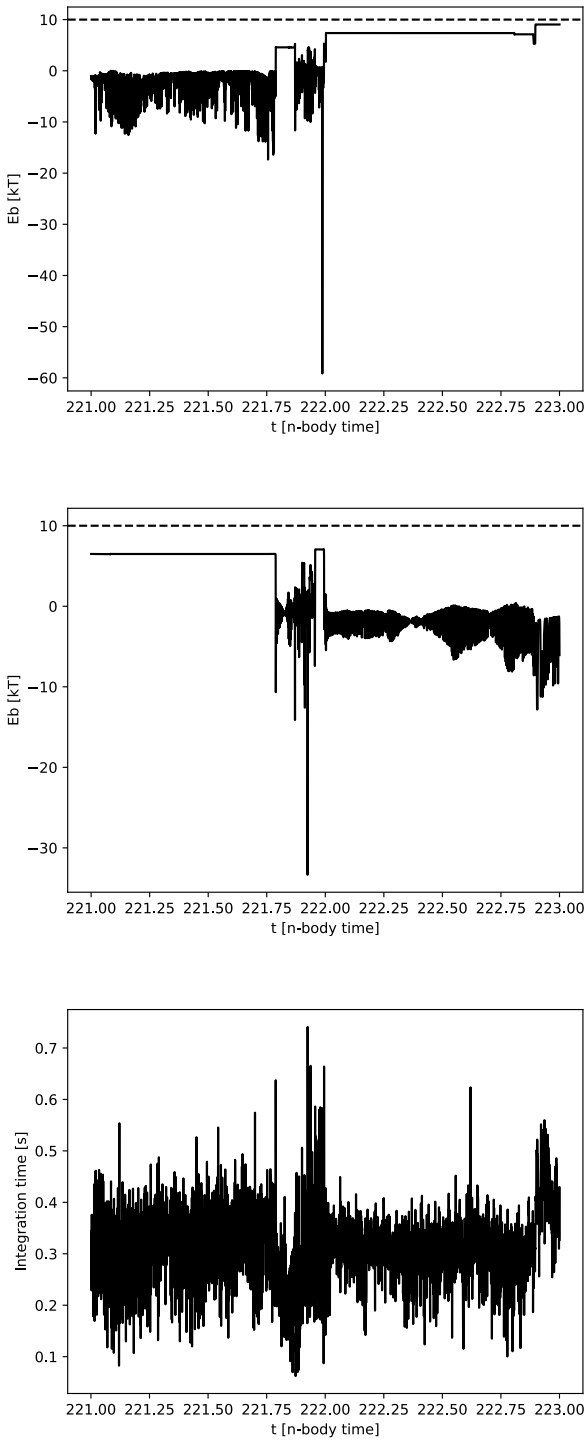


Figure 12: The top graph shows the binding energy for (34, 501) in the *detailed run* for `seed 1`. It does not reach the threshold of 10kT. The middle graph shows the binding energy for the initial binary (34, 123) in the exchange that forms (34, 501). The integration time in the figure at the bottom shows a dip and then peak when the exchange happens, and then a peak when the energy of (34, 501) increases.

5.2 Seed 2

For the second seed we went through a similar process: first run an initial, exploratory run, then a detailed run around the binary formation time.

The initial run reported finding binary (513, 883). In figure 13 we see the binding energy, with a highly correlating integration time.

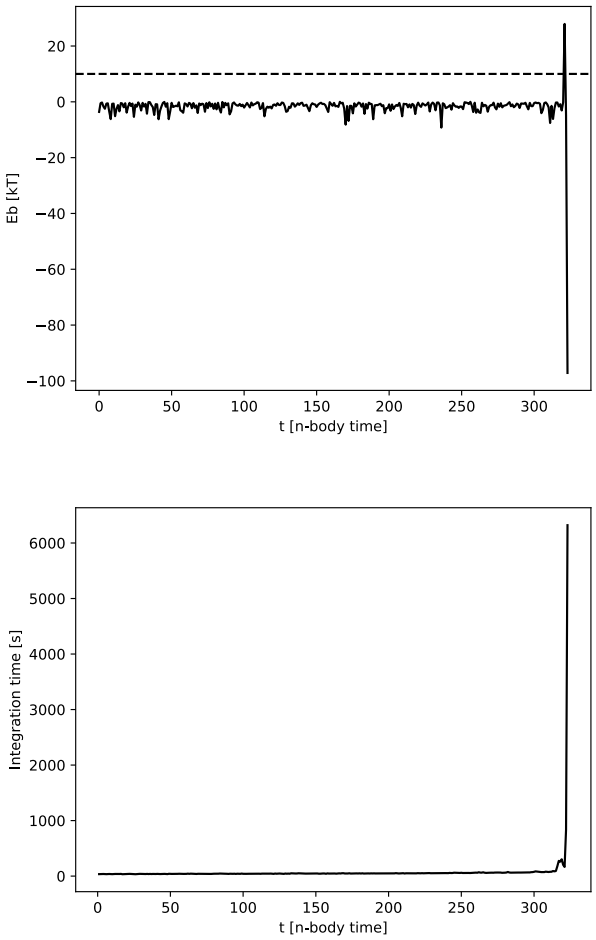


Figure 13: These graphs show the *initial run* for `seed 2`. The top graph contains the binding energy of the reported binary (513, 883), which reached the 10kT threshold. The bottom graph shows the integration time, which strongly correlated with the increase in binding energy for the binary.

Before it timed out, the detailed run for `seed 2` did not find a binary with binding energy over 10kT. In the top graph in figure 14 we see that the binary found in the initial run did not reach a positive binding energy. The integration time did increase dramatically around $t = 321$ N-body time.

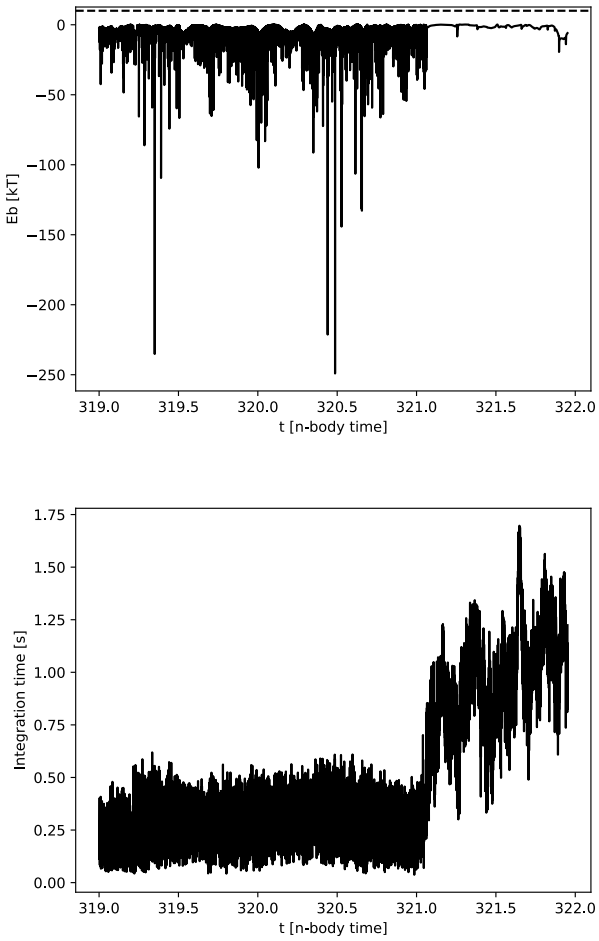


Figure 14: This figure shows data for the *detailed run* for `seed 2`. The top graph shows the binding energy of binary (513, 883), which was found in the initial run. It does not reach a positive binding energy here. The integration time in the graph below does increase dramatically around $t = 321$ N-body time.

5.3 Seed 3

For `seed 3`, we again have an initial and a detailed run. Figure 15 shows a binary (374, 508) reaching 10kT in the initial run, and a strongly correlating integration time.

The detailed run does not find this binary, and the integration time shows random behaviour with no clear peak. See figure 16.

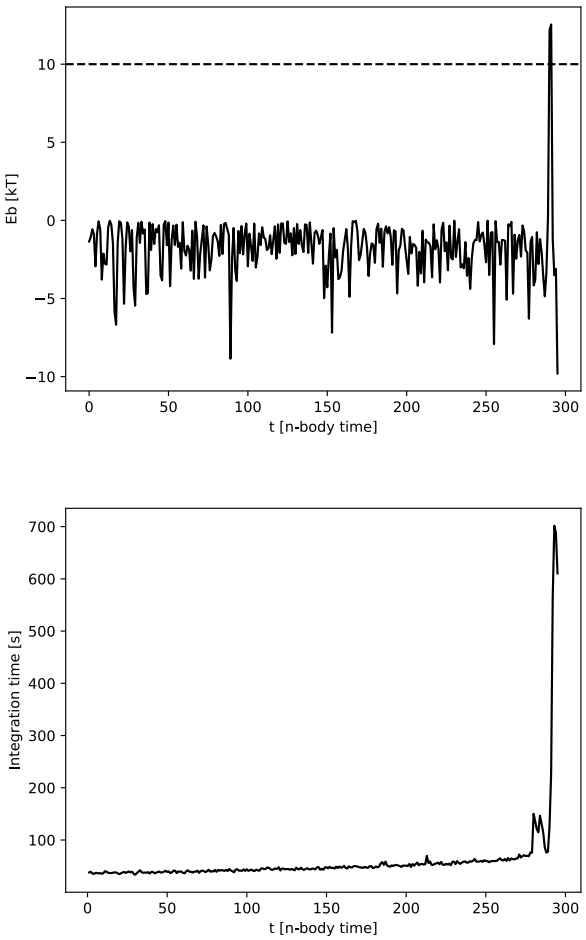


Figure 15: The top graph shows the binding energy for (374, 508), reaching 10kT in the *initial run* for `seed 3`. The bottom graph shows the integration time, which peaks right when the binary forms.

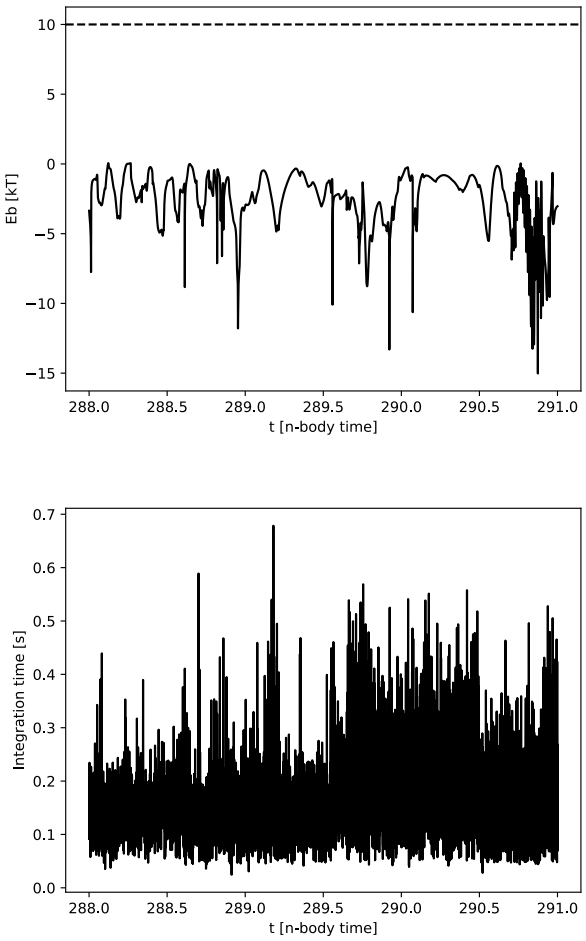


Figure 16: In the *detailed run* for `seed 3`, the binary found in the initial run (374, 508) does not reach positive binding energy. The integration time (bottom graph) does not peak either.

5.4 Run metrics

Table 1 shows the time in $t_{rh,i}$ at which the binary reached 10kT in each initial run. There is considerable variation between the runs.

seed	$t_{bin,10kT}$ $[t_{rh,i}]$
1	14.22
2	20.87
3	18.54

Table 1: The time at which the binary reached 10kT in the initial run for each seed. It is measured in initial half-mass relaxation times.

In table 2 we have gathered the time it took for each of the runs to complete.

seed	initial run	detailed run	continued run
1	3h:43m	1h20m	16h:14m
2	7h:00m (TIMEOUT)	23h:00m (TIMEOUT)	-
3	4h:54m	7h:20m	-

Table 2: The time it took to run each simulation on one CPU thread. Both runs for `seed 2` timed out, although their were both close to their end N-body time.

6 Discussion

Although we did not manage to find and analyse a binary formation event, there are some observations we can make about the runs.

6.1 Chaos and accuracy

One of the major issues we came across was that results were very different between different runs. Every time we decreased the time per integration step Δt, we found that the binary we had found did not form anymore. For the `seed 1` initial run we had to do a rerun because some data had been truncated, and the second run diverged so much from the first that it was unrecognisable.

When using an integrator scheme like Hermite, it is to be expected that slight changes in starting conditions lead to very different outcomes when looking at individual star orbits [PZM18, ch2 p12]. A globular cluster is a chaotic system after all. Our problem lies in the fact that we want to simulate a relatively large number of stars (1000) over a very long time (up to core collapse), while also looking at the interactions between individual stars.

The statistical behaviour of the whole cluster is broadly correct, as we clearly see the core collapse (see figure 6). The results for `seed 2` even suggest that the phase change from cluster without binary to cluster with binary might be preserved. The large increase in integration time in figure 14 is probably due to a binary formation. The integrator increases its resolution when a binary forms, to be able to simulate the tight orbit. Even though binary (513, 883) does not form, some other binary probably forms in this detailed run around the same time it did in the initial run for `seed 2`.

The fact that binary formation behaviours change drastically when changing parameters, however, shows that on the individual star level our results are not accurate enough. This could be due to the error inherent in the integrator. Our energy error of 10^{-8} is rather high for a simulation where we want to study star interactions on a detailed level. Another possibility is that the initial runs' $\Delta t = 1$ N-body time is too large. The difference in Δt with the detailed runs may simply be too great.

6.2 Binary finding

We did not find a binary formation event. Our script detected some binaries reaching over 10kT binding energy, but these were either exchange events, or binaries that did not form anymore in a

more detailed run. Using a binding energy threshold is probably too inaccurate. We missed the formation events of the two binaries that later showed up in the exchanges.

The method we used to detect the exchange events was, much like Tanikawa's, visual inspection. In the middle graph of figure 12 we can see that there is already a binary present when the detailed run starts. This binary has a very stable binding energy around 6kT, and is clearly hard enough not to dissolve (other than through an exchange event, which we do not mind). The binary that comes out of the exchange has a higher binding energy (see the top graph in the same figure), but still does not reach the threshold. It also seems there is no threshold that would work here, as both these binaries exceed a 5kT threshold right before they (temporarily) dissolve again.

6.3 Software engineering issues

During this research, we ran into some practical difficulties when it came to running the experiments.

6.3.1 Integration time explosion

As soon as a binary formed in the simulation, the time it took to integrate every time step increased dramatically. While this gives us a secondary method of detecting whether a binary has formed, it also complicates matters. For `seed 2` our runs kept timing out, because the simulation slowed down so much at the end that it did not manage to get through the last few time steps within a reasonable time. It will be difficult to follow a binary long after formation because of this phenomenon. This makes it harder to find an algorithm that checks for binary hardness.

6.3.2 Truncated runs

Some of our data got corrupted while writing it to disk. This greatly complicated the analysis, as we had to reconstruct some cluster metrics. Since runs timing out may not be an avoidable problem (see 6.3.1), this will be something that needs to be dealt with in further research.

6.3.3 Code duplication

It is customary in astronomy to write software very linearly. Load data, perform some transformations on the data, then generate a graph. We found that this was the most logical way to engineer the code initially, but as we added more control flow to the analysis phase, we realised this script-like structure did not work very well anymore. One way this manifested was in code duplication.

In our code, we calculate binding energies and try to find binaries within the data in several different places. The astronomer's linear programming style does not take into account that some algorithms need to be used in different parts of the program. We split our code up into separate functions, and grouped similar functions in separate modules, but found that it was difficult to untangle the binding energy and binary detection algorithms from their specific context. It became difficult to keep an overview of the code, and dependencies between different parts of the program made some changes take much more time than necessary.

6.3.4 Issues with Python

As user-friendly a programming language as Python is, there are some aspects of the language we do not agree with. Python is a dynamically types language, which means that each variable has a type, but it is always implicit. This makes Python code very flexible, as a variable can easily be implicitly converted to another type. It also result in a very clean, naturally readable code. The great disadvantage of this typing system, combined with the fact that it is an interpreted language, is that correctness of the code is not checked before run time.

A strongly typed and compiled language like C checks syntax and types during the compilation. This means that if you, for example, give a function a parameter of the wrong type, this will be caught before your program starts executing. This prevents a large class of errors and bugs. By far the most common run time error we received, was a `TypeError`, or one related to typing (an invalid operation on a certain type, for example). Not rarely did this happen after the program had been running for several minutes, or had been started as a job on the computing cluster.

AMUSE complicated the matter further, by introducing the `ScalarQuantity` and `VectorQuantity` types. These types wrap primitive Python types with units. This proved very useful in our unit testing (see 4.2.3), as we could simply use an assert statement to check that the unit of the function output was equal to the correct unit. It did also, however, introduce another class of run time type errors, where variables that were AMUSE `Quantities` got mixed up with Python integers and floats.

This confusion may be avoided by being consistent from the start with conversions between `Quantities` and primitives, and function parameters and outputs, but dynamical typing does not help in achieving this consistency. Combining the AMUSE types with strong typing would yield the best of both worlds.

6.4 Work function

Calculating the work function went much better than expected. We feared computing the integral over many small time steps for many stars might take too much time, but it turned out to be very feasible to calculate the work function for all the stars in the core.

6.4.1 Studying exchange events

The work function graph in figure 8 was what alerted us that we might be dealing with an exchange event. Though the actual value of work done by the third star in the exchange (star 34) is not very useful, someone trying to find exchanges in a simulation may well be able to make use of this formula. Of course, a simpler metric such as distance between the stars may suffice.

6.4.2 Studying formation events

As Tanikawa et al. showed in their second paper, the work function can be a very useful tool in studying dynamical interactions between particles. When automatically calculating the function for a formation event, it may prove difficult to get the start and end time right. This took us some trial and error. Deciding how to pick the top contributing stars to an interaction is not trivial either, as

we calculate a function and not a grand work total. We decided to count negative work done by a star as much as positive work. A star trying to pull a binary apart while it forms is, we think, just as interesting as one that pulls a binary together.

6.5 Comparing with Tanikawa

As we attempted to replicate Tanikawa et al.'s papers, in this section we will compare what results we managed to get with their results.

6.5.1 Binary formation time

Tanikawa et al. found that the first binary formed very consistently between 18 and 21 $t_{rh,i}$ [THM12]. Our results varied considerably. As we discussed in section 6.1, they are likely to not be very reliable. In table 1 we see that our binary formation times varied from around 14 to 21 $t_{rh,i}$. According to the literature, values of $15t_{rh,i}$ to core collapse have been observed in simulations [HH03, p187]. It is difficult to draw conclusions based on only three runs, and ones with fragile results at that.

6.5.2 Work function plots

In their second paper, Tanikawa et al. show several plots of the work function in different situations [THHM13]. Many of these show the work done by several hand-picked stars on a hand-picked tuple of up to five stars. They do not show a work function plot for a binary formation event. Our figure 8 of course shows an exchange event, and therefore it is difficult to compare it to Tanikawa's figures. We do not see the work values for the stars adding up to the binding energy of the binary, as most of their graphs show.

7 Conclusion and future work

Although our exploration of automating the analysis of binary formation events did not result in an automated method, there are many things we learned in the process. We have a good idea of where some of the difficult points in the automation process are. Here we present some possible solutions to the issues we came across, and some other ideas that might be helpful to future research. Note that the structure of this section mostly mirrors the discussion in section 6.

7.1 Chaos and accuracy

We need to deal with the chaos of the 1000-body (or more) system that we want to study. Runs for the same seed need to be reproducible and consistent, even when changing certain parameters. The most obvious method of addressing this will be to increase the accuracy of the simulation. We propose the following options to be explored, roughly in order of promise and feasibility:

- Run the initial, exploratory simulation with smaller Δt.
  We chose $\Delta t = 1.0$ N-body time rather arbitrarily. Decreasing this to 0.1 N-body time will increase the running time on a single core tenfold, changing a 3-5 hour simulation (see table 2) to a 30-50 hour one, but this is still feasible and may prevent many hours of headaches. Using

multiple CPU cores can help as well. We do not recommend switching to GPU computation, since this will only reduce replicability of results, and may not even speed up the simulation.

- Decrease the accuracy parameter for the Hermite integrator.
  We used the same values that Tanikawa et al. use, but since the accuracy of the simulation also depends on the frequency of snapshots in the initial run (which they did not specify), some more tweaking may be necessary.

- Examine the precision of pickled snapshots.
  We assumed that using Python's pickle package for snapshots meant we would achieve the same float precision for the snapshots as the script achieves in memory. Pickle dumps the Python object in memory to disk. Perhaps using a format such as hdf5, which is widely used in the astrophysical community, will be beneficial.

- Switch to a different integrator.
  The Hermite integrator scheme is very fast, but this comes at the cost of accuracy. The error in the simulation builds up over time, and in a simulation that covers a lot of real-life time (billions of years, in fact) this adds up. It may therefore be worthwhile to try out some other integrators, to see if they integrate fast enough while yielding a smaller error. A rather extreme example of a slower, but more accurate integrator is Brutus [BPZ14]. Brutus is available as a community package in AMUSE. It can be adjusted to be so accurate that a simulation is perfectly reversible in time, working back to its initial conditions. Unfortunately it is also very slow, having only been applied to small N-body problems. Using it to simulate a cluster of 1000 stars all the way to core collapse is simply not possible. It may be useful, however, to simulate just the moments around the binary formation.

If one successfully deals with the accuracy issue, the energy error of the simulation will decrease. Decreasing the time between snapshots will yield quantitatively similar results to the initial simulation, meaning the same binary forms, although individual orbits will probably still be different. Unless one can apply Brutus to the problem, this will be unavoidable.

7.2 Binary finding

We clearly need a more sophisticated algorithm to detect binary formations. Using a threshold for the binding energy made us miss the initial binary formation event in the runs for `seed 1`. We have some ideas that may be useful in the formulation of this algorithm:

- Exclude binaries with binding energy below $\frac{3}{2}kT$.
  Although the 10kT threshold proved too strict to be used in an automated script, a lower bound on the binding energy can exclude many stars from further analysis. The binding energy of a binary must be at least $\frac{3}{2}kT$, the average kinetic energy of the stars in the cluster, to keep it from dissolving very quickly. Any truly hard binary will cross this threshold during its formation, or very soon after.

- Look for near-constant binding energy.
  In our visual inspection of the simulation, we made extensive use of binding energy graphs like those in figures 10 and 12. These graphs show a binary forming through an exchange

event. This may not be fully representative of an initial binary formation, but they do show the stable binding energy that we expect from a hard binary. Simply calculating further into the future after a binary has formed shows whether it survives or not. This will probably be complicated by the integration time explosion issue detailed in section 6.3.1. Once the binary has formed, the simulation requires an order of magnitude more time per time step. Throwing more hardware at the problem may help.

- Use the integration time per step as a signal.
  Perhaps the integration time explosion can also be used to our advantage. It is worthwhile to examine whether this rapid, large increase in the time it takes to calculate each time step indeed correlates well with the binary formation we are looking for. Detecting it may be a challenge, as we see in for example figure 12. The graph for the integration time is very noisy. The detailed runs also use a variable time between snapshots, which needs to be taken into account.

- Compare the semi major axis of the binary to the distance between stars in the region.
  There is another metric to determine the likelihood that a binary will dissolve: its semi major axis, the maximum distance between the two stars in their elliptical orbit [Aar71]. Comparing this distance to the distance between the surrounding stars is an intuitive measure of the tightness of the binary. It is very similar to what we naturally do when we look at a video like the ones presented in sections 5.1.2 and 5.1.3. The paper by Aarseth that we cited gives several formulas that may be of use.

- Train a neural network to perform a "visual inspection".
  If we manage to find a reliable binary formation detection algorithm, but it turns out to be too expensive to compute, we could apply a neural network. The neural network would find possible formation events, and the expensive metric would then be used as a metrics for reinforcement learning. The network is punished if the event is not actually a binary formation, and rewarded if it is. The question here of course is whether the network would be able to achieve a good enough performance with lower computational costs. We also need to find an automated reinforcement metric first, otherwise training the network will be impossible.

A successful, automatic binary finding algorithm will be reliable enough at detecting formation events that we can run many simulations, and statistically aggregate the results. This means that the false negative rate would not necessarily need to be negligible, as long as it does not exclude a whole class of binary formations that would skew the results. The false positive rate would need to be very low, maybe near zero, since events like exchanges can have large distortive effects on global metrics like the average work done per star. Ideally, therefore, the algorithm would take metrics into account that relate directly to the definition of a hard binary formation.

7.3 Software engineering

When building software of any kind, there are many practical issues that can come up. Some of our data got corrupted when a simulation run timed out while it was writing to disk. Since writing to disk takes up much of the time that a simulation runs, especially when taking many snapshots, it is more likely than not that a simulation process is killed while writing data. We want to advise

anyone working with computational runs that take a long time to complete, like these simulations, to use multiple files to write data to. Do not overwrite the same file over and over, as one write error can corrupt all previous data. It is better to still have all the data up to the previous integration step, rather than losing everything.

Keeping an overview of the code proved difficult. We ended up calculating binding energies in multiple places, which did not help performance of especially the analysis script. Future code will need common functions between the simulation and analysis phases. Perhaps the simulation could, as it runs, create a data structure that contains a catalogue of potentially interesting binaries, with their binding energies over time. This can then easily be analysed, without doing much work twice.

As for the Python-specific issues we encountered due to dynamic typing, modern Python versions offer a solution: type hinting [Fou22]. Variables and function definitions can be annotated with types. While the Python interpreter does not take these type hints into account, other tooling such as an IDE can take advantage of them. Many of the run time errors can be caught this way, before running the code. We will certainly be using type hinting for our future projects.

7.4 Automating further

Assuming that we will, at some point in the future, have replicable runs and a reliable binary formation finding algorithm, what would still need to be done to achieve the goal of a fully automated analysis?

We calculated the work function from $t_{bin,0} - 0.1$ to $t_{bin,10} + 0.1$ where these two times are respectively the time that the binding energy reaches above zero, and the time it reaches 10kT. This seemed to be a reasonable margin that captured the interaction well enough. Depending on how far back one wishes to study the interactions that lead to the binary, this window may need to be extended. If the window becomes much larger, however, we will need to take into account more stars than just the ones in the core at $t_{bin,0}$. This will of course increase the time it takes to calculate the work functions.

Eventually we would like to have an automated script that takes a random seed as input and outputs a report on the binary formation event. This would make it possible to study many binary formations and come to statistically sound conclusions. The script could of course be modular, being split into a simulation, an analysis, and perhaps a visualisation phase, but these phases would have to autonomously start the next phase when one finishes, and pass on the necessary data. The whole script might perform the following tasks in order:

- Simulate the cluster from a given random seed.
  This part of the program would perform the binary detection algorithm, and stop when it determines that a hard binary has formed. It then passes on the snapshots between which the event happened, and the binary in question.

- Rerun the simulation with smaller intervals between snapshots.
  Similar to our detailed runs, the script would zoom in on the time around the binary formation, and confirm and pinpoint the formation event.

- Calculate the work function for core stars around the formation time.
- Find the stars that were most involved in the formation.
  It may turn out to be obvious which stars contribute significantly to the binary formation, but it may also prove to be difficult to set a work threshold.
- Report on the findings with metrics and visualisations.
  Maybe the script will output a single number: *it took this many stars to form a binary, this time*. More likely the results will be more complex: this many stars were involved, in these ways, while forming these n-tuples. Visualisations will also be useful as output, for humans to study the formations in more detail, and to validate the program's results. It would be very useful to automatically output a diagram like figure 5.

Solving simulation accuracy and binary finding, and producing visualisations like figure 5 automatically, will probably be complete research projects in their own right. We have come across some of the difficulties in automating binary formation analysis, but we do not doubt many surprising and interesting challenges still await those who continue this research.

References

[Aar71] S. J. Aarseth. Binary evolution in stellar systems. *Astrophysics and Space Science*, 13(2):324–334, Oct 1971.

[BPZ14] Tjarda Boekholt and Simon Portegies Zwart. On the reliability of n-body simulations. *Computational Astrophysics and Cosmology*, 2, 11 2014.

[BT08] James Binney and Scott Tremaine. *Galactic Dynamics: Second Edition*. 2008.

[Fou22] Python Software Foundation. Python docs: typing — support for type hints, 2022.

[GH94] M. Giersz and D. C. Heggie. Statistics of N-Body Simulations - Part One - Equal Masses Before Core Collapse. *MNRAS*, 268:257, May 1994.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991.

[HH03] Douglas Heggie and Piet Hut. *The Gravitational Million–Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics*. Cambridge University Press, 2003.

[LS78] Alan P. Lightman and Stuart L. Shapiro. The dynamical evolution of globular clusters. *Rev. Mod. Phys.*, 50:437–481, Apr 1978.

[Pvd+13] Pelupessy, F. I., van Elteren, A., de Vries, N., McMillan, S. L. W., Drost, N., and Portegies Zwart, S. F. The astrophysical multipurpose software environment. *A&A*, 557:A84, 2013.

[PZM18] Simon Portegies Zwart and Steve McMillan. *Astrophysical Recipes*. 2514-3433. IOP Publishing, 2018.

[PZMH+09] Simon Portegies Zwart, Steve McMillan, Stefan Harfst, Derek Groen, Michiko Fujii, Breanndán Ó Nualláin, Evert Glebbeek, Douglas Heggie, James Lombardi, Piet Hut, and et al. A multiphysics and multiscale software environment for modeling astrophysical systems. *New Astronomy*, 14(4):369–378, May 2009.

[PZMvE+13] Simon F. Portegies Zwart, Stephen L.W. McMillan, Arjen van Elteren, F. Inti Pelupessy, and Nathan de Vries. Multi-physics simulations using a hierarchical interchangeable software interface. *Computer Physics Communications*, 184(3):456–468, Mar 2013.

[PZvEP+21] Simon Portegies Zwart, Arjen van Elteren, Inti Pelupessy, Steve McMillan, Steven Rieder, Nathan de Vries, Marcell Marosvolgyi, Alfred Whitehead, Joshua Wall, Niels Drost, Lucie Jilkova, Carmen Martinez Barbosa, Edwin van der Helm, Jeroen Beedorf, Patrick Bos, Tjarda Boekholt, Ben van Werkhoven, Thomas Wijnen, Adrian Hamers, Daniel Caputo, Guilherme Ferrari, Silvia Toonen, Evghenii Gaburov, Jan-Pieter Paardekooper, Jurgen Janes, Davide Punzo, Chael Kruip, and Gabriel Altay. AMUSE: the Astrophysical Multipurpose Software Environment, July 2021.

[Spi87] Lyman Spitzer. *Dynamical evolution of globular clusters*. 1987.

[THHM13] Ataru Tanikawa, Douglas C. Heggie, Piet Hut, and Junichiro Makino. Few-body modes of binary formation in core collapse. *Astronomy and Computing*, 3-4:35–49, 2013.

[THM12] Ataru Tanikawa, Piet Hut, and Junichiro Makino. Unexpected formation modes of the first hard binary in core collapse. *New Astronomy*, 17(3):272–280, Apr 2012.