# Chapter 3

# Gravitational Dynamics

> Oh my God! — its full of stars!
>
> Arthur C. Clarke

Here, we discuss how to solve relatively simple problems in Newtonian gravity using the monophysics solvers in AMUSE. Aside from the specific applications, this chapter gives an overview of how to perform simple experiments, what to be aware of, and how to address technical and algorithmic issues. This chapter should therefore be read as a general overview of the working of AMUSE with gravity as a focus. Even if you are not interested in gravity—for example, if your research is dominated by problems in radiative transfer or hydrodynamics—this chapter should still be your starting point.

## 3.1 In a Nutshell

### 3.1.1 Equations of Motion for a Self-gravitating System

In the Philosophiæ Naturalis Principia Mathematica, Newton (1687) described his famous laws of motion in his chapter *Axiomata Sive Leges Motus* (the Axioms or Laws of Motion). He also laid the foundation for the general law of gravitation. In Newtonian gravity, the gravitational acceleration $\mathbf{a}_i$ of an object (subscript $i$) due to a group of objects (subscript $j$) is

$$\mathbf{a}_i \equiv \ddot{\mathbf{r}}_i = G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3}. \tag{3.1}$$

Here, $G$ is the gravitational constant, and $m_j$ and $\mathbf{r}_j$ are the mass and position of particle $j$. Equation (3.1) is a set of coupled, second-order, nonlinear, singular ordinary differential equations. Its solutions exhibit a bewildering variety of complex, often chaotic motions, and many books have been written about how to numerically solve it (Aarseth, 1985, 2003; Heggie & Hut, 2003). Furthermore, because gravity is a long-range force, this behavior can be seen on virtually all scales, from planets and moons to clusters of galaxies. Newton's equations of motion break down only for very large scales or very strong gravitational fields. They are however applicable on most scales considered in this book.

### 3.1.2 Gravitational time scales

Time scales are critical for understanding fundamental processes in (astro)physics. Therefore, we start each chapter on a new physical domain in AMUSE with a brief overview of the relevant time scales. In the case of self-gravitating systems, these are the *dynamical time scale*, the *relaxation time scale*, the *dynamical friction time scale*, and (in extreme circumstances) the *gravitational radiation time scale*.

#### 3.1.2.1 The Dynamical Time Scale

The dynamical time scale, or crossing time, is the time taken for a typical particle to cross the system,

$$t_{\rm dyn} = R/v, \tag{3.2}$$

where $R$ is a characteristic dimension and $v$ is a typical particle speed. In practice, $R$ is often taken to be the mean radius of the system (defined more carefully below) and $v$ is the root mean square velocity $\langle v^2 \rangle^{1/2}$ of all particles relative to the center of mass.

For a two-body gravitating system, the dynamical time scale is simply the orbital time scale. More generally, for a system in virial equilibrium (no net flux of particles across any radius), the virial theorem relates the (time-averaged) kinetic energy $T$ and the potential energy $U$ of the system by (Binney & Tremaine, 2008)

$$2T + U = 0. \tag{3.3}$$

For a system of mass $M$, we can write $T = 1/2M\langle v^2 \rangle$ and $U = -GM^2/2R_{\rm vir}$, where the latter expression defines the virial radius $R_{\rm vir}$. In that case, the virial theorem implies

$$\langle v^2 \rangle = \frac{GM}{2R_{\rm vir}}. \tag{3.4}$$

Taking $R = R_{\rm vir}$ (and neglecting constants of order unity), we can then write the dynamical time scale as

$$t_{\rm dyn} = \sqrt{\frac{R_{\rm vir}^3}{GM}}, \tag{3.5}$$

which we recognize as a generalization of Kepler's third law.

In defining the dynamical time scale, the virial radius is often replaced with the easier-to-measure half-mass radius, $R_{\rm h}$, which is the radius of the sphere enclosing half of the total system mass. Although they are commonly used interchangeably, these two radii are distinct physical quantities. However, Spitzer (1987) notes that $R_{\rm vir} \simeq 0.8 R_{\rm h}$ for a broad range of common dynamical models.

For an open star cluster with $R = 1$ pc and $M = 10^3$ $\rm M_\odot$, or a globular cluster with $R = 10$ pc and $M = 10^6$ $\rm M_\odot$, we find $t_{\rm dyn} \simeq 0.47$Myr. For our Galaxy, with $R =$ 10 kpc and $M = 10^{11}$ $\rm M_\odot$, $t_{\rm dyn}$ $\simeq 47$ Myr. Within AMUSE, we can easily perform

such a calculation as follows:

```
import numpy as np
from amuse.units import constants

def dynamical_time_scale(mass, radius, G=constants.G):
    return np.sqrt(radius**3 / (G * mass))
```

Recall that the first line ensures that you have access to all AMUSE functionality. Having imported the needed libraries and defined this function, you might try the following:

```
from amuse.units import units
mass = 10**11 | units.MSun
radius = 10 | units.kpc
time_dyn = dynamical_time_scale(mass, radius)
print(f'dynamical time scale={time_dyn}')
```

The first two lines initialize the parameters $M$ and $R$. The function is then called to calculate the dynamical time scale, and the result is printed. You'll probably find that the time is printed in a rather strange set of units. This reflects the way in which AMUSE carries out its dimensional calculations, operating on quantities and their units separately until the result is needed. It is usually best to convert into a definite set of units with

```
print(f'dynamical time scale={time_dyn.in_(units.Myr)}')
```

You will see many more examples like the above listing in this book. We encourage you to copy and paste these code snippets into a Python terminal for execution and experimentation, but note that leading spaces as well as single quotes may not be copied correctly from the PDF version of this book into your Python environment. Alternatively to cut and paste from the book, you can find all the source code in the AMUSE repository in the module: `amuse.examples` (see also Section 1.3.6)

#### 3.1.2.2 The Relaxation Time Scale

The dynamical time scale is the scale associated with orbital motion in a gravitating system. It is also the time scale on which virial equilibrium and stable stellar orbits are established (see the assignment on virial equilibrium in Section 3.7.5). The time scale on which the global characteristics (bulk system parameters and stellar orbital elements) of the system change is the relaxation time scale, $t_{\rm r}$. In stellar systems, relaxation is driven by two-body encounters among stars, which change stellar orbital parameters and provide a means for energy and angular momentum to diffuse around the system, causing long-term structural evolution.

The two-body relaxation time is the time scale on which a "typical" particle has its orbit significantly changed by (mostly distant) encounters with other particles. The relaxation time for a system of identical particles of mass $m$, number density $n$, and (three-dimensional) velocity dispersion $\langle v^2 \rangle$ is (Spitzer, 1987)

$$t_{\rm r} \simeq \frac{0.065\langle v^2\rangle^{3/2}}{nm^2G^2\ln\Lambda}, \tag{3.6}$$

where $\Lambda$ is the ratio of the size of the system to the "strong encounter distance" $p_0 = Gm/\langle v^2\rangle^{1/2}$, the impact parameter that would result in a 90 ° deflection for two typical cluster stars. For an idealized system of $N$ identical objects in virial equilibrium, Spitzer (1987) finds $\Lambda = \gamma N$, with $\gamma = 0.4$. For more realistic systems, $\gamma$ is probably considerably smaller: the value $\gamma = 0.11$ is in common use (Giersz & Heggie, 1994; Heggie & Hut, 2003).

A convenient global measure of relaxation in a system is the half-mass relaxation time, $t_{\rm rh}$, obtained by replacing all terms in Equation (3.6) by their global averages and assuming virial equilibrium:

$$t_{\rm rh} \simeq 0.138\frac{N}{\ln\gamma N}t_{\rm dyn}. \tag{3.7}$$

In AMUSE, we could expand the earlier code fragment from Section 3.1.2.1 to

```
import numpy as np
from amuse.units import constants, units

def dynamical_time_scale(mass, radius, G=constants.G):
    return np.sqrt(radius**3 / (G * mass))

def relaxation_time_scale(number_of_particles, mass, radius, G=constants.G):
    return (
    0.138 * number_of_particles / np.log(0.4 * number_of_particles)
    * dynamical_time_scale(mass, radius, G)
    )

number_of_particles = 500000
mass = 10**11 | units.MSun
radius = 10 | units.kpc
time_relax = relaxation_time_scale(number_of_particles, mass, radius)
print(f'relaxation time scale={time_relax.in_(units.Gyr)}')
```

For the Milky Way Galaxy, with number$_{\rm o}$f$_{\rm p}$articles $\sim 10^{11}$ and $t_{\rm dyn}$ as derived above, this gives $t_{rh} \simeq 2.7 \times 10^7$ Gyr, which indicates that relaxation is unimportant. In the jargon of the field, such systems are termed *collisionless*. However, for an open star cluster, with $N \sim 10^3$, $t_{\rm rh} \simeq 11$ Myr and we can expect significant dynamical evolution in much less than the age of the Galaxy or the lifetime of the cluster. This is an example of a *collisional* dynamical system (note that the term has nothing to do with physical stellar collisions, which we will discuss in subsequent chapters).

#### 3.1.2.3 The Dynamical Friction Time Scale

Chandrasekhar (1943) showed that the gravity of a massive body moving through a uniform background distribution of low-mass objects creates a "wake"—an overdense region—behind itself (see also Fellhauer & Lin (2007)). The gravitational pull of this wake always acts oppositely to the massive body's motion, leading to dynamical friction

that tends to slow it down. If the low-mass background has density $\rho$ and is (reasonably) characterized by an isotropic velocity distribution $\varphi(v)$ (where $\int 4\pi v^2 \phi(v)\, dv = 1$), then the acceleration of the massive body due to dynamical friction against the background is

$$\mathbf{a}_{\rm df} = -\frac{16\pi^2 G^2 M \rho \ln \Lambda \mathbf{V}}{V^3} \int_0^V v^2 \phi(v)\, dv, \tag{3.8}$$

where $M$ and $\mathbf{V}$ are the mass and velocity, respectively, of the massive body, and $\Lambda$ is defined as before—except that we replace $p_0$ with the radius $R$ of the massive body if $R > p_0$. Note that, unlike relaxation, this expression does not depend explicitly on either the number density or the mass of the low-mass background, just on the total density.

If (again reasonably) the velocity distribution of the background is thermal,[1] such that

$$\phi(v) = \frac{1}{(2\pi\sigma)^{3/2}}\, e^{-v^2/\sigma^2}, \tag{3.9}$$

where $\sigma$ is the one-dimensional root mean square velocity of the low-mass particles, this expression becomes (assuming $M \gg m$)

$$\mathbf{a}_{\rm df} = -\frac{4\pi^2 G^2 M \rho \ln \Lambda \mathbf{V}}{V^3} \left[ \mathrm{erf}(x) - \frac{2x}{\sqrt{\pi}} e^{-x^2} \right], \tag{3.10}$$

where $x = v/\sqrt{2}\sigma$ (Binney & Tremaine, 2008). The term in square brackets is zero for $x = 0$, scales as $x^3$ for small $x$, and tends to 1 for large $x$, effectively equaling 1 for $x > 3$.

The dynamical friction time scale is

$$t_{\rm df} = \frac{V}{a_{\rm df}} = \frac{0.059\, V^3}{G^2 M \rho \ln \Lambda}, \tag{3.11}$$

where we have conventionally evaluated the bracketed expression in Equation (3.11) for $x = 1$. Combining Equations (3.6) and (3.11) (and neglecting differences in the definition of $\Lambda$), we find

$$t_{\rm df} = 0.91 \left( \frac{V^2}{\langle v^2 \rangle} \right)^{3/2} \left( \frac{m}{M} \right) t_{\rm r}. \tag{3.12}$$

Thus, so long as $V^2$ is comparable to $\langle v^2 \rangle$, a massive body will sink toward the center of a galaxy or cluster on a time scale much shorter than the relaxation time of the background particles.

[1] More precisely, this generally means that the direction is isotropic and the speed is drawn from a Maxwell–Boltzmann velocity distribution associated with some temperature.

#### 3.1.2.4 The Gravitational Radiation Time Scale

Relativistic effects, caused (for example) by two nearly point masses on a close orbit, have a characteristic time scale that is usually much longer than any of the other dynamical time scales. We therefore tend to ignore them in $N$-body simulations, but it is still good to be aware of their magnitudes.[2] The time scale for orbital precession (often referred to as apsidal precession) is shortest, but it has limited effect on the global dynamical evolution of a stellar system. The relativistic apsidal precession (in radians per orbital period) for a two-body orbit of semimajor axis $a$, eccentricity $e$, and period $P$, is

$$\epsilon = 24\pi^3 \frac{a^2}{P^2 c^2 (1-e^2)} \tag{3.13}$$

Here, $c$ is the speed of light. For the planet Mercury, this turns out to be about $5 \times 10^{-7}$ radians per orbital period of $P \simeq 88$ days. It takes about a century for Mercury to make an extra loop around the Sun due to this effect.

The time scale for inspiral due to the emission of gravitational radiation is considerably longer, but more relevant for cluster dynamics. We can estimate this time scale using a post-Newtonian expansion of the orbital dynamics (Peters, 1964). Ignoring several complications, we arrive at

$$t_{\rm gwr} \simeq 0.02 \, \frac{c^5 a^4 (1-e^2)^{7/2}}{G^3 M m (M+m)}, \tag{3.14}$$

where $M$ and $m$ are the masses of the orbiting bodies and $a$ and $e$ are the Newtonian orbital semimajor axis and eccentricity. Adopting representative values ($M \simeq 1.44$ $M_\odot m \simeq 1.39$ $M_\odot$, $a \simeq 2.8$ $R_\odot$, and $e \simeq 0.62$) for the famous Hulse–Taylor pulsar (Hulse & Taylor, 1975), we find $t_{\rm gwr} \simeq 300$ Myr.

The merger of such objects will result in a burst of gravitational waves. The first detection of such waves was made by the LIGO consortium on 14 September 2015 (Abbott *et al.*, 2016).

#### 3.1.2.5 Other Time Scales

Many other time scales can be identified and defined. For example, time scales for specific processes, such as resonant relaxation (Rauch & Tremaine, 1996) or the Lidov–Kozai (Lidov, 1962; Kozai, 1962)[3] cycles in hierarchical triple systems, may be relevant in certain circumstances. In Sections 7.2.2 and 8.4.3, we present examples for which these additional time scales are important.

### 3.1.3 Star Cluster Dynamics

Stars form in clustered environments, and much of the story of stars in our Galaxy has to do with the dynamical evolution of star clusters. This book is not explicitly about

[2] The `Mikkola` integrator listed in Table 3.1 includes post-Newtonian corrections up to order 2.5.

[3] The Lidov paper was originally published in 1961 in Russian as Iskusstvennye Sputniki Zemli, #8, p. 5; the cited article is the English translation.

cluster dynamics, but clearly the behavior of stars in clumps of hundreds to millions is critical to our discussion. The *N*-body codes described in this chapter are central to this study. Because this topic plays such a key role in our narrative, we pause here to introduce some terminology, a few common diagnostics having to do with basic cluster dynamics, and an evolutionary timeline. We will return frequently to these concepts in our later discussion. The dynamics we describe are undoubtedly going on in, and may be modified by, the presence of a surrounding gas cloud, but the essential underlying processes are largely unchanged. However, we can only scratch the surface here, so we refer the reader to the extensive literature for more in-depth discussion (see, e.g., Spitzer, 1987; Heggie & Hut, 2003; Binney & Tremaine, 2008).

#### 3.1.3.1 Cluster Structure and Diagnostics

We begin with a newborn group of stars formed from an interstellar cloud (see Chapter 5). Both theory and observation (e.g. Gnedin *et al.*, 2015) suggest that these stars are unlikely to have formed in precise virial (dynamical) equilibrium (Section 3.1.2.1). However, simulations indicate that the system will virialize and form a more or less spatially smooth cluster in virial equilibrium within a few dynamical times, erasing any substructure associated with the formation process. For typical cluster parameters, the time scale to form a single, well-defined cluster is a few million years. From this point on (at least, to the extent that we can neglect overall rotation), we are justified in treating the cluster as a roughly spherically symmetric object. Consequently, simulations of the evolution of clusters after the formation stage generally use idealized Plummer- (Plummer, 1911) or King-like (King, 1966) models (Section 2.1) as initial conditions.

By this stage, clusters are simple enough that they can be described by a few parameters. The virial (or, as we have seen, the half-mass) radius defines the overall cluster scale. Theoretical cluster models generically predict—and most observations show—the presence of a roughly constant-density central region known as the core. Observers define the *core radius*, $r_c$, as the distance from the cluster center at which the surface brightness drops to half the central value. Because density is generally hard to measure in *N*-body models, theorists use a somewhat different definition, defining the core radius as the density-squared weighted root mean square stellar distance from the cluster center. (We also think that this sounds complicated!) Perhaps remarkably, the two definitions seem to broadly agree. The ratio of the core radius to the half-mass radius (or to the cluster's tidal radius in the ambient Galactic tidal field) is a key measure of cluster structure.

The cluster's *Lagrangian radii* are, by definition, the distances from the cluster center containing specified fractions of the total cluster mass. The 50% Lagrangian radius is the half-mass radius. Studying selected Lagrangian radii of key mass groups provides critical insight into the cluster's evolving density distribution. Figure 3.1(a) shows an *HST* image of the globular cluster M80. The cluster's core radius and half-mass radius are indicated by dashed red circles. Figure 3.1(b) shows the density profile of the best-fitting King model (with $W_0 = 7.5$; see Section 2.1). The cluster's tidal radius lies well outside the image shown in part (a); for many clusters, the tidal radius is determined from the best-fitting King model.

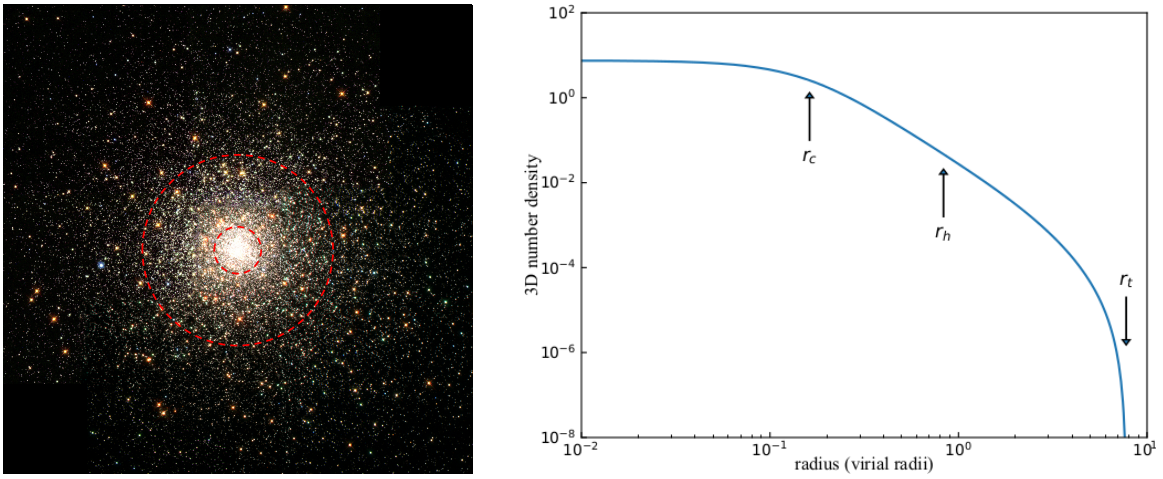


Figure 3.1: (a) The globular cluster M80 lies 10 kpc away and contains several hundred thousand stars. The cluster's core and half-mass radii are indicated by dashed red circles. The field of view is 3 arc minutes across. (b) The King model that best fits the cluster, with core, half-mass, and tidal radii marked. (Credit: F. R. Ferraro (ESO/Bologna Obs.), M. Shara (STSci/AMNH) et al., & the Hubble Heritage Team (AURA/STScI/NASA). Reproduced with permission.)

#### 3.1.3.2 Essential Dynamics

As we just saw, dynamical friction causes the most massive stars in the cluster to sink toward the center. The time scale can be very short, often less than the evolutionary lifetimes of the stars themselves (see Section 3.1.2.3). The most importance consequences of this process are: (1) the surrounding lower-mass stars gain energy and expand outward as the massive stars lose energy and sink to the center of the cluster potential well; (2) the high density of massive stars near the center means a greatly increased rate of both dynamical interactions (including binary formation) and physical collisions between stars (see Sections 5.2.6 and 7.4.4), which can significantly influence the dynamical evolution of the system; (3) when stars lose mass—for example, due to winds or supernovae—the effect on the cluster is much stronger when the mass loss comes from the center rather than the periphery.

Mass segregation is often interpreted in terms of energy equipartition, the expected end-point of relaxation, which should lead to $\langle mv^2 \rangle$ = constant for every stellar mass group. As a result, more massive stars slow down and sink toward the center of the cluster (Spitzer, 1969). However, perhaps surprisingly, the degree to which a system actually achieves equipartition is still not fully understood. Trenti & van der Marel (2013) reported that energy equipartition is never actually reached in *HST* observations and *N*-body simulations of clusters with realistic mass functions. On the other hand, simulations of more idealized two-component systems (containing just "light" and "heavy" stars; see below) indicate that equipartition is achieved, at least when the component mass ratio is not too large. The fine details of this fundamental process remain to be resolved.

On longer time scales, the escape of high-velocity stars (evaporation) and the internal effects of two-body relaxation, which transports energy from the inner to the outer regions of the cluster, lead to a phenomenon known as core collapse (Antonov, 1962; Cohn, 1980). During this phase, the central portions of the cluster accelerate

toward infinite density while the outer regions expand. The process can be understood by recognizing that, according to the virial theorem (Equation (3.3)), a self-gravitating system has negative specific heat, so reducing its energy causes it to heat up. Hence, as energy moves from the (dynamically) warmer central core to the cooler outer regions, the core contracts and gets warmer, accelerating the contraction. The time scale for the process to reach completion (i.e., a core of formally zero size and infinite density) is $t_{\rm cc} \sim 15 t_{\rm rh}$ for a system of identical masses, and significantly less in the case of a broad spectrum of masses (Inagaki & Saslaw, 1985), for which the core-collapse time scale falls to $t_{\rm cc} \sim 0.2 t_{\rm rh}$ (Portegies Zwart & McMillan, 2002).

Figure 3.2 illustrates some of these dynamical stages. It shows the evolution of a simple two-component stellar system, in which both components are initially distributed as the same $W_0 = 5$ King model. The mass ratio of the two components (1 and 2) is $\mu = m_2/m_1 = 3$, and the ratio of total masses is $M_2/M_1 = 1.6 \times 10^{-2}$. According to Equation (3.7), the half-mass relaxation time is $t_{\rm rh} \simeq 300$ time units. Mass segregation, dynamical equipartition, core collapse, and dynamical binary formation and evolution in the more massive component can all be seen.

Several physical processes can counteract core collapse. As just mentioned, stellar mass loss (due to stellar winds or supernovae) tends to unbind the cluster, and mass segregation amplifies the effect on the core. In addition, binary systems, whether present initially in the cluster or formed dynamically (see Section 3.7.3), can also play an important role. Briefly, "soft" binaries that have less binding energy than the mean stellar kinetic energy of the system tend to be destroyed by encounters with other stars. However, "hard" binaries (with binding energy greater than the mean) tend to become more tightly bound following an encounter, effectively heating the stellar system (Hills, 1975; Heggie, 1975). We will discuss in more detail how binary interactions and dynamics are handled in AMUSE in Section 7.5.

One important guideline in understanding global cluster dynamics is Hénon's principle (Hénon, 1975): When two coupled dynamical processes operate on very different time scales—here, core energy production and the much slower outward conduction of energy through the half mass radius—the faster process will always come into equilibrium with the demands of the slower one. As a result, the overall evolution of the cluster—long-term expansion and dissolution in the Galactic tidal field—is largely independent of the detailed physics operating in the core. Regardless of the detailed core dynamics, a typical cluster is expected to dissolve in the Galactic tidal field in about 10 half-mass relaxation times (Lee & Ostriker, 1987).

There is considerable literature about the late stages of star cluster evolution, in what is often referred to as the post-collapse phase, during which so-called gravothermal oscillations may play a major role (Bettwieser & Sugimoto, 1984; Makino, 1996). Although this epoch is of considerable interest from a theoretical perspective, it is not clear what its observational relevance is in our relatively young universe.

### 3.1.4 Physics of the Integrator

Evaluating Equation (3.1) is not hard, nor is integrating the equations of motion, but it is important to appreciate the limitations of the adopted methods. It is sometimes

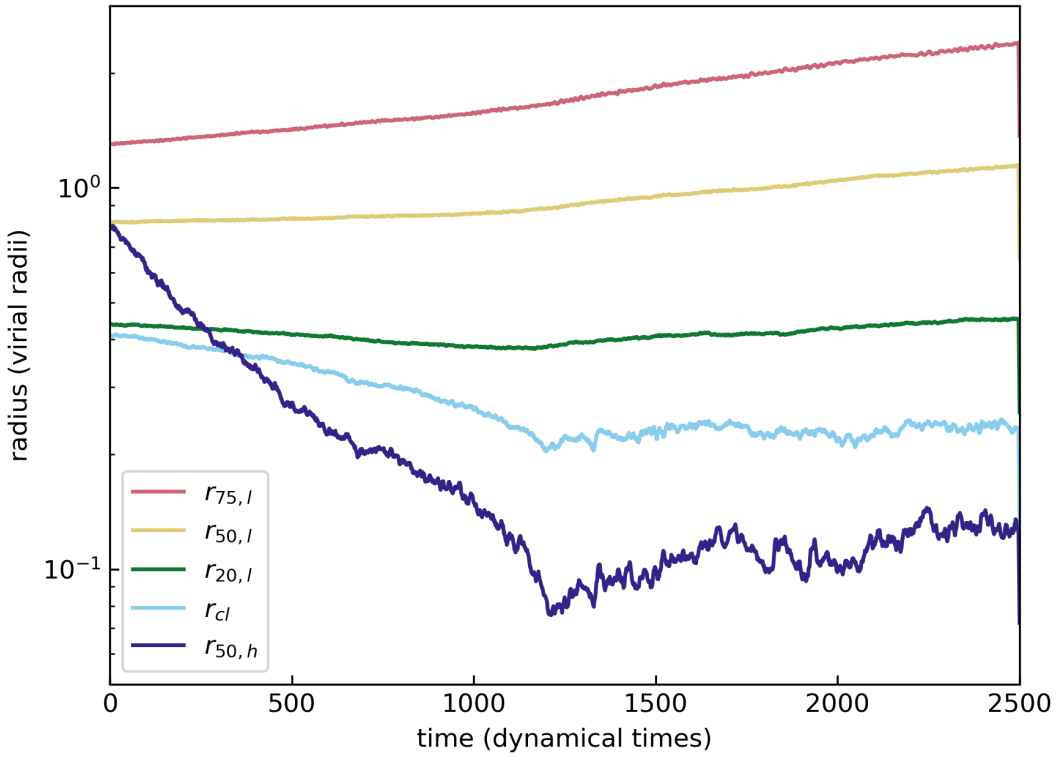


Figure 3.2: Evolution of a simple two-component system comprising 16k "light" (1) and 88 "heavy" (2) stars, with mass ratio $m_2/m_1 = 3$. The half-mass radius of the massive component, and the core and 20%, 50%, and 75% Lagrangian radii of the light component, are shown. The data have been smoothed using a square window of total width 6 time units. Mass segregation occurs during the first 600 time units; subsequently, both the massive system and the light core contract in quasi-equilibrium. The system remains close to energy equipartition for $t > 600$. Eventually, after core collapse at $\sim 1200$ time units, binary formation and dynamical heating cause both the massive system and the light core to expand. (Data courtesy of M. Brewer.)

said that gravity solvers, and $N$-body solvers in particular, have no underlying assumptions, and that all physical processes are properly taken into account. This, of course, is untrue. They solve the equations of motion in the non-relativistic, point-mass approximation, quietly neglecting the spatial extent of the gravitating objects and the effects of their tidal interactions. For many studies, this does not pose a problem, but it is nevertheless important for the user to be aware of these shortcomings.

In solving the fundamental equations that govern the dynamical evolution of a self-gravitating system, we generally assume that the following conditions are met.

1. **Point-mass approximation**. An important and common underlying assumption is that all particles are point masses. Even if the particles should have nonzero sizes, their non-sphericity is generally ignored. The center of mass is

assumed to be the geometric center of the particle. The consequences of this assumption are probably very small for most applications, but there certainly are cases where it is unwarranted. In Section 7.4.1, we will discuss a case in which the size of objects actually drive part of the physics, and it will turn out that dimensions of stars and planets are important drivers for interesting physical phenomena. In many hydrodynamical applications (discussed in Chapter 5), the finite size of objects motivates the performance of numerical simulations.

2. **Newton is correct**. The force in Equation (3.1) is pure Newtonian dynamics, ignoring relativistic corrections. The improvements made by Einstein (1916) and Verlinde (2017) are usually ignored. Some implementations of the $N$-body problem in AMUSE include additional terms that take relativistic effects into account, but most do not. Currently, there is no treatment of modified Newtonian dynamics (Milgrom, 1983) in AMUSE, and no prescription for dark energy. These purported physical effects may be taken into account by modifying the spatial geometry or by introducing corrections to the force law, realized via the Bridge coupling strategy (see Chapter 8). Dark matter can be mimicked by introducing additional gravitating particles, although this is a coarse-grained approximation to the real physics.

3. **Gravity acts instantaneously**. In classical theories of gravitation, the "speed of gravity" is the rate at which a change in the distribution of energy or momentum of matter results in a change in the gravitational field it produces. In a more physically correct sense, the "speed of gravity" refers to the speed of a gravitational wave. In Newtonian $N$-body dynamics, this speed is infinite; in Einstein's theory of general relativity, it is the speed of light. This discrepancy can be safely neglected in many simulations because typical gas and stellar velocities are much smaller than the speed of light.

#### 3.1.4.1 Numerical Accuracy

The digital computers on which we perform our calculations are often limited to only 16 decimal places.[4] This sounds like a lot for everyday calculations, but it is quite limited if one considers that some simulations may require $\gtrsim 10^{16}$ operations (10 Petaflop), in which case round-off must play a potentially important role. This amount of computing corresponds roughly to the number of operations required to integrate the orbit of the Moon over the lifetime of the solar system. The effect of round off is rarely studied in detail in $N$-body calculations, and the assumption is always made that it does not affect the long-term evolution of the system.

The solution of an $N$-body problem are deterministic. Initial conditions define one and only one trajectory in phase space. Each initial point in phase space then maps to a final state of the system. In practice, we expect the calculated phase-space trajectory of an $N$-body simulation to differ from the true phase-space trajectory—$N$-body simulations give the wrong results (see Portegies Zwart & Boekholt, 2018). Deviations

[4] In 64-bit IEEE-754 compliant numerical representation, each floating point number is presented by 1 bit for the sign, 11 for the exponent, and 52 for the mantissa.

from the trajectory, and the eventual location of the final state, can be caused by external perturbations, numerical round-off, and other sources of error. These sources of error may themselves be non-deterministic (e.g. in case partial forces are calculated in parallel).

In Section 3.4.2 we present an example of how to measure the degree of chaos in an N-body system, by comparing a more accurate solution with a apprehensive solution.

If the errors are truly random and without systematic effects, the final state of an ensemble of inaccurate calculations should be centered around the true final state. Even though we do not know the shape or the dispersion of this distribution, the mean should still be consistent with the true solution. In a computer simulation, any initial point in phase space will map to a final state-point, but this is not the true final point. It is commonly believed that if we generate an ensemble of initial points in phase space and calculate them all to their final points in phase space, the distribution of final points will represent the true solution for the given initial distribution.

Every implementation of the $N$-body problem has its own characteristic numerical fingerprint that can be recognized by the growth of the numerical error. This is caused, in part, by specific choices of additional parameters, such as the time-stepping scheme or the introduction of a softening length that limits the depth of the potential. These choices are often hidden deep within the code, and it is generally assumed that they don't materially affect the outcome of the simulation. Despite these potential sources of error and uncertainty, our confidence in $N$-body simulations rests on the basic observed fact that all implementations lead to similar outcomes when applied to standard problems.

The detailed long-term evolution of a star cluster—virialization, mass segregation, core collapse, and long-term evolution (see Section 3.1.3)—appears to be substantially independent of both the initial model and the specific integration technique used. This has been repeatedly verified in a host of code comparisons over the past three decades (Chernoff & Weinberg, 1990; Giersz & Heggie, 1994; Fukushige & Heggie, 1995; Giersz & Heggie, 1996; Takahashi & Portegies Zwart, 1998; Spinnato *et al.*, 2003; Anders *et al.*, 2012; Whitehead *et al.*, 2013; Rodriguez *et al.*, 2016). As a rule of thumb, we can trust predictions of the bulk or statistical properties of the system, but we should not take individual stellar orbits too seriously.

## 3.2 *N*-body Integration Strategies

In principle, the force calculation and the pushing of the particles are separate operations in a gravitational dynamics code, but they are coupled in most "pure" $N$-body implementations (see Section 3.2.2). Calculating the force in Equation (3.1) is straightforward, but the number of operations for the force on a single particle scales as the number of particles $[\mathcal{O}(N)]$, and calculating the forces on all particles then scales as $\mathcal{O}(N^2)$, which becomes impractical for current computers without GPU acceleration for $N \gtrsim 10^4$.

### 3.2.1 Global Structure of an *N*-body Code

Figure 3.3 presents a simple workflow for a basic *N*-body code. A handy programmer can probably rewrite this in about 100 lines of source code. Such codes can be (and have been) tuned for parallel operation, graphical processing units, and other high-performance/exotic hardware, and they form an excellent basis for subsequent numerical studies. Converting the flow chart in Figure 3.3 into a working program is not entirely trivial, but we refrain from going into more detail here because there are many excellent textbooks that do just that (e.g. Aarseth, 2003; Heggie & Hut, 2003). [5]

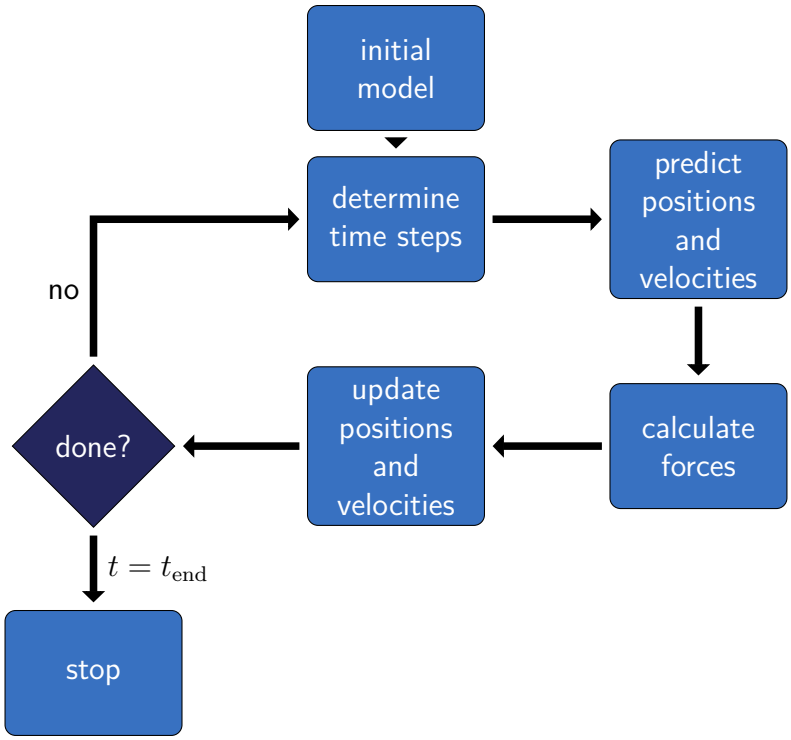


Figure 3.3: Workflow for a typical direct *N*-body code. **[removed the initialization ,most Nbody codes just read input – Inti]**

The heart of an *N*-body code is the integration scheme, the mathematical time-discretization used to solve Equation (3.1). An important consideration in choosing an integration scheme is its order, which tells us how the algorithmic error (per step) $\delta E$ scales with time step $\delta t$. For a scheme of order $k$, $\delta E \propto \delta t^k$. Typical schemes in common use are of second-, fourth-, or sixth-order. Higher-order schemes are preferred in direct-summation *N*-body calculations because a given accuracy can be achieved using longer time steps.

The program illustrated in Figure 3.3 uses a predictor–corrector scheme to advance the particles in time (see Section 3.2.3). There are many alternative approaches, but these schemes have become the de facto standard in astrophysical *N*-body simulations.

[5] For an illustration of how this problem can be addressed in a number of programming languages, see nbabel.org. STEVEN: URL is dead...

### 3.2.2 Types of *N*-body Code

Several families of algorithms have been developed to solve the $N$-body problem. Some scale as $\mathcal{O}(N \log N)$ (Barnes & Hut, 1986), some scale better, $\mathcal{O}(N)$ (Ambrosiano *et al.*, 1988; Dehnen, 2014), some much worse, $\mathcal{O}(N^4)$ (Boekholt & Portegies Zwart, 2015). The differences are rooted in optimization, where computational speed is, broadly speaking, traded for accuracy. We draw a distinction between *pure* $N$-body, *direct* $N$-body, and *approximate* $N$-body codes.

- Pure $N$-body codes solve Newton's equations of motion with no free physical parameters, although most have some capacity to flag special events, such as close encounters or binary dynamics. The only adjustable quantity in a pure $N$-body code is the time-stepping criterion used to integrate the equations of motion. Examples are `ph4` and `Hermite`, both based on the fourth-order Hermite predictor–corrector scheme (Makino & Aarseth, 1992), (cf. Hut *et al.*, 1995, for a lower-order approach). Tunable precision $N$-body codes an interesting subset of pure $N$-body codes. They go beyond the standard double precision of digital computers, and allow energy conservation to be tuned to arbitrary precision. The code `Brutus` (Boekholt & Portegies Zwart, 2015) has an extra free parameter, which controls the mantissa of the floating-point operations.

- **Direct *N*-body codes** solve Newton's equations of motion with additional parameters, often to speed up the code or to reduce round-off. Many such codes incorporate regularization techniques for resolving close encounters and close multiple subsystems, and/or neighbor schemes to reduce the force calculation time. The cost per step of these codes still scales as the square of the number of particles. `NBODY6` is a prime example (Aarseth, 1985; Harfst *et al.*, 2007).

- **Approximate methods** relax the strict $N^2$ procedure for computing the gravitational force. These include various flavors of tree codes, particle-mesh, and other multipole treatments. Such methods are very popular, mainly because they are very fast. The most common example is the Barnes–Hut tree scheme (Barnes & Hut, 1986), in which forces from nearby particles are calculated directly, but more and more distant particles are grouped together into clumps of increasing size, whose gravity is approximated by the first few terms of a multipole expansion. Very often, for computational efficiency, only the monopole term is retained—when calculating the orbit of the Sun in our Galaxy, it is reasonable to treat the Andromeda galaxy as a point mass. The result is a reduction of the $\mathcal{O}(N^2)$ complexity of a direct force calculation to $\mathcal{O}(N \log N)$. Examples of tree codes in AMUSE include `BHTree` (Barnes & Hut, 1986), `MI6` (Iwasawa *et al.*, 2015), and `Bonsai` (Bédorf *et al.*, 2012; Bédorf *et al.*, 2014). Another popular method is the fast multipole method (Greengard & Rokhlin, 1987; Ambrosiano *et al.*, 1988), which scales even more favorably—$\mathcal{O}(N)$.

### 3.2.3 Discretization Strategies in *N*-body Simulations

A system of differential equations, such as Equation (3.1), can be discretized in time and solved using a variety of numerical techniques. Two basic strategies for obtaining a solution are *explicit* and *implicit* schemes. An explicit method provides a formula or procedure that allows the state of the system at time $t + \delta t$ to be determined entirely from the state of the system at time $t$. In an implicit method, the state of the system at time $t+\delta t$ also appears in the formula, leading to an implicit equation that generally must be solved iteratively.

Implicit methods tend to be more stable and (in principle) allow much longer time steps, although they are generally more expensive due to the iterative solution entailed. However, the longer time steps are often unattainable due to the chaotic and singular nature of the underlying equations—the characteristic time scales of the motion can differ by many orders of magnitude from particle to particle and from one time to another. As a result, explicit methods are generally favored over implicit methods for *N*-body applications.

Many explicit schemes exist, but by far the most commonly used approach in *N*-body simulations is the predictor–corrector scheme (see Section 3.2.2). Historically, these methods appeared as the first iteration in implicit solutions to the discretized equations of motion, but in their modern incarnation they have become explicit solvers.

Broadly speaking, predictor–corrector schemes proceed in three steps:

- The predictor step uses whatever derivative information is available at time $t$ (usually at least the acceleration, but often higher derivatives as well) to extrapolate all positions and velocities at time $t + \delta t$ as Taylor series.
- The accelerations (and higher derivatives) are then reevaluated using the predicted positions and velocities, and the new and old accelerations are combined to yield estimates of higher derivatives. For example, we can estimate the "jerk" (the derivative of the acceleration) as $j \simeq (a^{\rm new} - a^{\rm old})/\delta t$.
- The corrector step then refines the predicted positions and velocities using the additional derivative information, essentially adding extra terms to the Taylor series.

These schemes are compact, easy to program, and generally proceed with a single evaluation of the accelerations at each step (the new accelerations are reused in the next prediction), making them very efficient in terms of computer resources.

The second-order predictor-corrector scheme (often called the velocity Verlet method, and sometimes implemented as the "Leapfrog" scheme; see Verlet, 1967) is very widely used; we discussed in Chapter 1 we demonstrated how this integrator finds its origin in the Babylonian civilication 350 to 50 BC. Note that the terminology between Leaf-frog, Verlet and Bablyonian scheme are a bit confusing. The velocity-Verlet does not correct the new positions. The predictor and corrector steps are usually reserved for methods that do at least one iteration, but possibly more. Although of low order, its simplicity and the fact that it is symplectic (see Section 3.2.3.1) make it an attractive choice in many applications. It uses only accelerations. The fourth-order Hermite scheme

(Makino, 1991) is a natural generalization of the Verlet method (except that it is not symplectic). It uses accelerations and jerks and has for some time been the method of choice in large $N$-body simulations. The last decade has seen the development and implementation of sixth- and eighth-order versions in large-scale integrators. Again, it goes beyond the scope of this book to dwell on this topic, as excellent texts already exist. We direct the interested reader to Hockney & Eastwood (1988) and Nitadori & Makino (2008).

One disadvantage of explicit schemes is that errors on successive steps tend to be correlated, so the cumulative error grows faster than the random walk expected for uncorrelated errors (Brouwer, 1937), for which the net error would grow as the square root of the number of steps. This can be solved with compensated Kahan-Babuška summation (Kahan, 1965; Babuška, 1969). In most applications, this more rapid error growth is an acceptable trade-off for the improved speed of the method.

The recently revived family of hybrid methods, which combine the good qualities of two or more integration algorithms (e.g. McMillan & Aarseth, 1993), have very promising characteristics. A recent example is MI6 (Oshino *et al.*, 2011; Iwasawa *et al.*, 2015), which combines a Hermite direct code with a Barnes-Hut tree code[6]. Finally, it is worth mentioning some recent Kepler-splitting methods, in which the problem is separated into $N$ Kepler problems plus perturbations. The superposition principle is subsequently applied to knit the $N^2$ Kepler solutions together into a coherent $N$-body solution. The methods works, parallelizes almost trivially, and the errors do not grow on long time scales (compared to the dynamical time), making such methods very suitable for studying the long-term dynamical evolution of the solar system (Wisdom & Holman, 1991; Yoshida, 1993; Duncan *et al.*, 1998; Chambers, 1999; Laskar & Robutel, 2001; Saha & Tremaine, 1992; Rein & Liu, 2012; Liu *et al.*, 2016).

#### 3.2.3.1 Symplectic and Time-Reversible Schemes

In the context of long time scale calculations, the so-called symplectic integrators are of particular interest. The defining property of a symplectic scheme is that it preserves phase-space area in a dynamical system. It follows that the phase-space volume element is preserved—the familiar Liouville Theorem. Given that Hamiltonian systems also have these properties, symplectic integration of Hamiltonian systems yields numerical solutions that stay faithful to the true solutions, in the sense that physically conserved quantities are conserved numerically over long periods of time. Such schemes are also time-reversible, meaning that the initial conditions can be recovered (to machine accuracy) by running a simulation backward in time (see also Tremaine (2015)).

In a symplectic scheme, it is common to split the underlying Hamiltonian into slowly and rapidly varying parts. For example, when discussing the orbits of minor bodies in the solar system, we could rewrite the Hamiltonian of the multibody system as

$$H = H_{\rm kep} + H_{\rm int}. \tag{3.15}$$

[6]Steven: this is not MI6, but Pikachu and Pentacle!

Here, we follow the example of Wisdom & Holman (1991). The leading term $H_{\rm kep}$ represents interactions between the minor bodies with the Sun, while $H_{\rm int}$ represents interactions among the minor bodies. (The latter term could, in turn, be further separated into changes on time scales that are short or long compared to the dynamical time.) Each component of the split Hamiltonian can be solved independently, using explicit schemes, and the solutions combined. We will return to symplectic schemes in Chapter 8, where we will adopt this approach as our standard for combining codes.

Explicit symplectic schemes can be very accurate, approaching machine precision with sufficiently small time steps, even when the equations are stiff. Their main drawback is that they require the time steps for all objects to be the same (Wisdom & Holman, 1991). Such shared time step schemes are often employed for planetary, galaxy-scale, and cosmological simulations, but are prohibitively expensive for star cluster simulations, which generally require individual and time-variable steps.

In computational astrophysics, probably the most widely used symplectic integrator is the Verlet (leapfrog) method or Babylonian scheme described in Section 3.2.3. Although only of second order, its simplicity and long-term stability make it the method of choice for many applications. It is often implemented in "standard" symplectic form as a kick–drift–kick scheme:

$$\begin{aligned} v' &= v_n + \tfrac{1}{2}a(x_n)\delta t \\ x_{n+1} &= x_n + v'\delta t \\ v_{n+1} &= v' + \tfrac{1}{2}a(x_{n+1})\delta t. \end{aligned} \tag{3.16}$$

A moment's reflection reveals that this is equivalent to the second-order predictor–corrector implementation described earlier. We note that, in practice, the acceleration used in the first kick of the next step is the same as that used in the last kick of the current step, meaning that only one acceleration need be computed per step, a fact of great importance in large $N$-body calculations.

An alternative (also symplectic) drift–kick–drift scheme is

$$\begin{aligned} x' &= x_n + \tfrac{1}{2}v_n\delta t \\ v_{n+1} &= v_n + a(x')\delta t \\ x_{n+1} &= x' + \tfrac{1}{2}v_{n+1}\delta t. \end{aligned} \tag{3.17}$$

This approach is much less widely used than its kick–drift–kick cousin, mainly because of its significantly poorer performance in applications where variable time steps are required (Springel, 2005).

We will discuss these methods in more detail in Chapter 8. Higher-order symplectic schemes also exist (e.g. Yoshida, 1990; Rein & Spiegel, 2015; Tricarico, 2012), and are sometimes employed in astrophysical contexts. Finally, we note that time-reversibility in even a non-symplectic scheme is sufficient to ensure that the obtained solution stays close to the true solution and that the error in the energy does not drift. Hut *et al.* (1995) describe an iterative algorithm to make any integrator time-reversible; it is implemented in the AMUSE `smallN` module.

#### 3.2.3.2 General relativistic effects

Most $N$-body codes in AMUSE adopt Newton's equations of motion, but there are a few exceptions that also solve for the general relativsitic equations of motion.

One example is given in Figure 3.4 where we integrate the first stable 3-body solution found by Montgomery (1998). Usually this system is integrated in the Newtonian regime, but here we also show a solution where the system is relativistic. Even though, we only integrate for a short time span, it is clear that the relativistic case the system is considerably less stable than in the Newtonian case.

This example was calculated using `hermite_grx` using the 2.5th order Taylor expansion to the Einstein-Infeld-Hoffman equations of motion with a scling constant of $\gamma = 0.01c$. The stellar masses were 1 $M_\odot$ and the mutual distance 1 au. For a usual speed of light, this system is barely relativsitic (represented with the thick green curve). But when we scale the speed of light to $\gamma = 0.01c$, the orbits deviate considerably from their Newtonian trajectories. Interestingly, one naively would expect the system to shrink, rather than the wider excursions as observable in Figure 3.4. In the relativistic case, two of the three stars collide (top middle). The setting up of the code is listed in Listing 3.1.

```
95      n_body_code = Hermitegrx
96      grav = n_body_code(converter)
97      pert = "2.5PN_EIH"
98      grav.parameters.perturbation = pert
99      grav.parameters.integrator = "RegularizedHermite"
100     grav.parameters.dt_param = 0.01
101     grav.parameters.light_speed = relative_light_speed * constants.c
102
```

Listing 3.1: Setting-up the relativistic gravitational N-body code to integrate to 2.5th order post-Newtonian terms with the regularized Hermite integrator. The full script can be found in `amuse.examples.montgomery`.

#### 3.2.3.3 Parallelization and Hardware Acceleration

The $N$-body problem is relatively easy to parallelize. Several algorithms are known to perform well for each of the families of integration strategies mentioned in Section 3.2.2. However, we will not dwell on the details, as there are many excellent papers and textbooks already written on these topics. The problem is also well suited to hardware acceleration using graphics processing units (GPUs). Although worthy competitors exist, the most common acceleration hardware currently used in the gravitational $N$-body community is the NVIDIA GPU, programmed using the CUDA language.[7] These tiny computers combine large amounts of computing power with relatively low prices, consistently leading the market in both teraflops per dollar and teraflops per watt. Many of the $N$-body codes listed in Table 3.1 are parallel, GPU-accelerated, or both.

[7] See https://developer.nvidia.com/cuda-zone

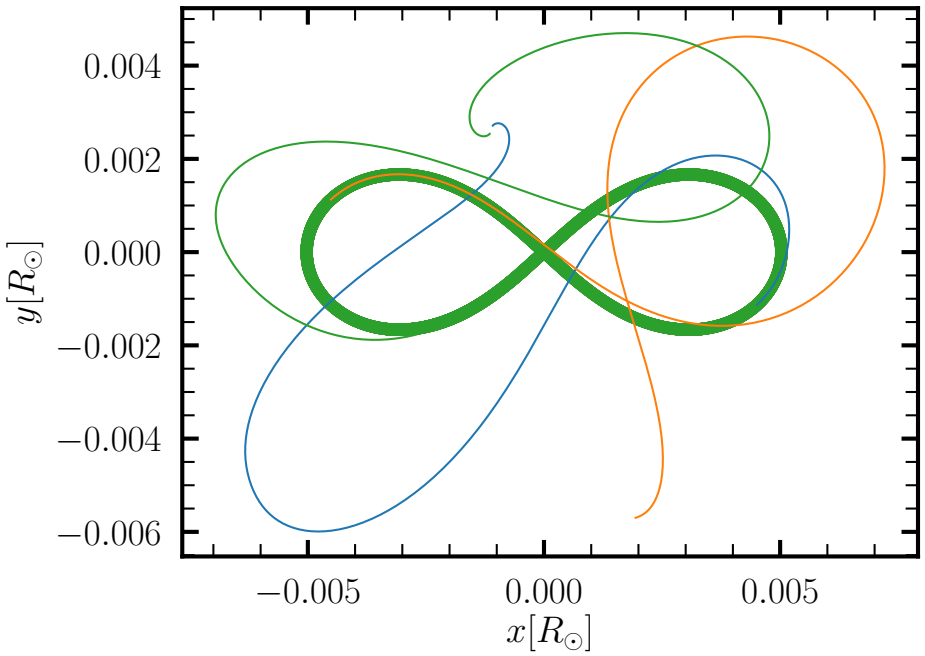


Figure 3.4: Montgomery's (1998) 3-body problem, or the so-called stable figure 8 configuration in Newtonian gravity (thick green curve) for three black holes of $1\,M_\odot$. The three thin curves (green, orange and blue) give the trajectories for the same initial configuration but when the velocities of the particles approach the speed of light. The calculations were conduced using `hermite_grx` with a scaling to the speed of light of $0.01c$. Whereas the Newtonian case shows the expected behavior predicted by Montgomery, the relativistic case ends in a merger between two, and eventually all three stars. The script to generate this figure can be found in `amuse.examples.gravity.montgomery`.

## 3.3 The AMUSE interface to Gravity Solvers

The preceding discussion of gravity solvers may mask a very basic fact: the codes have all very similar input and output. And this is necessarily so, since they aim to solve the same physics. We take advantage of this in AMUSE to define a minimal Gravity Interface that can be used to access any gravitational solver in a homogeneous way.

A simple script to integrate Newton's equations of motion is presented in Listing 3.2. You can test it yourself with the following code:

```
import amuse.examples as ae
ae.run_example("gravity_minimal")
```

We will dissect the complete example in some detail here. The script starts by importing the required AMUSE packages:

```
from amuse.units import nbody_system
from amuse.ic.plummer import new_plummer_model
from amuse.community.hermite import Hermite
```

The first import of the `nbody_system` makes a *unit system* available (we will explain this in more detail below). The second import statement loads a function `new_plummer_model` that generates the initial conditions. Finally the third import loads a class, `Hermite` which is an example of a *community code* that *implements* the

```
"""
Minimal example for gravity calculations in AMUSE
"""

from amuse.units import nbody_system
from amuse.ic.plummer import new_plummer_model
from amuse.community.hermite import Hermite


def gravity_minimal(number_of_stars=100, time_end=1 | nbody_system.time):
    """
    Initialises a Plummer sphere of length 'number_of_stars' and integrates until
    'time_end' with the Hermite integrator
    """
    stars = new_plummer_model(number_of_stars)

    gravity = Hermite()
    gravity.particles.add_particles(stars)

    energy_total_initial = gravity.kinetic_energy + gravity.potential_energy

    gravity.evolve_model(time_end)

    energy_total = gravity.kinetic_energy + gravity.potential_energy
    energy_error = (energy_total_initial - energy_total) / energy_total_initial

    print(
        f"model time = {gravity.model_time} "
        f"total mass = {stars.total_mass()} "
        f"total energy = {energy_total} "
        f"relative energy error = {energy_error}"
    )

    gravity.stop()


def main():
    """
    Runs example with default parameters.
    """
    gravity_minimal(number_of_stars=100, time_end=1 | nbody_system.time)


if __name__ == "__main__":
    main()
```

Listing 3.2: Minimal AMUSE code for solving a simple $N$-body problem. Source code is available in `amuse.examples.gravity_minimal`.

gravitational dynamics interface. All the codes that conform to this interface (and we will encounter more of them later) implement in some way (discussed in Section 3.2.2) a solver for Equation (3.1).

The code in Listing 3.2 is as simple as we could make it—and honestly, it doesn't

do much. It can, however, be the basis for more rewarding investigations and it does contain in essential form all necessary ingredients of more complicated scripts. As we will see in later chapters, it can be expanded and used for far more complex simulations. This minimal script, and many of the following scripts, is set up as follows:

- generate a realization of the starting conditions (Section 3.3.1),
- specify and initialize the (gravity) solver to use (Section 3.3.2),
- evolve the model to a specific time (Section 3.3.4),
- print diagnostics,
- produce a figure of the results or analyze the data (Section 1.4.3),
- shutdown the community code.

After loading in the appropriate packages the Python interpreter processes the definition of the function `amuse.examples.gravity_minimal` and proceeds to the line

```
if __name__ ==  "__main__":
```

This piece of Python tells the interpreter that the following lines are to be ignored when the script is not run standalone (see Appendix F). It is standard practice in Python to do this as it makes your code easily reusable: another script can import the function `gravity_minimal` without triggering execution of a run.

Here, the main function is called with its default parameters.

```
    gravity\_minimal(number_of_stars=100, time_end=1 | nbody_system.time)
```

The quantity `time_end` defined here has a unit, which is attached to the numerical value using the "`|`" operator.[8] Gravitational $N$-body codes often use a dimensionless system of $N$-body units (sometimes called Hénon units; Hénon, 1972; Heggie & Mathieu, 1986).[9] In this system, the gravitational constant $G$, the total cluster mass $M$, and the cluster virial radius $R_{\rm vir}$ (see Section 3.1.2.1) each have a value of 1. For a cluster in virial equilibrium, the kinetic, potential, and total energies then are

$$T = \frac{1}{4}, \quad U = -\frac{1}{2}, \quad E = -\frac{1}{4}. \tag{3.18}$$

In this case, the unit of `time_end` is the dimensionless $N$-body time unit. It may seem odd to assign a "unit" with no "dimension" to an already dimensionless parameter, but the $N$-body units in AMUSE, called `nbody_system`, are not so much dimensionless as scaleless. This convention is useful because it enables dimension checking and allows combination with other dimensional units. The $N$-body units in AMUSE imply that $G$ has the numerical value 1 (but not necessarily the other two conditions).

[8] In the units module of the `astropy` package (see http://www.astropy.org/), the "`*`" operator is used instead.

[9] See the excellent overview at `http://en.wikipedia.org/wiki/N-body_units`.

*N*-body units are handy when we are interested in the outcome of a pure gravitational problem. Internally, *N*-body integrators have almost universally adopted them, as well as in *N*-body initial condition generating functions such as `new_plummer_model`, `new_king_model`, etc. (see Section 2.1). They greatly facilitate comparison between different codes. In addition, they afford the minor computational advantage of keeping the magnitudes of most physical variables close to unity, although they make little difference—at best, avoiding a single multiplication by $G$ in each force calculation. On the downside, dimensionless units have caused confusion among generations of students and other noncognoscenti in their attempts to reconcile physical and *N*-body quantities. And as soon as other physical processes intrude, such as when combining gravitational dynamics with stellar evolution or hydrodynamics, *N*-body units rapidly lose their usefulness.

In reality, a typical *N*-body code does not care at all about units. It accepts a number with no questions asked, except perhaps whether it is a floating point number or an integer. However, the AMUSE interface does care about dimensional and non-dimensional units, because it mediates the conversion between the units used in different modules. For this reason, any calculation involving *N*-body units that plans to use other modules with units must be calibrated by defining a `converter` specifying the *N*-body mass, length, and/or time scales in physical units (see Section 3.3.5). In Section 3.4 (Listing 3.5 in particular) we introduce the command-line argument parser `argparse`, and present an example of a script in which we use this with units (see **??** in Appendix B Appendix B.4.2).

### 3.3.1 Generating Initial Conditions

The main routine in Listing 3.2, where the real work is done, starts with

```
def minimal(number_of_stars=100, time_end=1 | nbody_system.time):
```

the declaration of the function that actually does the work. The next line

```
    stars = new_plummer_model(number_of_stars)
```

generates the initial density and velocity of the particles, in this case a Plummer model with 'number_of_stars' stars. The result is a list of `stars`, which contain the identities, masses, positions and velocities of the $N$ particles generated. You could inspect the content of the `stars` at this point by adding an additional line

```
    print(stars)
```

This would result in something like (do absolutely try this yourself!):

```
      key      mass    radius          vx  ...         x          y          z
        -      mass    length length/time  ...    length     length     length
=========  ========  ========  ==========  ... =========  =========  =========
120995...  1.00e-02  0.00e+00   7.666e-01  ...  1.95e-01  -2.01e-01   7.47e-02
 88511...  1.00e-02  0.00e+00   1.219e-01  ... -1.65e+00  -1.55e+00  -2.92e+00
 19874...  1.00e-02  0.00e+00  -2.131e-01  ...  3.08e-02   1.20e+00  -1.18e+00
      ...       ...       ...         ...  ...       ...        ...        ...
 18055...  1.00e-02  0.00e+00  -3.412e-01  ... -1.90e+00   2.35e-01   2.04e+00
 46733...  1.00e-02  0.00e+00   6.214e-01  ... -2.43e-01  -1.97e-02  -1.61e-01
 67731...  1.00e-02  0.00e+00  -1.089e-02  ... -8.33e-01   7.62e-01   2.43e+00
=========  ========  ========  ==========  ... =========  =========  =========
```

Note that the output format from your code will look somewhat different, as we have abbreviated the output due to page limitations (and we did not specify the seed in the random number generator).

Here the `key` is a unique identifier for each particle, followed by the mass, radius, velocity, and position. Except for the first (`key`) entry, these lines may appear in any order, and the columns also may appear in a different order. The first line identifies the parameter, while the second gives the unit. In this example the units are generic (scaleless) because we have not specified any units explicitly.

We refer to this list as a *particle set*. We could construct an empty particle list with `stars = Particles(N)` and subsequently set the masses, for example as we did in the previous chapter (see Section 2.3.1):

```
stars.mass = [0.2, 0.2, 0.2, 0.2] | nbody_system.mass
```

If all particles have the same mass (as in this example), we could equivalently write

```
stars.mass = 0.2 | nbody_system.mass
```

In this way we can also set the positions and velocities of all $N$ stars. Initializing the solar system was explicitly demonstrated in Listing 2.6 of Section 2.3.1.

### 3.3.2 Specifying and initializing the gravity solver

The next two lines in the listing initialize an instantiation of the $N$-body integrator, in this case the `Hermite()` integrator, and send the previously created generated particles to the integrator. Much of the magic in AMUSE is in this simple line :

```
    gravity = Hermite()
```

The result is a detached process called `hermite_worker` that starts on your computer, running simultaneously with the Python program that spawned it. You can check that multiple processes are now running on your computer by typing

```
%> top
```

on the command line (use `top -o cpu` on a Mac), which in our case shows something like:

```
...
13369 amuse   20   0 99.0m 4452 2692 R  100  0.1   2:07.80 hermite_worker
13365 amuse   20   0  231m  32m 8184 R   99  0.6   2:07.02 python
...
```

The `hermite_worker` process is the *N*-body integrator. This code polls the AMUSE framework asking for things to do. The Python instance `gravity` is our portal to the *N*-body code. When we are done with `Hermite` we terminate it by

```
    gravity.stop()
```

This will send a kill signal to `Hermite`, and all data in the code will be lost.

`gravity` is an instance of a class implementing the AMUSE gravity interface. This object manages and transparently handles communication with the `hermite_worker` process or any other *N*-body code (see Section 1.4.3). Any of the other gravity solvers can be used in the same way by changing the call to the integrator. For example:

```
    gravity = Bhtree()
```

instantiates and initializes the gravity solver `BHTree`, a Barnes-Hut tree code. Aside from this line, the interface is the same in all cases, as long as the other code is a gravity code.

### 3.3.3 Feeding Particles to the *N*-body Code

So far, we have not yet informed the *N*-body code which data it should use. These data come in the form of a particle set. In Section 3.3.1, we initialized a particle set that we called `stars`. We can send this particle set to the *N*-body code by adding the `stars` to the code already running

```
gravity.particles.add_particles(stars)
```

We now have two sets of particles—those in `stars` and the set that now lives in the memory of the running code. The latter set can be addressed directly by querying `gravity.particles`. Once we tell the gravity solver to advance in time, these two particle sets will diverge. The particle set in Python will remain unchanged, but the copy resident in the integrator will be updated by the community code, in our case a `Hermite` instance.

### 3.3.4 Evolving the Model to the Desired Time

The actual computational work is done in

```
gravity.evolve_model(time_end)
```

Here, the *N*-body code runs to time `time_end`, i.e. the solution to Equation (3.1) is advanced to the desired end time. After the code is done, we calculate the kinetic and potential energies.

To test the accuracy of the integrator, we print the final energies and the energy error since the system was initialized. Note that, because the *N*-body code is a detached process, we must send it a termination signal from the AMUSE script at the end of the calculation. Otherwise the *N*-body code will keep running in the background, until the Python interpreter cleans up ("garbage collects") the `gravity` variable.

We are now ready to run the minimal script with the following arguments

```
%> python -m amuse.examples.gravity_minimal
```

which will produce the following (slightly edited to fit the text column) output:

```
T= 1.0 time M= 1.0 mass E= -0.250000001272 mass * length**2 * time**-2
       Q= -0.5068237292374607 dE= -5.089542761314269e-09

In this session you have used the AMUSE modules below.
Please cite any relevant articles:

   "AMUSE-Hermite (framework 2023.10.1.dev1+gb159086e4.d20240124)"
           ADS:1995ApJ...443L..93H (Hut, P., Makino, J. & McMillan, S., *
    Astrophysical Journal Letters* , **443**, L93-L96 (1995))


   "AMUSE (unknown version)"
           DOI:10.5281/zenodo.1435860
           ADS:2018araa.book.....P (Portegies Zwart, S. & McMillan, S.L.W., 2018)
           ADS:2013CoPhC.183..456P ** (Portegies Zwart, S. et al., 2013)
           ADS:2013A&A...557A..84P ** (Pelupessy, F. I. et al., 2013)
           ADS:2009NewA...14..369P (Portegies Zwart, S. et al., 2009)
```

**SPZ: Make remark about gravity.parameters.end_time_accuracy_factor = 0.5 #old default**

**[I think it would be better if the model time actually is actually closer to the target time, otherwise we need to discuss the meaning of the tend in the evolve model right here! – Inti]**

This final message report to the user the codes and components used and provides references that we hope he/she will cite if the code and script are run for production and/or publication purposes.

This concludes our first acquaintance with an AMUSE simulation script. Of course some elements are still missing to be able to analyse and change our simulation. We want to be able to examine the data in more detail (Section 1.4.3), store the data for publication and analysis (Section 1.4.4), change to the units used to communicate with the code (Section 3.3.5). Also, in order to change the numerics of the simulation we need to be able to change solver parameters (Section 3.3.6) and to change to a different solver (Section 3.3.7).

### 3.3.5 Scaling the simulation

So far, we have discussed only dimensionless *N*-body units. These are convenient for performing scale-free simulations, but very often in astronomy we have a specific system in mind, or we may want to couple a dimensionless *N*-body code to a dimensional code,

such as a stellar evolution or hydrodynamics module. Non-gravitational codes depend critically on units, even though the units are generally assumed and rarely made explicit in the source. Often constants, such as the gravitational constant $G$, the Boltzmann constant $k$, the speed of light $c$, and the Planck constant $h$, are unapologetically hard-coded in some preferred system.

As mentioned in Chapter 1, units are fully supported in AMUSE. Running an $N$-body code with units is not difficult, but it does lose the scale-free characteristics of the gravitational problem. The units in an $N$-body system are specified by providing any two of the mass, length, and time scales of the problem. As a practical matter, conversion to other systems requires the creation of a `converter`, which is a Python class specifically designed to keep track of units. We can construct a converter with mass unit 1 $M_\odot$ and spatial unit 1 au as follows:

```
converter = nbody_system.nbody_to_si(1 | units.MSun, 1 | units.au)
```

However, the following line

```
converter = nbody_system.nbody_to_si(72 | units.kg, 175 | units.cm)
```

which happens to be the weight and height (erroneously) listed in Simon's passport, would work equally well. (Steve won't reveal the corresponding numbers.) In this last example we really did use the International System of Units, whereas for most astronomical simulations, we are more likely to use solar masses, millions of years, and parsecs. Nevertheless, the conversion system is still generically called `nbody_to_si`.

Again, if you are unfamiliar with Python classes, don't despair. A `converter` is simply an object that contains knowledge of the units in a problem, which can be passed as needed as an argument to other functions requiring this information. For example, a unit converter can be provided when initiating an $N$-body solver:

```
gravity = Hermite(converter)
```

This establishes the connection between dimensionless and physical units. Now we can feed particle sets to the $N$-body code in any units we choose, and retrieve dimensional information in any convenient set of units. The conversion is automatic as long as we stay within one system of units.

```
print(f"Earth's mass is: {(1 | units.MEarth).in_(units.kg)}")
```

or

```
print(
    f"The Sun's luminosity is about {(3.8e+33 | units.erg/units.s).in_(units.LSun)}"
)
```

A system in physical units can be converted back to $N$-body units using the same converter. For example, converting the positions back to $N$-body units can be accomplished with

```
positions_in_nbody_units = converter.to_nbody(bodies.position)
```

or back to SI units:

```
positions_in_si_units = converter.to_si(positions_in_nbody_units)
```

More information about alternative converters can be Appendix B.2.5

### 3.3.6 Changing Parameters of the simulation

Often the functionality of a community code needs to be modified, for example to improve accuracy or set some other parameter. This is controlled by internal parameters within the code. Changing those internal parameters is always possible, but we want to avoid modifying and recompiling the code. For this reason, many internal code parameters can be queried and (re)set. The names of these parameters may be the same for some codes, but completely different for others. Fortunately, as described in Section 1.4.1, Python and the AMUSE interface have the ability to provide help on these matters. The command

```
dir(Ph4)
```

provides a extensive list of all the ingredients of the direct $N$-body code `ph4`. The same command with the code `BHTree` will provide a different list of functionalities for the tree code. More extended information can be obtained with

```
help(Ph4)
```

A list of parameters for `ph4`, and their default values, can be obtained with

```
gravity.parameters?
```

A somewhat more elaborate explanation of the parameters is provided by

```
help(gravity.parameters)
```

The parameters used in two seemingly similar codes may have different names. This sounds very confusing, but it reflects the diversity of the authors of the various interfaces. Our philosophy here is to try to reserve identical names for parameters that do exactly the same thing, but because many codes have slightly different functions or objectives, it is hard to be consistent and clairvoyant at the same time. One case where we failed to be consistent is in the two direct $N$-body codes `ph4` and `Hermite`. The parameters to control the size of the time step in these codes are called `timestep_parameter` and `dt_param`, respectively. A comparable parameter in `BHTree` is called `timestep`. However, the latter is not inconsistent because `timestep` really sets the actual time step in `BHTree`.

Some parameters control the actual working of a code, while others govern hardware specifics. In the previous section, we discussed how to specify that `ph4` should use a graphical processing unit in your computer (assuming that AMUSE was compiled with runtime support for this hardware). The same effect can be achieved with

```
gravity.parameters.use_gpu = True
```

A very important parameter in many $N$-body codes is the softening parameter

$\varepsilon$ (Aarseth, 1963). Softening removes the singularity in the inverse-square force by replacing Equation (3.1) with

$$\mathbf{a}_i = G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{\left[(\mathbf{r}_j - \mathbf{r}_i)^2 + \epsilon^2\right]^{3/2}}. \tag{3.19}$$

This is done to accommodate integration schemes that can't handle point-mass potentials. Column 8 of Table 3.1 lists whether or not a code is (or can be) softened. In Section 7.5 we describe a module that adds point-mass support to any otherwise softened code. The square of the softening parameter is generally referred to as `epsilon_squared` in all $N$-body codes. For example, $\varepsilon^2$ can be set to the squared mean interparticle spacing, in $N$-body units, with

```
gravity.parameters.epsilon_squared = 1./N**(2./3)
```

In case you prefer to look at good old-fashioned source code, these parameter settings are all defined in the interface file of each community module in the AMUSE source code. For the `Hermite` package, for example, the parameters can be found in the file `${AMUSE_DIR}/src/amuse/community/hermite0/interface.py`, in the function `define_parameters()`.

### 3.3.7 Changing the solver used in the simulation

**[landing spot for all commented out stuff on the different nbody intgrators and the performance graph – Inti]** As mentioned in Chapter 1, the `amuse.lab` package includes all of the commonly used $N$-body codes in AMUSE, such as `Hermite`, `Huayno`, `Mercury`, `BHTree`, `ph4`, and `Bonsai`, but there are many others, see Appendix B.7. It also includes the $N$-body system of units `nbody_system` (see below), and packages for generating initial conditions, in this case `new_king_model`. Each of these needs to be imported before it can be used in the script. Some of the imported codes are threaded, multi-processor, parallelized for distributed memory machines, and may provide support for acceleration hardware, such as GPUs. Table 3.1 gives a brief overview of the current codes in AMUSE and their capabilities. Note that, while some codes, such as `ph4`, `SmallN`, `Huayno`, `BHTree`, and `Bonsai`, were added for strategic reasons, to fill a specific need, many others have been contributed by AMUSE users who used the framework to tackle some problem.

Figure 3.5 shows the performance of a variety of $N$-body solvers in AMUSE. Initial conditions are those of a Plummer sphere in virial equilibrium. No softening was used, but the runs were carried out for only 10 $N$-body time units; no significant close encounters occurred during this short time, and therefore no softening was necessary. We see two distinct performance behaviors in Figure 3.5. At the upper left, we have a group of direct $N$-body codes with $O(N^2)$ scaling. At right center, we see the ($N$log($N$)) behavior characteristic of tree codes. Code-to-code differences span about an order of magnitude and are the result of implementation details. The fastest code (bottom red curve) is the GPU-native `Bonsai`; the hardware characteristics of the GPU architecture explain its relatively poor performance for up to about 1000 particles but much better performance for larger numbers of particles.

Table 3.1: Overview of the $N$-body codes incorporated in AMUSE. Here TC refers to tree code (see Section 3.2.2), and PN to the possibility of including post-Newtonian terms. References: 1 Boekholt & Portegies Zwart (2015), 4 Spera *et al.* (2012); Capuzzo-Dolcetta *et al.* (2013), 5 Harfst *et al.* (2007), 7 Rein & Liu (2012, 2011), 8 Jänes *et al.* (2014), 9 Chambers & Migliorini (1997), 10 Gaburov *et al.* (2010), 11 Bédorf *et al.* (2012), 12 Oshino *et al.* (2011); Iwasawa *et al.* (2015), 13 Mikkola (1983); Mikkola & Merritt (2008), 15 Jänes *et al.* (2014), 16 Aarseth (1985); Wang *et al.* (2015), 17 Barnes & Hut (1986), 18 Hamers & Portegies Zwart (2016).
The columns give the order of the integrator, the language used, the number of particles for which the code is most suitable and whether or not the code can handle softening, is parallelized or is GPU enabled.

| name | ref | type | order | language | $N$ | soft. | parallel | GPU |
|---|---|---|---|---|---|---|---|---|
| Brutus | 1 | pure | 2nd | C++ | $\lesssim 10$ | no | yes | no |
| Hermite | | pure | 4th | C++ | $\lesssim 10^4$ | n/y | yes | no |
| ph4 | | pure | 4th | C++ | $\lesssim 10^5$ | n/y | yes | yes |
| HiGPUs | 4 | pure | 6th | C++ | $\lesssim 10^4$ | n/y | yes | yes |
| PhiGRAPE | 5 | pure | 4th | F77 | $\lesssim 10^5$ | n/y | yes | no |
| SmallN | | pure | 4th | C++ | $\lesssim 10$ | n/y | yes | no |
| Rebound | 7 | direct | var | C | $\lesssim 10^4$ | no | yes | yes |
| Huayno | 8 | direct | sympl. | C | $\lesssim 10^3$ | no | no | no |
| Mercury | 9 | direct | sympl. | F77 | $\lesssim 10^4$ | no | no | no |
| octgrav | 10 | approx. | TC 2nd | C++ | $10^4$–$10^7$ | yes | yes | yes |
| Bonsai | 11 | approx. | TC 2nd | CUDA | $10^4$–$10^{11}$ | yes | yes | yes |
| MI6 | 12 | approx. | PN 4th | C++ | $\lesssim 10^6$ | n/y | yes | yes |
| Mikkola | 13 | direct | PN 4th | F77 | $\lesssim 100$ | no | no | no |
| TwoBody | | direct | Kepler | Python | 2 | no | no | no |
| Sakura | 15 | direct | Kepler | C++ | $\lesssim 10^5$ | no | yes | no |
| NBODY6++ | 16 | direct | 4th | F77 | $\lesssim 10^5$ | no | yes | yes |
| BHTree | 17 | approx. | TC 2nd | C++ | $10^3$–$10^6$ | yes | yes | no |
| SecMult | 18 | approx. | orbit av. | C++ | $\lesssim 10$ | no | no | no |

## 3.4 Examples

### 3.4.1 Integrating the Orbits of Venus and Earth

In Section 2.3.1, we presented a plot of the integrated orbits of the Sun, Venus, and Earth (Figure 2.4). Now we can improve our understanding of the scripts used to make this figure.

Listing 3.3 presents the initialization routine, in which the masses, positions, and velocities of the Sun, Venus, and Earth are stored in a particle set. After initialization, the particle set is moved to the center of mass of the three-body system, to prevent the entire system from drifting, since we initialized it with the Sun at the origin. Subsequently, if we chose, we could give the solar system a position and systematic velocity in space by writing

```
particles.position += (-8.5, 0.0, 0.0) | units.kpc
particles.velocity += (11.1, -228.3, 7.25) | units.kms
```

These are the approximate position and velocity of the Sun relative to the Galactic center. These lines are best included after repositioning the solar system to its center

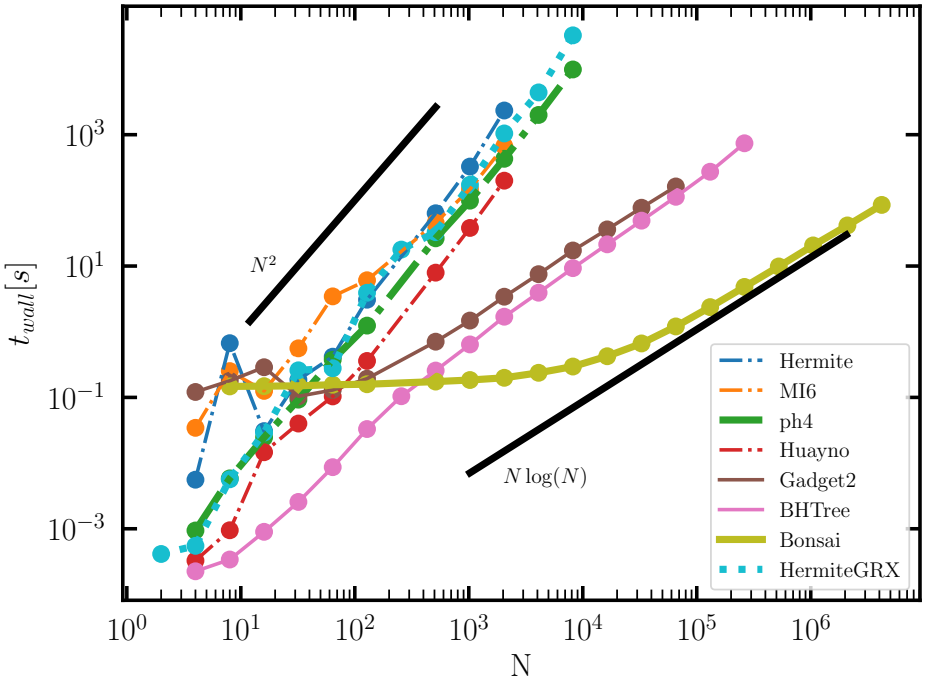


Figure 3.5: Wall-clock time as a function of $N$ for a range of $N$-body solvers in AMUSE, integrating of a Plummer sphere for 1 $N$-body time unit. The runs were performed for 10 $N$-body time units. A script to generate this figure can be found at `${AMUSE_DIR}/examples/textbook/plot_Nbody_performance.py`.

of mass—otherwise the Sun, rather than the solar system barycenter, would be placed at this location (although the difference is very small).

The integration routine presented in Listing 3.4 follows the orbital motion of Venus and Earth. In the first few lines, we load specific parts of the AMUSE lab and units modules. Given the particle set in the argument list, we use the total mass and the distance to one of the particles (Earth—1 au) to initialize the converter. This converter is subsequently used to initialize the $N$-body code. We then add the particles to the $N$-body code and make special references to the two planets, Venus and Earth.

In the following lines, we initialize lists of $x$ and $y$ positions, which we update in the subsequent loop over time, with a time step of one day. In this case, we do not bother to initialize a channel to communicate between the framework and the running code. The function with a channel would be essentially the same, but would run slightly faster. After the time loop, we stop the $N$-body code and return the $x$- and $y$-position lists of the two planets.

Instead of storing the positions of Venus and Earth in arrays, we could write snapshots to a file at the end of each loop, using `write_set_to_file()`, and subsequently analyze the data with a separate script. To improve our diagnostics, we might also print or store the energy error produced by the $N$-body integrator at every diagnostic output interval.

```
13 from amuse.datamodel import Particles
14 from amuse.units import units
15
16 # import amuse.examples.plot as aplot
17
18
19 def sun_venus_and_earth():
20     particles = Particles(3)
21     sun = particles[0]
22     sun.name = "Sun"
23     sun.mass = 1.0 | units.MSun
24     sun.radius = 1.0 | units.RSun
25     sun.position = (855251, -804836, -3186) | units.km
26     sun.velocity = (7.893, 11.894, 0.20642) | (units.m / units.s)
27
28     venus = particles[1]
29     venus.name = "Venus"
30     venus.mass = 0.0025642 | units.MJupiter
31     venus.radius = 3026.0 | units.km
32     venus.position = (-0.3767, 0.60159, 0.03930) | units.au
33     venus.velocity = (-29.7725, -18.849, 0.795) | units.kms
34
35     earth = particles[2]
36     earth.name = "Earth"
37     earth.mass = 1.0 | units.MEarth
38     earth.radius = 1.0 | units.REarth
39     earth.position = (-0.98561, 0.0762, -7.847e-5) | units.au
40     earth.velocity = (-2.927, -29.803, -0.0005327) | units.kms
41
42     sun.color = "#FFFF00"
43     venus.color = "wheat"
44     earth.color = "deepskyblue"
45
46     particles.move_to_center()
47     return particles
```

Listing 3.3: Program to initialize the orbits of Venus and Earth around the Sun. The full script can be found in `amuse.examples.sun_venus_earth`.

#### 3.4.1.1 Plotting the Results

We can subsequently plot the resulting data file from the simulation using the `pyplot` routines in the `matplotlib` package. An example code is presented in Listing 3.5. If we did not choose to store the data, we could instead simply have passed the arrays returned by `integrate_solar_system()` directly to the plotting routines. Alternatively, we could make a simple animation of the data, as presented in Listing 3.6.

#### 3.4.1.2 Calculating Orbital Elements as Diagnostics

It is often useful to convert the six-dimensional Cartesian coordinates of a two-body system into Kepler orbital elements—semimajor axis, eccentricity, phase, etc. Because this is a common and important conversion, AMUSE offers a special routine to compute

the orbital semimajor axis and eccentricity of any two-body system. A more detailed description of the use of Kepler elements, including discussion of some efficiency considerations, is presented in Section 7.3.2.

The two particles for which orbital elements are to be calculated must be stored in a separate particle set containing only those particles. For example, when we consider the solar system calculation in Section 3.4.1 (Listing 3.3), we can declare a subset of local particles `SunEarth` containing the Sun and Earth, respectively.

```
SunEarth = Particles()
SunEarth.add_particle(particles[0])
SunEarth.add_particle(particles[2])
```

If desired, we could create a separate channel to this particle set:

```
channel_from_to_SunEarth = gravity.particles.new_channel_to(SunEarth)
```

The orbital elements can then be determined by calling

```
orbital_elements = orbital_elements_from_binary(SunEarth, G=constants.G)
```

This function is part of the extended AMUSE package, imported via

```
from amuse.ext.orbital_elements import orbital_elements_from_binary
```

The Kepler package underlying this function is written in standard $N$-body units and requires some notion of the value of Newton's constant $G$, which is provided as an argument. The routine returns a tuple containing the masses of the primary and secondary stars and the six Kepler elements: semimajor axis, eccentricity, inclination,

```
54 from amuse.units import nbody_system
55 from amuse.community.huayno import Huayno
56
57
58 def integrate_solar_system(particles, end_time):
59     convert_nbody = nbody_system.nbody_to_si(
60         particles.mass.sum(),
61         particles[1].position.length(),
62     )
63
64     gravity = Huayno(convert_nbody)
65     particles_in_code = gravity.particles.add_particles(particles)
66
67     while gravity.model_time < end_time:
68         gravity.evolve_model(gravity.model_time + (1 | units.day))
69         particles_in_code.new_channel_to(particles).copy()
70         particles.savepoint(timestamp=gravity.model_time)
71
72     gravity.stop()
73
74     return particles
```

Listing 3.4: Program to integrate the orbits of Venus and Earth around the Sun. (The full listing is in `amuse.examples.sun_venus_earth`.)

```
import matplotlib.pyplot as plt
from amuse.plot import scatter, xlabel, ylabel
from amuse.io import read_set_from_file


def plot(particles):
    plt.figure(figsize=(8, 8))

    colormap = ["yellow", "green", "blue"]  # specific to a 3-body plot
    size = [40, 20, 20]
    edgecolor = ["orange", "green", "blue"]

    for snap in particles.history:
        scatter(snap.x, snap.y, c=colormap, s=size, edgecolor=edgecolor)
    xlabel("x")
    ylabel("y")

    save_file = "plot_gravity.png"
    plt.savefig(save_file)
    print(f"\nSaved figure in file {save_file}\n")
    plt.show()
```

Listing 3.5: Minimal routine to plot $N$-body simulation data. Full source code is available in `amuse.examples.plot_gravity`.

argument of periapsis, line of the ascending node, and true anomaly.

We can integrate the inner solar system and study the variation in Earth's semimajor axis and eccentricity over an interval of 90,000 years. We present this evolution in Figure 3.6. Instead of initializing the solar system from the simplified initial conditions given earlier in Listing 3.3, we use AMUSE's built-in generator for any Julian date. To generate an initial realization of the solar system at the Julian date of 2,438,871.5 day, we write

```
particles = new_solar_system(Julian_date=2438871.5 | units.day)
```

Figure 3.7 presents the results of the numerical integration of the solar system over a much longer time scale. We will use this script in Section 3.4.1.3 to compare the variations in the eccentricity of Earth's orbit versus the temperature measurements at the Vostok station in Antarctica.

#### 3.4.1.3 Temperature Record at Vostok Station

While the cycles in Earth's orbital elements are intrinsically interesting, it is even more interesting to compare them with observations. Instead of integrating just the innermost two planets, we include all eight planets in the solar system. In order to draw a comparison with observations, we integrate the solar system backward in time and check the results against the historical temperature record obtained from the ice at Vostok station in Antarctica. The backward calculation can be realized using one of the time-reversible $N$-body codes in AMUSE, or simply by reversing the velocities and integrating forward in time (the latter method was adopted here).

Figure 3.8 shows the measured changes in the temperature at Vostok station and

```
import matplotlib.pyplot as plt
from matplotlib import animation
from amuse.io import read_set_from_file
from amuse.units import units


def animate(x, y):
    """
    Creates and shows an animation of the xy position of a particle.
    """

    def update(i):
        while i >= number_of_snaps:
            i -= number_of_snaps
        off = []
        for j in range(len(x[i])):
            off.append(x[i][j])
            off.append(y[i][j])
        off = np.array(off).reshape(3, 2)
        scat.set_offsets((off))
        return (scat,)

    number_of_snaps = len(x)
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(1, 1, 1)
    ax.set_xlim(-1.2, 1.2)
    ax.set_ylim(-1.2, 1.2)

    colormap = ["#FFFF00", "wheat", "deepskyblue"]
    size = [40, 20, 20]
    edgecolor = ["orange", "wheat", "deepskyblue"]

    ax.set_facecolor("black")
    scat = ax.scatter(x[0], y[0], c=colormap, s=size, edgecolor=edgecolor)
    anim = animation.FuncAnimation(fig, update, interval=100)
    plt.show()
```

Listing 3.6: Minimal routine to animate $N$-body simulation data. Full source code is available in `amuse.examples.anim_gravity`.

the numerically computed changes in Earth's orbital eccentricity over the past 450,000 yr. The variations in eccentricity seem to correspond to variations in temperature (and $CO_2$) in Earth's atmosphere (Gildor & Tziperman, 2000). It is interesting to note that the sharp rise in temperature seems to correlate with a rising eccentricity when it exceeds about 0.035. It has been suggested that these variations in the Earth's orbital eccentricity drive the 100,000 year ice-age cycle found in the historical temperature record from the Vostok ice core (Milankovitch, 1941), but Earth's obliquity is important too.

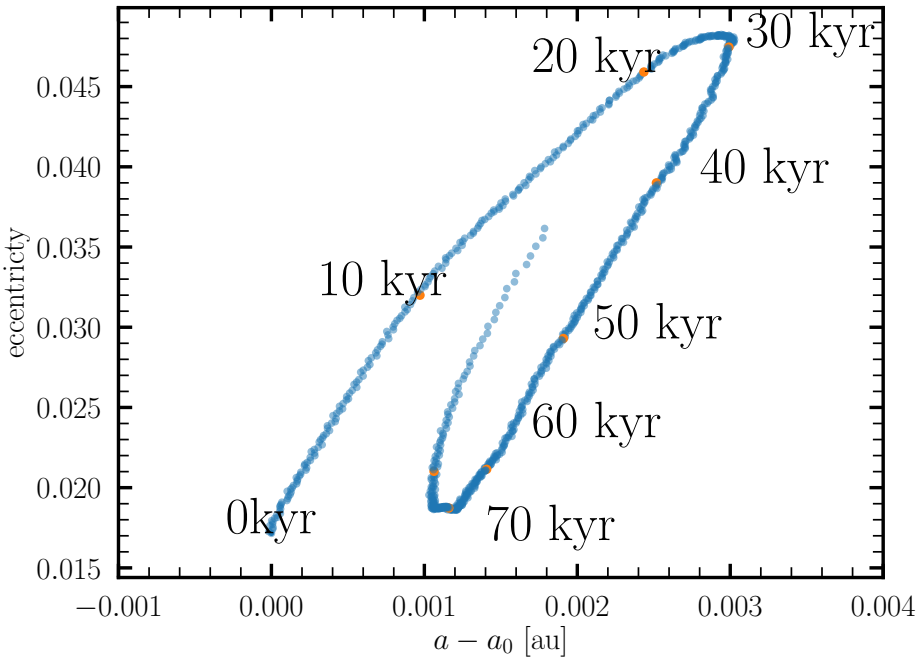


Figure 3.6: Earth's orbital semimajor axis ($x$-axis) and eccentricity ($y$-axis) as functions of time over a period of 90,000 yr. Initial conditions are taken from Table 2.1. The complete listing for the script that produced this figure is in `${AMUSE_DIR}/examples/textbook/earth_orbit_variation.py`. It should take a minute or so to run on a laptop.

### 3.4.2 Fighting exponential divergence

```
gravity = Brutus()
gravity.parameters.bs_tolerance = bs_tolerance
gravity.parameters.word_length = word_length
gravity.parameters.dt_param = dt_param

gravity.particles.add_particles(bodies)
channel_to_framework = gravity.particles.new_channel_to(bodies)
```

### 3.4.3 Secular Multiples

When integrating planetary systems, sometimes resolving the individual orbital phases of all planets is simply too time consuming, particularly when the system is sufficiently stable that an approximate approach can suffice. In those cases, we can orbit-average the equations of motion. The basic idea behind such orbit averaging is the "wire" method, in which each planet is imagined to be spread out over its Keplerian orbit as though on a wire. Along that imaginary wire, the local density is proportional to the time spent at that orbital phase, with higher density near apocenter and lower density near pericenter. This approach can result in a substantial increase in computational speed. `SecularMultiple` is such a code (Hamers & Portegies Zwart, 2016) in AMUSE.

In `SecularMultiple`, the secular (i.e., orbit-averaged) evolutions of hierarchical multiple systems are composed of binary orbits, in much the same way as is done in

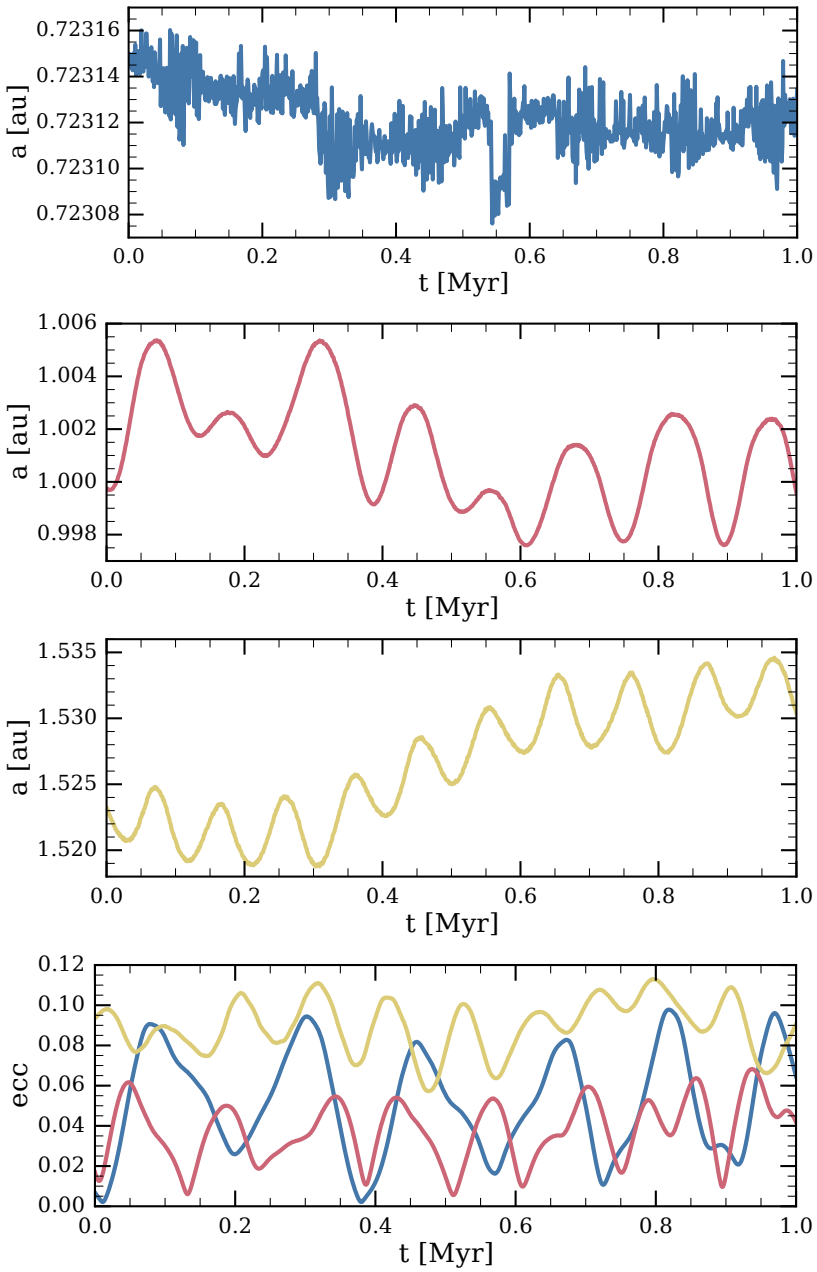


Figure 3.7: From top to bottom, the respective evolutions of the orbital semimajor axes of Venus (blue), Earth (red), and Mars (yellow)[10] for the next million years. The bottom figure gives the orbital eccentricities of the three planets as functions of time, using the same color coding. The integration was performed using a script similar to that presented in Listing 2.7.

`Sakura` (see Table 3.1 or Appendix B.7), with an arbitrary number of bodies. For application domains, one can think of a hierarchical triple system with two stars in an "inner" binary orbited by a third body, a planetary system with a central star and several planets, or a hierarchical quintuple system with multiple planets. The code is based on an expansion of the Hamiltonian of the system in terms of ratios of orbital separations, which are assumed to be small. Subsequently, the Hamiltonian is averaged

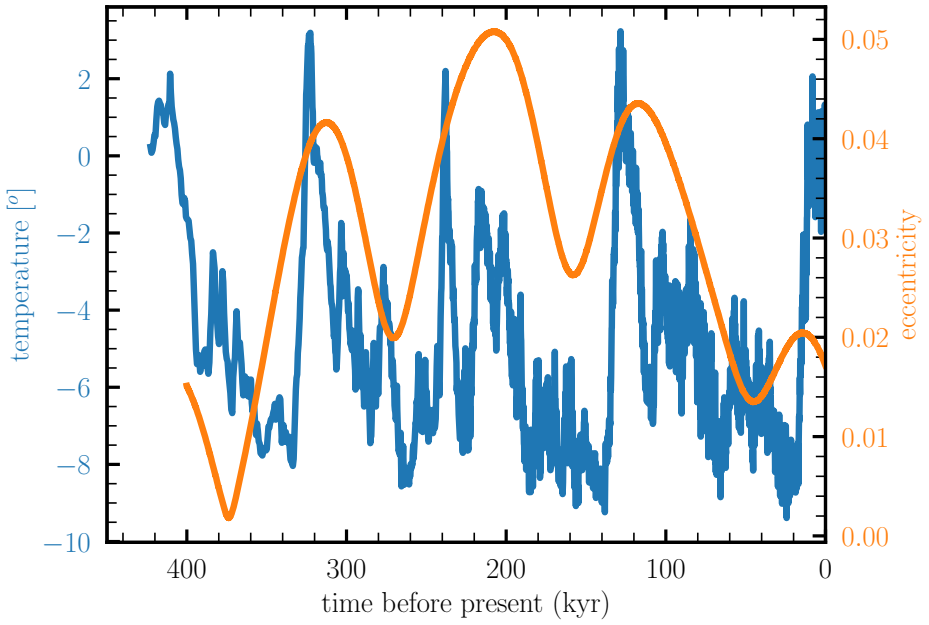


Figure 3.8: (Blue) Historical isotopic temperature record from the Vostok ice core (Jouzel *et al.*, 1987; Petit *et al.*, 1999). (Red) Orbital eccentricity evolution of the Earth over the last 400,000 yr. Integration was performed using `Huayno` with an accuracy parameter $\varepsilon = 10^{-3}$. The full listing of the script that produced this figure is in `${AMUSE_DIR}/examples/textbook/vostok_1999_temperature.py`. The Vostok ice-core temperature measurements are available at `${AMUSE_DIR}/examples/textbook/vostok_1999_temperature.data`.

analytically over all orbits and the equations of motion are solved numerically. Speed is the main advantage of this approach: the code runs faster than any direct $N$-body integrator while still capturing the secular evolution. Individual terms in the expansion can easily be switched on or off to provide a better understanding of their effects on the evolution of the system. A downside is that the system must be hierarchical, as non-secular effects are not captured. In particular, systems in which mean-motion resonances are important (such as tightly packed planetary systems) should be avoided.

We illustrate `SecularMultiple` for a "2 + 2" quadruple system: two binaries orbiting one another. As in other $N$-body codes in AMUSE, `SecularMultiple` works with particle sets. However, in addition to specifying parameters for the individual particles, the hierarchy of the system must also be specified. The declaration of particle attributes in a hierarchical quadruple system is illustrated in Listing 3.7. These attributes connect particles to one another. In this case, a body is a particle with a mass, and with attribute `is_binary` set `False`. Binaries are also part of the particles set, but with a True value for `is_binary`. Each binary has two children, `child1` and `child2`, indicating other members (bodies or other binaries) of the particle set, as well as attributes describing its orbital elements.

The example script has four bodies, `particles[0]` to `particles[3]`, and three binaries. The first binary $\mathcal{A}$ is `particles[4]`, the second binary $\mathcal{B}$ is `particles[5]`, and the outer orbit $\mathcal{C}$ of the $\mathcal{A}-\mathcal{B}$ binary is `particles[6]`. We adopt masses 1.0 $M_\odot$ and 0.8 $M_\odot$ for binary $\mathcal{A}$ with $a_\mathcal{A} = 1.0$au, $e_\mathcal{A} = 0.1$, and $i_\mathcal{A} = 75°$. Binary $\mathcal{B}$ has primary and secondary masses of 1.1 $M_\odot$ and 0.9 $M_\odot$, $a_\mathcal{B} = 1.2$au, $e_\mathcal{B} = 0.1$, and

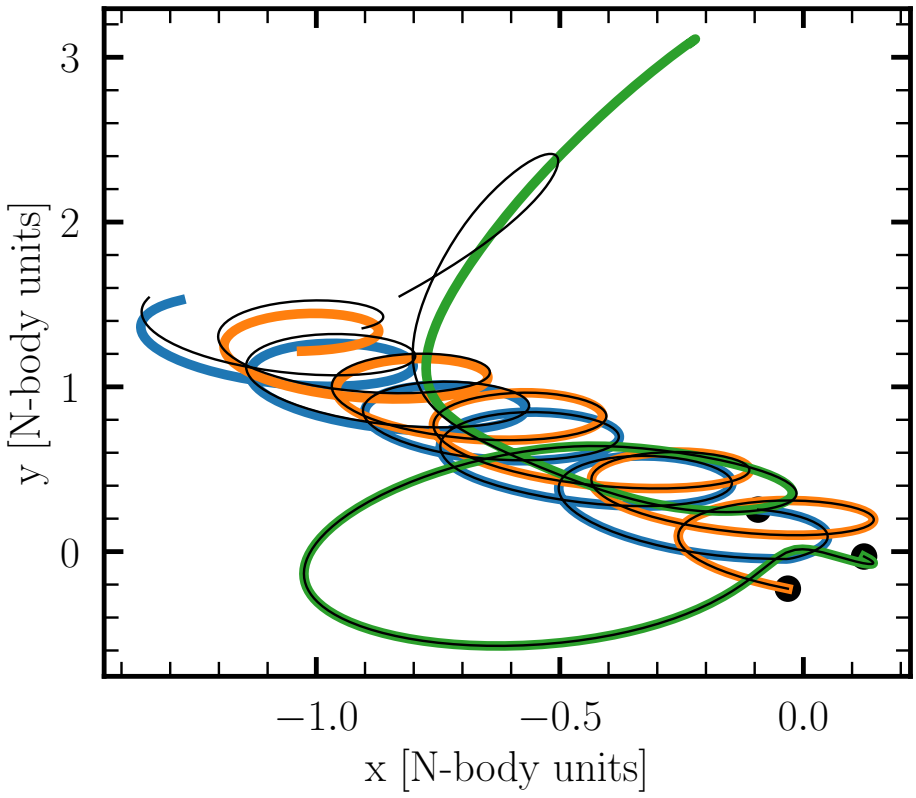


Figure 3.9: The script to generate this figure can be found at `${AMUSE_DIR}/examples/textbook/brutus_example.py`.

$i_{\mathcal{B}} = 80°$. The outer binary, $\mathcal{C}$, has $a_{\mathcal{C}} = 100$au, $e_{\mathcal{C}} = 0.3$.

Running `SecularMultiple` is no different from running any other $N$-body code in AMUSE, using the function `evolve_model()` in the event loop. Figure 3.11 presents the evolution of the orbital separation and eccentricity of the various components of this quadruple system. The top panel shows the semimajor axes (solid curves) and periapsis distances (dotted lines) of the three orbits $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$; the bottom panel shows the inclinations of $\mathcal{A}$ and $\mathcal{B}$ relative to $\mathcal{C}$. High Lidov–Kozai-like eccentricity oscillations occur in the $\mathcal{A}$ and $\mathcal{B}$ orbits, although they are more complex than the equivalent isolated hierarchical triple systems and the maximum eccentricities are considerably higher. (To study the difference due to the fourth body in the system, we can easily reduce the semimajor axis of one of the binaries $\mathcal{A}$ or $\mathcal{B}$ by a factor of 100, effectively reducing it to a point-mass. This quenches the effect of one of them being a binary and recovers the classic Lidov–Kozai effect in the remaining triple system.)

In addition to point-mass secular Newtonian dynamics, other physical processes can also be included in `SecularMultiples`. These include post-Newtonian (PN) effects up to and including order 2.5, and tidal evolution (including misaligned spins). The Mardling & Aarseth (2001) stability criterion for hierarchical triples is also implemented and checked.

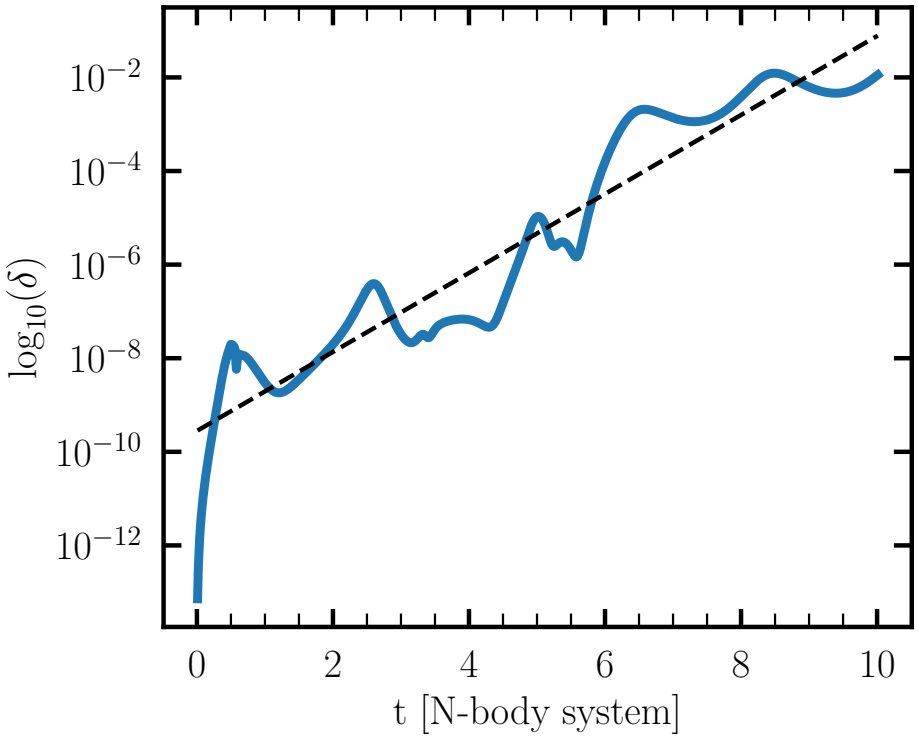


Figure 3.10: A script to generate this figure can be found at `${AMUSE_DIR}/examples/textbook/brutus_example.py`.

### 3.4.4 Merging Galaxies

There are at least three methods whereby gravity is calculated in SPH codes: these include direct summation, the tree algorithm, and self-consistent mean fields using fast Fourier transforms (Hockney & Eastwood, 1988). Direct summation has the advantage of allowing accurate energy conservation, which can be important for problems near a stability limit, for example for a binary star system near its innermost stable circular orbit. These direct summation methods are typically employed on GPUs using the same algorithms as for $N$-body codes (Portegies Zwart *et al.*, 2007).

The most common engine for computing self-gravity in an SPH code is a tree code. Such codes are well-suited to simulating galaxy mergers. In Section 2.1, we discuss how to generate a galaxy model using `Halogen` (Zemp *et al.*, 2008; Zemp, 2014). Here we will use `GalactICs` (Kuijken & Dubinski, 1995), which is designed to set up a self-consistent galaxy model with a disk, bulge, and dark halo. In principle, this code can be used to construct realistic models with both gas and particles for the Milky Way Galaxy, Andromeda, or any other galaxy, but it often requires some delicate tuning to get the proper structure.

We use `GalactICs` to generate two identical disk galaxies with `n_halo` particles in each halo, `n_bulge` particles in each bulge, and `n_disk` particles in each disk. We eventually scale the models to virial equilibrium. The first galaxy is created as follows:

```
def initialize_multiple_system(
    number_of_bodies,
    masses,
    semimajor_axis,
    eccentricity,
    inclination,
    argument_of_pericenter,
    longitude_of_ascending_node,
):
    """
    Initializes a system of multiples
    """

    number_of_binaries = number_of_bodies - 1
    particles = Particles(number_of_bodies + number_of_binaries)
    for index in range(number_of_bodies):
        particle = particles[index]
        particle.mass = masses[index]
        particle.is_binary = False
        particle.radius = 1.0 | units.RSun
        particle.child1 = None
        particle.child2 = None

    for index in range(number_of_binaries):
        particle = particles[index + number_of_bodies]
        particle.is_binary = True
        particle.semimajor_axis = semimajor_axis[index]
        particle.eccentricity = eccentricity[index]
        particle.inclination = inclination[index]
        particle.argument_of_pericenter = argument_of_pericenter[index]
        particle.longitude_of_ascending_node = longitude_of_ascending_node[index]

        # Specify the `2+2' hierarchy:

        if index == 0:
            particle.child1 = particles[0]
            particle.child2 = particles[1]
        elif index == 1:
            particle.child1 = particles[2]
            particle.child2 = particles[3]
        elif index == 2:
            particle.child1 = particles[4]
            particle.child2 = particles[5]
    binaries = particles[particles.is_binary]

    return particles, binaries
```

Listing 3.7: Snippet to setup a "2+2" hierarchical quadruple system composed of two binaries that orbit each other. The full script can be found in `amuse.examples.hierarchical_quadruple`.

```
converter = nbody_system.nbody_to_si(M_galaxy, R_galaxy)
galaxy1 = new_galactics_model(n_halo,
                              converter,
                              do_scale=True,
                              bulge_number_of_particles=n_bulge,
                              disk_number_of_particles=n_disk)
```

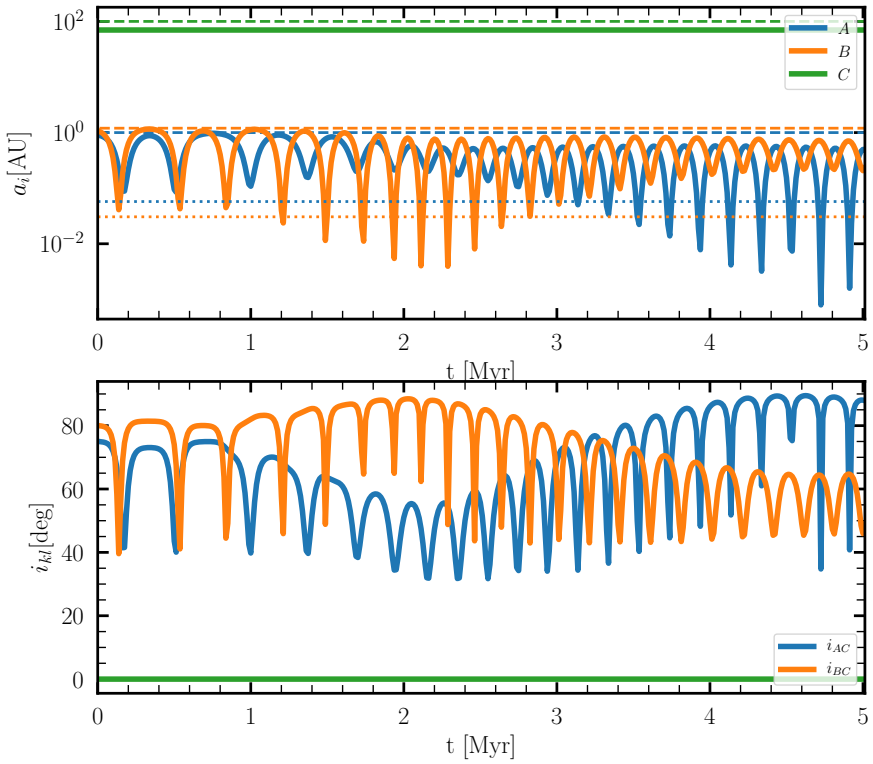


Figure 3.11: Example evolution of a 2 + 2 quadruple system. Top panel: the semimajor axes (dashed lines) and periapsis distances (solid curves) of orbits $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ as a function of time. Bottom panel: the inclinations of $\mathcal{A}$ and $\mathcal{B}$ relative to $\mathcal{C}$. Canonical values expected for Lidov–Kozai oscillations in the three-body test-particle quadrupole-order approximation are shown in the top panel as dotted lines. The dynamics are more complex in this quadruple system, compared to the situation of two uncoupled triple systems. The script that produced this figure is `${AMUSE_DIR}/figures/secularmultiples_quadruple.py`, which takes about half a minute to run.

The second galaxy is constructed by simply making a copy of the first. Both galaxies are placed in an interacting orbit; the hydrodynamics solver is then provided with the particles and integrates the equations of motion without taking hydrodynamics into account.

```
converter = nbody_system.nbody_to_si(1.0e12 | units.MSun, 100 | units.kpc)
dynamics = Gadget2(converter, number_of_workers=4)
dynamics.parameters.epsilon_squared = (100 | units.parsec)**2
set1 = dynamics.particles.add_particles(galaxy1)
set2 = dynamics.particles.add_particles(galaxy2)
dynamics.particles.move_to_center()
```

For plotting purposes, we want to keep track of only the disk and bulge particles. This is accomplished by selecting only those particles added after the halo was generated (the order of creation is halo, bulge/disk). For this reason, we save pointers to the particles in the running code in the local parameters `set1` and `set2` for the first and second galaxy, respectively. Here, we take advantage of the fact that `add_particles` always returns a pointer to the last object added (in this case the bulge/disk particles).

Figure 3.12 presents two snapshots of the interacting galaxies, one at the start of the simulation and the other 200 Myr later when the two galaxies have just passed each

other. The two galaxies are identical copies, but they are rotated using the particle-set attribute `rotate`. For the galaxy on the left in Figure 3.12 (red), this is realized with

```
galaxy1.rotate(0.0, numpy.pi/2, numpy.pi/4)
```

The galaxies are subsequently translated to a new position about 200 kpc apart and given a velocity toward one other of about 10 km $s^{-1}$. We refrain here from a detailed analysis of the resulting merger, but it may be rewarding to simulate the cosmic collision between the Milky Way and the Andromeda galaxy.

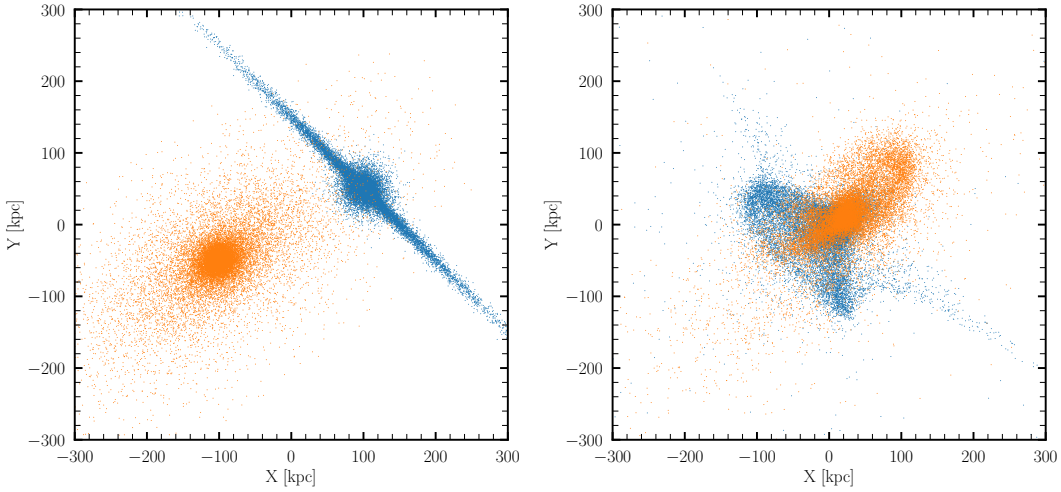


Figure 3.12: Collision between two galaxies, just before they interact (left) and 200 Myr later (right), when both galaxies have been distorted by the interaction. The calculation was performed using `Gadget2` (see Chapter 5 Springel, 2000, 2005) as a gravity solver, without hydrodynamics. Both galaxies are made of 20,000 halo particles, with 10,000 for each the bulge and the disk. In the figure, we only show the bulge and disk particles. The simulation was performed using the script `${AMUSE_DIR}/examples/textbook/merge_two_galaxies.py`. It should take a couple of minutes to run on a laptop.

## 3.5 Choosing the right code

It is not always trivial to choose the right code for your project. There generally is an optimum choice, and one can wonder if there is a wrong choice. Deciding on what code to select is not trivial, and the optimization matrix may be complex. The arguments that tips the balance in favor of one code to another can be based on the physics, precision, accuracy, long term stability, small energy error, angular momentum conservation, performance, et cetera.

The first question to ask is to what extend the gravity in your problem shouldbe coupled to another physical domain. We cannot really recommend on this aspect, as each code can be coupled to other solvers in AMUSE, although there are a few details to consider.

### 3.5.1 Coupling with other physics

Some codes, such as `Gadget2`, are missing the essential function `get_gravity_at_point` in their AMUSEinterface. As a consequence, any bridge operation with `Gadget2` requires you to take care of this coupling. This is not hard, but memory-wise it is not optimal. Another code that one probably should not couple to other codes is `Brutus`. Acquiring an arbitrary precise converged solution to any $N$-body problem is the major advantage of using `Brutus`, but coupling such solution to less precise solutions to the hydrodynamics does not strike us as an effective use of your computer time; although coupling `Brutus` with an analytic background potetial might have it's specific applications.

Overall, any code in AMUSE can be coupled to other codes without much effort, and the best way to explore the effectiveness of coupling gravity solvers is by trying it.

### 3.5.2 Included physics

If one is interested in the dynamics of black holes or other compact objects in a relativistic regime, one probably desires to use a code that incorporates some form of general relativity. Of course, one could incorporate general relativistic effects in any of the N-body codes through a augmented bridge (see Chapter 8). There may be good reasons to add general relativity through a bridge, but this would bring a whole range of other aspects to be considered to the calculation. The easier solution is to opt for a relativistic code. At the moment AMUSE has three codes that incorporate some form of generalitivsitc effects, these include `AarzethZare`, `Hermite_GRX`, `Brutus_GRX`, and maybe one could consider `Mercury`.

The latter code, `Mercury`, incorporates 1-st order Taylor expanion of the Einstein Infeld Hoffman equation in the two-body limit. This means, in practice, that the absidal motion of the inner most planets, such as Mercury in the Solar system, can be taken into account.

Another code that incorporates 2-boyd approximations to the Einstein Infeld Hoffman equation is `AarzethZare`. This code is also regularized for few-body democratic encounters. It incorporates general relativisic equations of motion up to 2.5th order, but these do not incorporate the cross terms, which are important for multi-body ($N \geq 3$) close encounters between compact objects.

When multi-body dynamics of compact objects are considered, one should probably opt for `Hermite_GRX` or `Brutus_GRX`. The latter is an expansion of `Brutus` and meant for acquiring converged solutions. The workhorse for this type of calculations is probably `Hermite_GRX`, which is reasonably fast, sufficiently precise for production simultions and incorporates the cross terms for the relativistic equations of motion to first order, and the pair wise equations to 3.5th order.

`Ph4 Hermite Brutus Huayno`

`BHTree Gadget2 Bonsai`

`Mercury AarzethZare Hermite_GRX Brutus_GRX`

## 3.6 Validation

Quickly prototyping a small script that runs a large scientific simulation is easy with AMUSE, but how do you guarantee that everything works correctly? You can't, and we can't either. However, we can at least guarantee that the coupling of the code is done correctly and that the units are properly converted and transferred.

Some codes are easier to validate than others. Direct gravitational solvers have to conserve energy, at least if they are run as isolated, standalone modules. Checking for energy conservation is therefore always a good thing to do. For stellar evolution, there is no conserved energy, but you would surely like the total mass of the star plus the total mass lost in a wind and a supernova to be conserved. Each methodology has its own conserved quantities. As a researcher and user of any scientific code, you should always explicitly check them.

Time-stepping in any given code is generally managed internally, but as a user, you can in principle change the time steps as you like. However, in addition to the fact that we do not recommend fiddling with the community source code, we also do not recommend overriding the safety settings for things like time step control. Some of the community modules are very sensitive to such changes and may simply fail, but others will always produce some output. These latter codes are the most dangerous: "garbage in, garbage out." If a code receives completely wrong information, in the form of nonsensical initial conditions or parameter settings, then the best thing that can happen is that it crashes. The worst outcome is that the code will continue to run, producing reasonable-looking nonsense.

### 3.6.1 Error Propagation and Validation

The solar system is about 4.5 Gyr old, which means that Earth has made more than four billion orbits around the Sun. During each integration step in a simulation, the program makes tiny mistakes (such as round-off errors at about the 15th digit, or larger errors due to algorithmic limitations), which slowly accumulate with time. The accumulated energy error may eventually grow to exceed the total energy of the system, at which point we would be forced to conclude that the integration has failed to represent the physical system we are trying to model.

For lack of a satisfactory alternative, the energy error is often used as the sole diagnostic for quantifying the quality of an $N$-body integration—but beware: a gravitational system (probably) responds exponentially to small errors, drifting away from whatever the true solution is and probably rendering the numerically determined position and velocity of any particle meaningless. However, the error in the energy may grow randomly (according to Brouwers' law; see Section 3.2.3) or, more likely, systematically and slowly. Therefore, the energy error is a rather poor indicator of the quality of an $N$-body integration (partly for reasons discussed in Section 3.1.4.1), but so long as it does not grow beyond a certain fraction of the total energy, the result may be sufficiently accurate for scientific interpretation.

The boundary above which we consider the energy error acceptable is quite arbitrary and context-dependent. If no number is quoted, you should, as a scientist, doubt the

quality of the calculation (even though the animations and figures may look great). As a rule of thumb, for direct $N$-body simulations, the cumulative energy error over the duration of the calculation should remain below 1% of the total energy of the entire system, and below $10^{-4}$ of the total energy of the system over an interval of one dynamical time. Many researchers would regard these limits as far too high.

#### 3.6.1.1 Error Behavior and Analysis

The errors in $N$-body integrations can be measured objectively by checking conserved quantities, such as energy and angular momentum. In Listing 3.2, we demonstrated how energy conservation can be checked at runtime. This provides a useful first-order diagnostic, although for the solar system (for example), it informs you mainly about the integration errors for the most massive bodies. It is far less informative about errors in the integration of minor bodies.

The error behavior of $N$-body codes depend both on the algorithm used to calculate the forces and how the equations of motion are integrated. Figure 3.13 shows the relative energy error for three $N$-body solvers—a (fourth-order) direct-summation Hermite solver, a similar high-order solver but including the Einstein-Infeld-Hoffman equations of motion for general relativity, and a (second-order) Barnes–Hut tree code with a Leapfrog solver. The greneral relativistic run was performed with a speed of light one hundred times higher than the maximum velocity of any of the particles, which is rather arbitrary, but demonstrates the point that the scaling is identical to the Newtonian 4th order integrator.

Each solver behave somewhat different, but this is not unexpected. The direct $N$-body integration can reach very small energy errors, below $10^{-13}$. For smaller values of the time-step parameter $\eta \lesssim 0.005$, the error starts increasing again. From this point, numerical round-off starts to drive the energy error in the calculation. For a time step parameter of unity (and larger) the error suddenly increases to a value of $\mathcal{O}(1)$. Once the time step parameter becomes too large (apparently around order unity), the integrator is unable to follow individual tranjectories accurately. The integrator does not break completely, but fails to acquire convergence in the solution. This is reflected in the large energy error. This is one of the reasons why it is important to always check the energy conservation in runs.

The relativisitc integrator Hermite_GRX shows a similar behavior as the Newtonian integrator, but it is much more expensive to calculate (the force in general relativity also depends on the velocity and therefore scales $\propto N^3$). The numerical round-off in the relativistic equations of motion starts to become noticeable at a much larger value of $\eta \lesssim 0.1$ compared to the Newtonian calculation, becasue the number of operations per formce calculation is orders of magntiude larger. It is therefore not very usefull to run relativistic calculations with extremely small values of $\eta$ (except, of course, when one runs with an arbitrary precise integrator, such as Brutus).

With the tree code, the error saturates because the error at short time steps is dominated by the tree force evaluation algorithm, not by the integration scheme. However, this camparison is unfair because BHTree this paremter is defined differently. In BHTree the time step is either an absulute fraction of the local crossing time (when

running in dimension-less N-body units) or it is an absolute value in units of (million) years. In this case, the calculations are performed using dimension-less N-body unuts, and $\eta$ for BHTree should be interpreted as fractions of the dynamical crossing time. The other parameter that can be used to tune the precision performance of a tree code such as BHTree is the opening angle $\theta$. In this example we used the default value of $\theta = 0.75$.

The only way to achieve accuracy better than $dE/E \simeq 10^{-14}$ is by adopting a longer mantissa in the calculations. This can be achieved with the `Brutus` integrator in `AMUSE`, in which case one can reach energy errors as low as one desires, so long as one is suffiicently patient. In the relativistic version of Brutus, called Brutus_GRX the Einstein-Infeld-Hoffman equations of motion are solved in similar fashion as in Hermite_GRX, except that it remains a $2^{\rm nd}$ order scheme, as in Brutus. Both Brutus and Brutus_GRX require considerable patience of the user, because reaching convergence is extraordinarily expensive.

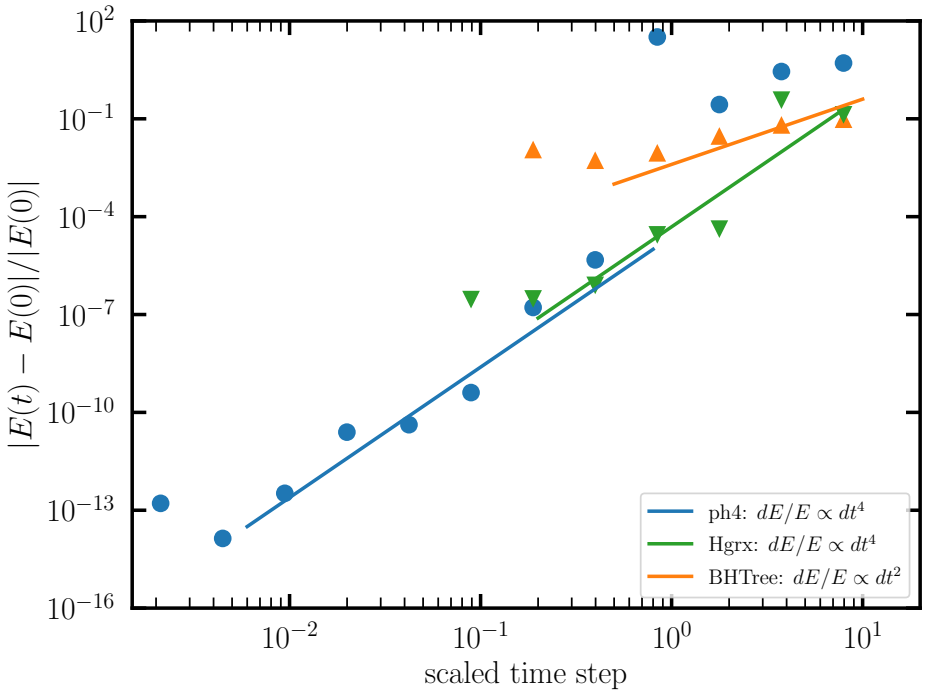


Figure 3.13: Energy error of the integration for one dynamical time of $N = 1000$ equal-mass particles initially distributed in a Plummer sphere. The integration was carried out using the direct $N$-body integrator `ph4` (blue bullets), Hermite_grx (green triangles), and with the `BHTree` tree code (orange triangles). The three solid lines show the global trends in energy conservation of the $N$-body codes (these are not fits). The Hermite_grx run was performed for only 0.1 N-body time units, to reduce the computational cost. The script to generate this figure can be found in `amuse.examples.gravity.run_nbody_precision`. The ph4 and BHTree runs take a few seconds each, but the runs with Hermite_grx are considerably more expensive, and takes about an hour to run on a laptop.

## 3.7 Assignments

### 3.7.1 Orbital Trajectories

Take the initial conditions for the solar system (Table 2.1) and integrate the planets' orbits for 100 yr with the integrator of your choice (Table 3.1). Explain why you have selected this integrator, and discuss the effect of using another integrator.

1. Draw graphs of the energy error as a function of time, and of the change in the planet's semimajor axis, $da/a$, as a function of time.

2. Draw a graph of the evolution of the semimajor axes and eccentricities of Venus, Earth, and Mars, and compare them with the results in Figure 3.7.

3. Run the same initial conditions with two other integrators, and explain the differences in the results.

   Compare the evolution of the semimajor axes and eccentricities for Venus, Earth, and Mars with those presented in Figure 3.7. In what quantity are the differences most pronounced?

### 3.7.2 Vostok

In Section 3.4.1.3 we discussed the variation in the eccentricity of Earth's orbit over the last 450,000 yr. Although the variations in the average temperature measured at the Vostok station are not entirely attributable to the apparent cycle in Earth's eccentricity, it is evident that some periodicity appears in Earth's orbit.

These variations are probably due to the influence of only a few (maybe only one) other planet. It is unlikely that Mercury or Neptune drives this cycle, but the $N$-body code we used to create Figure 3.8 can help us find out which planet is responsible for the variations in Earth's orbit.

Perform this analysis and find the planet that dominates the evolution of Earth's orbit around the Sun.

### 3.7.3 Dynamical Binary Formation

(a) In cluster dynamics, it is common to define an energy scale, called $kT$ (as in thermodynamics), by $K = 3/2NkT$, where $N$ is the number of stars and $K$ is the total kinetic energy of the system. This turns out also to be an interesting energy threshold for binary systems (see Section 3.1.3.2. Run an $N$-body simulation with $N = 100$ stars using a code of your choice (see Table 3.1), until the appearance of a binary with a binding energy exceeding 100 $kT$. You can take the code Listing 3.2 as a basis for the $N$-body simulation. The simplest way to do this is to run in steps of (say) one dynamical time, and compute the relative binding energies of all pairs of stars at the end of each step. This is fairly inefficient, but infrequent and fast enough with 100 stars that it should be doable in Python.

Be careful here, as several of the *N*-body codes in AMUSE use softened potentials and are unable to reach such hard binaries, while others will have great difficulty integrating hard binaries.

Use a Plummer distribution as the initial density profile and take all the masses to be the same.

(b) Perform the run with a different *N*-body code, using exactly the same initial realization of the initial conditions.

(c) Is the moment at which the first 100 $kT$ binary appears the same, or does it depend on the code?

(d) What changes when a mass function is introduced? Adopt a Salpeter mass function (see Equation (2.1)) with $\alpha = -2.35$ (Salpeter, 1955). Remember to rescale the system to virial equilibrium (see Section 3.3.1) after setting the stellar masses.

Explain why the system should be rescaled to virial equilibrium.

In an *N*-body integration performed in dimensionless *N*-body units, the only parameter to specify for the mass function is the range over which the masses should be varied (the total mass is normalized), assuming that the power-law slope remains the same. How does the moment of the formation of the first hard binary depend on the range of masses?

### 3.7.4 $L_1$ Lagrangian Point

Lagrangian points in a self-gravitating system of two bodies are locations at which there is either a peak or a saddle point in the effective potential surface (de Lagrange, 1772) in the frame rotating with the bodies. An *N*-body code can easily be adapted to draw such an equipotential surface. Figure 3.14 presents an example for the case of the Sun and a 0.2 $M_\odot$ mass companion at a distance of 1 au. The script to generate these equipotential surfaces can easily be adapted to any binary system, including those with nonzero eccentricity.

Interplanetary travel is expensive in terms of time and fuel. Moving through Lagrange points provides an efficient way to travel from one celestial body to another (although they can also lead to objects becoming trapped in a local orbit, as is the case with Jupiter's Trojan satellites). In Figure 3.14, the most efficient way to travel from the star to the left to the one to the right is through $L_1$. This trajectory can be calculated most easily by starting at $L_1$ and allowing a test mass to "fall" into the potential well to the left or the right.

(a) Calculate the most energy-efficient orbit from Earth to the Moon in an isolated system, but with Earth and Moon on the proper relative orbit. Plot this trajectory in Cartesian coordinates.

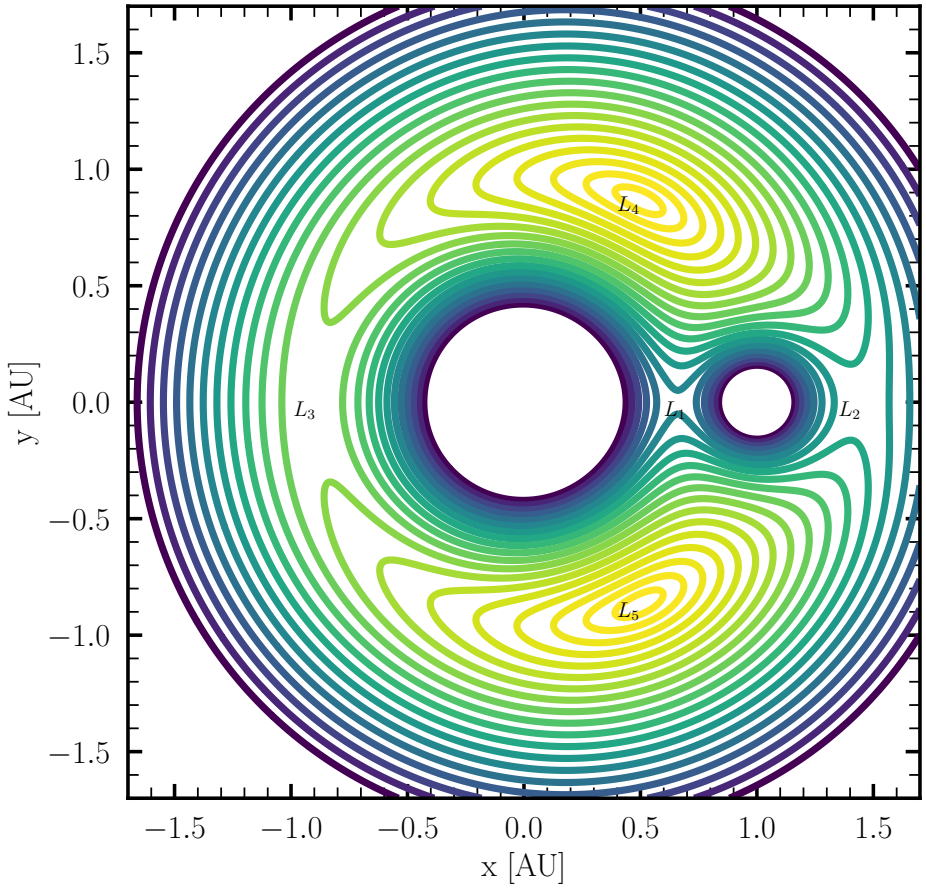


Figure 3.14: Equipotential surfaces for a two-body system. In this example, we adopted a primary mass of 1 $M_\odot$ with a 0.2 $M_\odot$ companion in a circular orbit at a distance of 1 au (to the right). The classical five Lagrange points are indicated by $L_1 - L_5$. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/lagrange_points.py`.

(b) Repeat the calculation for Earth and the Moon in orbit around the Sun, with the planet Jupiter added to further perturb the system. Are the two trajectories the same?

*Hint:* Start by initializing a particle on the first Lagrangian point of the Earth–Moon system and give it a small velocity toward Earth. Repeat with a small velocity in the direction of the Moon.

### 3.7.5 Virial Equilibrium

(a) Calculate the evolution of the virial ratio for an $N$-body system with a mass function between 0.1 and 100 $M_\odot$ using the codes `ph4`, `Huayno`, and `BHTree`, starting with an initial virial ratio of $Q = 0.2$ (Section 3.3.1), and continue until the system virializes. Perform the calculation with $N = 100$, increasing by factors of 2 up to $N = 1600$, and make a plot of the first moment in time for which $Q = 0.5$ as a function of $N$ for each of the three codes.

The virial ratio will oscillate due to random close encounters, but the virialization time scale can be estimated by studying the way in which the oscillations damp out and the viral ratio approaches its equilibrium value of 0.5.

(b) Figure 3.15 presents an example $q(t)$ for three $N$-body codes, using 1000 stars of equal mass.

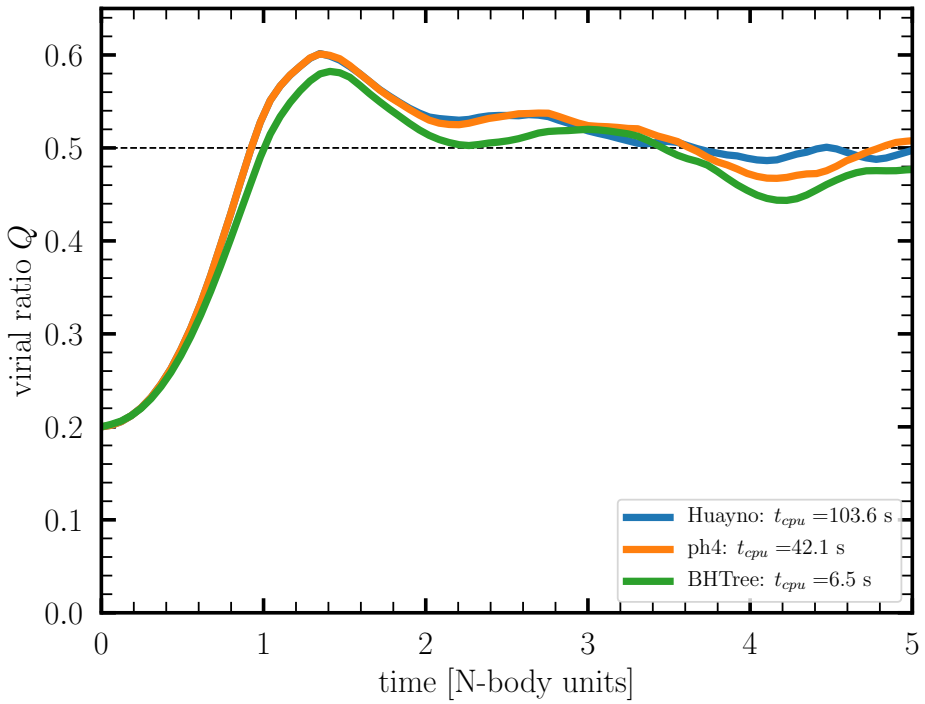


Figure 3.15: Virial ratio $Q$ as a function of time for 1000 equal mass particles initially distributed in a Plummer sphere with $Q = 0.2$. The integrations were performed using `ph4` (blue line), `Huayno` (red), and `BHTree` (yellow). The script that produced this figure is in `${AMUSE_DIR}/examples/textbook/gravity_to_virial.py`. It should run in a couple of minutes on a laptop.

Answer the following questions:

(i) Why does the system with equal-mass stars virialize on a time scale of about an $N$-body time unit?

(ii) What is the effect of introducing a mass function on the virialization time scale?

(iii) Why does the way in which the system virializes depend on the code?

(iv) What is the effect of increasing the number of stars on the virialization time scale?

(v) Why do the two direct $N$-body systems remain slightly above virial after about 1 $N$-body time unit?

# Bibliography


Aarseth, S. A. 2003. *Gravitational N-body simulations*. Cambdridge University press, 2003.

Aarseth, S. J. 1963. Dynamical evolution of clusters of galaxies, I. *MNRAS* , **126**(Jan.), 223.

Aarseth, S. J. 1985 (Jan.). Direct methods for N-body simulations. *Pages 377–418 of: Multiple time scales*.

Abbott, B. P., Abbott, R., Abbott, T. D., Abernathy, M. R., Acernese, F., Ackley, K., Adams, C., Adams, T., Addesso, P., Adhikari, R. X., & et al. 2016. Properties of the Binary Black Hole Merger GW150914. *Phys. Rev. Lett.* , **116**(24), 241102.

Ambrosiano, J., Greengard, L., & Rokhlin, V. 1988. The fast mutipole method for gridless particle simulation. *Computer Physics Communications*, **48**(1), 117–125.

Anders, P., Baumgardt, H., Gaburov, E., & Portegies Zwart, S. 2012. How well do STARLAB and NBODY compare? II. Hardware and accuracy. *MNRAS* , **421**(4), 3557–3569.

Antonov, V. A. 1962. *Solution of the problem of stability of stellar system Emden's density law and the spherical distribution of velocities*.

Babuška, Ivo. 1969. Numerical Stability in Mathematical Analysis. *Pages 11–23 of: Proc. IFIP Congress*. Information Processing 68. ,.

Barnes, Josh, & Hut, Piet. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nat* , **324**(6096), 446–449.

Bédorf, Jeroen, Gaburov, Evghenii, & Portegies Zwart, Simon. 2012. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, **231**(7), 2825–2839.

Bédorf, Jeroen, Gaburov, Evghenii, Fujii, Michiko S., Nitadori, Keigo, Ishiyama, Tomoaki, & Portegies Zwart, Simon. 2014. 24.77 Pflops on a Gravitational Tree-code to Simulate the Milky Way Galaxy with 18600 GPUs. *Pages 54–65 of: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. Piscataway, NJ, USA: IEEE Press.

Bettwieser, E., & Sugimoto, D. 1984. Post-collapse evolution and gravothermal oscillation of globular clusters. *MNRAS* , **208**(June), 493–509.

Binney, James, & Tremaine, Scott. 2008. *Galactic Dynamics: Second Edition*.

Boekholt, Tjarda, & Portegies Zwart, Simon. 2015. On the reliability of N-body simulations. *Computational Astrophysics and Cosmology*, **2**(Mar.), 2.

Brouwer, Dirk. 1937. On the accumulation of errors in numerical integration. *AJ* , **46**(Oct.), 149–153.

Capuzzo-Dolcetta, R., Spera, M., & Punzo, D. 2013. A fully parallel, high precision, N-body code running on hybrid computing platforms. *Journal of Computational Physics*, **236**(Mar.), 580–593.

Chambers, J. E. 1999. A hybrid symplectic integrator that permits close encounters between massive bodies. *MNRAS* , **304**(4), 793–799.

Chambers, J. E., & Migliorini, F. 1997 (July). Mercury - A New Software Package for Orbital Integrations. *Page 27.06 of: AAS/Division for Planetary Sciences Meeting Abstracts #29*. AAS/Division for Planetary Sciences Meeting Abstracts, vol. 29.

Chandrasekhar, S. 1943. Dynamical Friction. I. General Considerations: the Coefficient of Dynamical Friction. *ApJ* , **97**(Mar.), 255.

Chernoff, David F., & Weinberg, Martin D. 1990. Evolution of Globular Clusters in the Galaxy. *ApJ* , **351**(Mar.), 121.

Cohn, H. 1980. Late core collapse in star clusters and the gravothermal instability. *ApJ* , **242**(Dec.), 765–771.

de Lagrange, J-L. 1772. *Chapitre II: Essai sur le Problème des Trois Corps.*

Dehnen, W. 2014. A fast multipole method for stellar dynamics. *Computational Astrophysics and Cosmology*, **1**(Dec.), 1.

Duncan, Martin J., Levison, Harold F., & Lee, Man Hoi. 1998. A Multiple Time Step Symplectic Algorithm for Integrating Close Encounters. *AJ* , **116**(4), 2067–2077.

Einstein, A. 1916. Die Grundlage der allgemeinen Relativitätstheorie. *Annalen der Physik*, **354**(7), 769–822.

Fellhauer, M., & Lin, D. N. C. 2007. The influence of mass-loss from a star cluster on its dynamical friction - I. Clusters without internal evolution. *MNRAS* , **375**(2), 604–614.

Fukushige, Toshiyuki, & Heggie, Douglas C. 1995. Pre-collapse evolution of galactic globular clusters. *MNRAS* , **276**(1), 206–218.

Gaburov, Evghenii, Bédorf, Jeroen, & Portegies Zwart, Simon. 2010 (Oct.). *OCTGRAV: Sparse Octree Gravitational N-body Code on Graphics Processing Units.* Astrophysics Source Code Library, record ascl:1010.048.

Giersz, M., & Heggie, D. C. 1994. Statistics of N-Body Simulations - Part One - Equal Masses Before Core Collapse. *MNRAS* , **268**(May), 257.

Giersz, Mirek, & Heggie, Douglas C. 1996. Statistics of N-body simulations - III. Unequal masses. *MNRAS* , **279**(3), 1037–1056.

Gildor, Hezi, & Tziperman, Eli. 2000. Sea ice as the glacial cycles' climate switch: Role of seasonal and orbital forcing. *Paleoceanography*, **15**(6), 605–615.

Gnedin, Nickolay Y., Glover, Simon G. O., Klessen, Ralf S., & Springel, Volker. 2015. Star Formation in Galaxy Evolution: Connecting Numerical Models to Reality. *Page 1 of: Star Formation in Galaxy Evolution: Connecting Numerical Models to Reality. Edited by N.Y. Gnedin et al. Springer.*

Greengard, L., & Rokhlin, V. 1987. A fast algorithm for particle simulations. *Journal of Computational Physics*, **73**(2), 325–348.

Hamers, Adrian S., & Portegies Zwart, Simon F. 2016. Secular dynamics of hierarchical multiple systems composed of nested binaries, with an arbitrary number of bodies and arbitrary hierarchical structure. First applications to multiplanet and multistar systems. *MNRAS* , **459**(3), 2827–2874.

Harfst, Stefan, Gualandris, Alessia, Merritt, David, Spurzem, Rainer, Portegies Zwart, Simon, & Berczik, Peter. 2007. Performance analysis of direct N-body algorithms on special-purpose supercomputers. *New Astron.* , **12**(5), 357–377.

Heggie, D. C. 1975. Binary evolution in stellar dynamics. *MNRAS* , **173**(Dec.), 729–787.

Heggie, D. C., & Mathieu, R. D. 1986. Standardised Units and Time Scales. *Page 233 of:* Hut, Piet, & McMillan, Stephen L. W. (eds), *The Use of Supercomputers in Stellar Dynamics*, vol. 267.

Heggie, Douglas, & Hut, Piet. 2003. *The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics*.

Hénon, M. 1972 (Jan.). The Monte Carlo Method. *Page 406 of:* Lecar, Myron (ed), *IAU Colloq. 10: Gravitational N-Body Problem*. Astrophysics and Space Science Library, vol. 31.

Hénon, M. 1975 (Jan.). Two Recent Developments Concerning the Monte Carlo Method. *Page 133 of:* Hayli, Avram (ed), *Dynamics of the Solar Systems*, vol. 69.

Hills, J. G. 1975. Encounters between binary and single stars and their effect on the dynamical evolution of stellar systems. *AJ* , **80**(Oct.), 809–825.

Hockney, R.W., & Eastwood, J.W. 1988. *Computer Simulation Using Particles*. Adan Hilger Ltd., Bristol, UK, 540 pp.

Hulse, R. A., & Taylor, J. H. 1975. Discovery of a pulsar in a binary system. *ApJL* , **195**(Jan.), L51–L53.

Hut, Piet, Makino, Jun, & McMillan, Steve. 1995. Building a Better Leapfrog. *ApJL* , **443**(Apr.), L93.

Inagaki, S., & Saslaw, W. C. 1985. Equipartition in multicomponent gravitational systems. *ApJ* , **292**(May), 339–347.

Iwasawa, Masaki, Portegies Zwart, Simon, & Makino, Junichiro. 2015. GPU-enabled particle-particle particle-tree scheme for simulating dense stellar cluster system. *Computational Astrophysics and Cosmology*, **2**(July), 6.

Jänes, Jürgen, Pelupessy, Inti, & Portegies Zwart, Simon. 2014. A connected component-based method for efficiently integrating multi-scale N-body systems. *A&A* , **570**(Oct.), A20.

Jouzel, J., Genthon, C., Lorius, C., Petit, J. R., & Barkov, N. I. 1987. Vostok ice core - A continuous isotope temperature record over the last climatic cycle (160,000 years). *Nat* , **329**(Oct.), 403–408.

Kahan, W. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. *Commun. ACM*, **8**(1), 40.

King, Ivan R. 1966. The structure of star clusters. III. Some simple dynamical models. *AJ* , **71**(Feb.), 64.

Kozai, Yoshihide. 1962. Secular perturbations of asteroids with high inclination and eccentricity. *AJ* , **67**(Nov.), 591–598.

Kuijken, K., & Dubinski, J. 1995. Nearly Self-Consistent Disc / Bulge / Halo Models for Galaxies. *MNRAS* , **277**(Dec.), 1341.

Laskar, Jacques, & Robutel, Philippe. 2001. High order symplectic integrators for perturbed Hamiltonian systems. *Celestial Mechanics and Dynamical Astronomy*, **80**(1), 39–62.

Lee, Hyung Mok, & Ostriker, Jeremiah P. 1987. The Evolution and Final Disintegration of Spherical Stellar Systems in a Steady Galactic Tidal Field. *ApJ* , **322**(Nov.), 123.

Lidov, M. L. 1962. The evolution of orbits of artificial satellites of planets under the action of gravitational perturbations of external bodies. Planet. Space Sci., **9**(10), 719–759.

Liu, Lei, Wu, Xin, Huang, Guoqing, & Liu, Fuyao. 2016. Higher order explicit symmetric integrators for inseparable forms of coordinates and momenta. *MNRAS* ,

**459**(2), 1968–1976.

Makino, Junichiro. 1991. Optimal Order and Time-Step Criterion for Aarseth-Type N-Body Integrators. *ApJ* , **369**(Mar.), 200.

Makino, Junichiro. 1996. Postcollapse Evolution of Globular Clusters. *ApJ* , **471**(Nov.), 796.

Makino, Junichiro, & Aarseth, Sverre J. 1992. On a Hermite Integrator with Ahmad-Cohen Scheme for Gravitational Many-Body Problems. *Publ. Astr. Soc. Japan* , **44**(Apr.), 141–151.

Mardling, Rosemary A., & Aarseth, Sverre J. 2001. Tidal interactions in star cluster simulations. *MNRAS* , **321**(3), 398–420.

McMillan, Stephen L. W., & Aarseth, Sverre J. 1993. An O(N N) Integration Scheme for Collisional Stellar Systems. *ApJ* , **414**(Sept.), 200.

Mikkola, S. 1983. Encounters of binaries. I - Equal energies. *MNRAS* , **203**(June), 1107–1121.

Mikkola, Seppo, & Merritt, David. 2008. Implementing Few-Body Algorithmic Regularization with Post-Newtonian Terms. *AJ* , **135**(6), 2398–2405.

Milankovitch, Milutin. 1941. Zavod za Udzbenike i Nastavna Sredstva: Kanon der Erdbestrahlung und seine Andwendung auf das Eiszeiten-problem (English Translation, 1998, Canon of Insolation and the Ice Age Problem. With introduction ISBN 86-17-06619-9).

Milgrom, M. 1983. A modification of the Newtonian dynamics as a possible alternative to the hidden mass hypothesis. *ApJ* , **270**(July), 365–370.

Montgomery, R. 1998. The N -body problem, the braid group, and action-minimizing periodic solutions. *Nonlinearity*, **11**(2), 363.

Newton, I. 1687. *Philosophiae Naturalis Principia Mathematica.* Vol. 1.

Nitadori, Keigo, & Makino, Junichiro. 2008. Sixth- and eighth-order Hermite integrator for N-body simulations. *New Astron.* , **13**(7), 498–507.

Oshino, Shoichi, Funato, Yoko, & Makino, Junichiro. 2011. Particle-Particle Particle-Tree: A Direct-Tree Hybrid Scheme for Collisional N-Body Simulations. *Publ. Astr. Soc. Japan* , **63**(Aug.), 881.

Peters, P. C. 1964. Gravitational Radiation and the Motion of Two Point Masses. *Physical Review*, **136**(4B), 1224–1232.

Petit, J. R., Jouzel, J., Raynaud, D., Barkov, N. I., Barnola, J. M., Basile, I., Bender, M., Chappellaz, J., Davis, M., Delaygue, G., Delmotte, M., Kotlyakov, V. M., Legrand, M., Lipenkov, V. Y., Lorius, C., Pépin, L., Ritz, C., Saltzman, E., & Stievenard, M. 1999. Climate and atmospheric history of the past 420,000 years from the Vostok ice core, Antarctica. *Nat* , **399**(6735), 429–436.

Plummer, H. C. 1911. On the problem of distribution in globular star clusters. *MNRAS* , **71**(Mar.), 460–470.

Portegies Zwart, Simon F., & Boekholt, Tjarda C. N. 2018. Numerical verification of the microscopic time reversibility of Newton's equations of motion: Fighting exponential divergence. *Communications in Nonlinear Science and Numerical Simulations*, **61**(Aug.), 160–166.

Portegies Zwart, Simon F., & McMillan, Stephen L. W. 2002. The Runaway Growth of Intermediate-Mass Black Holes in Dense Star Clusters. *ApJ* , **576**(2), 899–907.

Portegies Zwart, Simon F., Belleman, Robert G., & Geldof, Peter M. 2007. High-performance direct gravitational N-body simulations on graphics processing units. *New Astron.* , **12**(8), 641–650.

Rauch, Kevin P., & Tremaine, Scott. 1996. Resonant relaxation in stellar systems. *New Astron.* , **1**(2), 149–170.

Rein, H., & Liu, S. F. 2012. REBOUND: an open-source multi-purpose N-body code for collisional dynamics. *A&A* , **537**(Jan.), A128.

Rein, Hanno, & Liu, Shang-Fei. 2011 (Oct.). *REBOUND: Multi-purpose N-body code for collisional dynamics*. Astrophysics Source Code Library, record ascl:1110.016.

Rein, Hanno, & Spiegel, David S. 2015. IAS15: a fast, adaptive, high-order integrator for gravitational dynamics, accurate to machine precision over a billion orbits. *MNRAS* , **446**(2), 1424–1437.

Rodriguez, Carl L., Morscher, Meagan, Wang, Long, Chatterjee, Sourav, Rasio, Frederic A., & Spurzem, Rainer. 2016. Million-body star cluster simulations: comparisons between Monte Carlo and direct N-body. *MNRAS* , **463**(2), 2109–2118.

Saha, Prasenjit, & Tremaine, Scott. 1992. Symplectic Integrators for Solar System Dynamics. *AJ* , **104**(Oct.), 1633.

Salpeter, Edwin E. 1955. The Luminosity Function and Stellar Evolution. *ApJ* , **121**(Jan.), 161.

Spera, M., Capuzzo Dolcetta, R., & Punzo, D. 2012 (July). *HiGPUs: Hermite's N-body integrator running on Graphic Processing Units*. Astrophysics Source Code Library, record ascl:1207.002.

Spinnato, Piero F., Fellhauer, Michael, & Portegies Zwart, Simon F. 2003. The efficiency of the spiral-in of a black hole to the Galactic Centre. *MNRAS* , **344**(1), 22–32.

Spitzer, Lyman, Jr. 1969. Equipartition and the Formation of Compact Nuclei in Spherical Stellar Systems. *ApJL* , **158**(Dec.), L139.

Spitzer, Lyman. 1987. *Dynamical evolution of globular clusters*.

Springel, Volker. 2000 (Mar.). *GADGET-2: A Code for Cosmological Simulations of Structure Formation*. Astrophysics Source Code Library, record ascl:0003.001.

Springel, Volker. 2005. The cosmological simulation code GADGET-2. *MNRAS* , **364**(4), 1105–1134.

Takahashi, Koji, & Portegies Zwart, Simon F. 1998. The Disruption of Globular Star Clusters in the Galaxy: A Comparative Analysis between Fokker-Planck and N-Body Models. *ApJL* , **503**(1), L49–L52.

Tremaine, Scott. 2015. The Statistical Mechanics of Planet Orbits. *The Astrophysical Journal*, **807**(2), 157.

Trenti, Michele, & van der Marel, Roeland. 2013. No energy equipartition in globular clusters. *MNRAS* , **435**(4), 3272–3282.

Tricarico, Pasquale. 2012 (Apr.). *ORSA: Orbit Reconstruction, Simulation and Analysis*. Astrophysics Source Code Library, record ascl:1204.013.

Verlet, Loup. 1967. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, **159**(Jul), 98–103.

Verlinde, Erik. 2017. Emergent Gravity and the Dark Universe. *SciPost Physics*, **2**(3), 016.

Wang, Long, Spurzem, Rainer, Aarseth, Sverre, Nitadori, Keigo, Berczik, Peter, Kouwenhoven, M. B. N., & Naab, Thorsten. 2015. NBODY6++GPU: ready for the gravitational million-body problem. *MNRAS* , **450**(4), 4070–4080.

Whitehead, Alfred J., McMillan, Stephen L. W., Vesperini, Enrico, & Portegies Zwart, Simon. 2013. Simulating Star Clusters with the AMUSE Software Framework. I. Dependence of Cluster Lifetimes on Model Assumptions and Cluster Dissolution Modes. *ApJ* , **778**(2), 118.

Wisdom, Jack, & Holman, Matthew. 1991. Symplectic maps for the N-body problem. *AJ* , **102**(Oct.), 1528–1538.

Yoshida, Haruo. 1990. Construction of higher order symplectic integrators. *Physics Letters A*, **150**(5-7), 262–268.

Yoshida, Haruo. 1993. Recent Progress in the Theory and Application of Symplectic Integrators. *Celestial Mechanics and Dynamical Astronomy*, **56**(1-2), 27–43.

Zemp, Marcel. 2014 (July). *Halogen: Multimass spherical structure models for N-body simulations*. Astrophysics Source Code Library, record ascl:1407.020.

Zemp, Marcel, Moore, Ben, Stadel, Joachim, Carollo, C. Marcella, & Madau, Piero. 2008. Multimass spherical structure models for N-body simulations. *MNRAS* , **386**(3), 1543–1556.