IOP Astronomy

Astrophysical Recipes

The art of AMUSE

Simon Portegies Zwart and Steve McMillan

Chapter 2

Gravitational Dynamics

Oh my God!—it's 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.

2.1 In a Nutshell

2.1.1 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

$$\mathbf{a}_i \equiv \ddot{\mathbf{r}}_i = G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3}. \tag{2.1}$$

Here, G is the gravitational constant, and m_j and $\mathbf{r}_j$ are the mass and position of particle j. Equation (2.1) is a set of coupled, second-order, nonlinear, singular ordinary differential equations. Its solutions exhibit a bewildering variety of

doi:10.1088/978-0-7503-1320-9ch2

complex, often chaotic motions, and many books have been written about how to numerically solve it (Aarseth 1985, pp. 377–418; Aarseth 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 are applicable on all scales considered in this book.

2.1.2 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*.

2.1.2.1 The Dynamical Time Scale

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

$$t_{\rm dyn} = R/v, \tag{2.2}$$

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)

$$2T + U = 0. \tag{2.3}$$

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

$$\langle v^2 \rangle = \frac{GM}{2R_{\rm vir}}. \tag{2.4}$$

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

$$t_{\rm dyn} = \sqrt{\frac{R_{\rm vir}^3}{GM}}, \tag{2.5}$$

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

```
from amuse.lab import *
import numpy

def dynamical_time_scale(M, R, G=constants.G):
    return numpy.sqrt(R**3/(G*M))
```

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:

```
M = 10**11 |units.MSun
R = 10 |units.kpc
tdyn = dynamical_time_scale(M, R)
print "dynamical time scale=", tdyn
```

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. We could translate the time into a more useful set of units with

```
print "dynamical time scale=", tdyn.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.

2.1.2.2 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 2.6.5). 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)

$$t_{\rm r} \simeq \frac{0.065 \langle v^2 \rangle^{3/2}}{nm^2 G^2 \ln \Lambda}, \tag{2.6}$$

where Λ 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, γ probably is 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 (2.6) by their global averages and assuming virial equilibrium:

$$t_{\rm rh} \simeq 0.138 \frac{N}{\ln \gamma N} t_{\rm dyn}. \tag{2.7}$$

In AMUSE, we could expand the earlier code fragment from Section 2.1.2.1 to

```
from amuse.lab import *
import numpy

def dynamical_time_scale(M, R, G=constants.G):
    return numpy.sqrt(R**3/(G*M))

def relaxation_time_scale(N, M, R, G=constants.G):
    return 0.138*N/numpy.log(0.4*N) * dynamical_time_scale(M, R, G)

N = 500000
M = 10**11 |units.MSun
R = 10 |units.kpc
trel = relaxation_time_scale(N, M, R)
print "relaxation time scale=", trel.in_(units.Gyr)
```

For the Milky Way Galaxy, with $N \sim 10^{11}$ and $t_{\rm dyn}$ as derived above, this gives $t_{\rm rh} \simeq 2.7 \times 10^7$ 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).

2.1.2.3 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 ρ and is (reasonably) characterized by an isotropic velocity distribution $\phi(v)$ (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

$$\mathbf{a}_{\rm df} = -\frac{16\pi^2 G^2 M \rho \ln \Lambda \mathbf{V}}{V^3} \int_0^V v^2 \phi(v) dv, \tag{2.8}$$

where M and $\mathbf{V}$ are the mass and velocity, respectively, of the massive body, and Λ 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

$$\phi(v) = \frac{1}{(2\pi\sigma)^{3/2}} e^{-v^2/\sigma^2}, \tag{2.9}$$

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

$$\mathbf{a}_{\rm df} = -\frac{4\pi^2 G^2 M \rho \ln \Lambda \mathbf{V}}{V^3} \left[\mathrm{erf}(x) - \frac{2x}{\sqrt{\pi}} e^{-x^2} \right], \tag{2.10}$$

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

$$t_{\rm df} = \frac{V}{a_{\rm df}} = \frac{0.059\ V^3}{G^2 M \rho \ln \Lambda}, \tag{2.11}$$

where we have conventionally evaluated the bracketed expression in Equation (2.11) for $x = 1$. Combining Equations (2.6) and (2.11) (and neglecting differences in the definition of Λ), we find

$$t_{\rm df} = 0.91 \left(\frac{V^2}{\langle v^2 \rangle} \right)^{3/2} \left(\frac{m}{M} \right) t_{\rm r}. \tag{2.12}$$

[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.

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.

2.1.2.4 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

$$\epsilon = 24\pi^3 \frac{a^2}{P^2 c^2 (1 - e^2)} \tag{2.13}$$

Here, c is the speed of light. For the planet Mercury, this turns out to be about 5×10^{-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

$$t_{\rm gwr} \simeq 0.02 \frac{c^5 a^4 (1 - e^2)^{7/2}}{G^3 M m (M + m)}, \tag{2.14}$$

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 2015 September 14 (Abbott et al. 2016).

2.1.2.5 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 4.2.2 and 7.4.3, we present examples for which these additional time scales are important.

[2] The `Mikkola` integrator listed in Table 2.1 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.

2.1.3 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 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).

2.1.3.1 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 2.1.2.1). 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 1.4.3) 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.

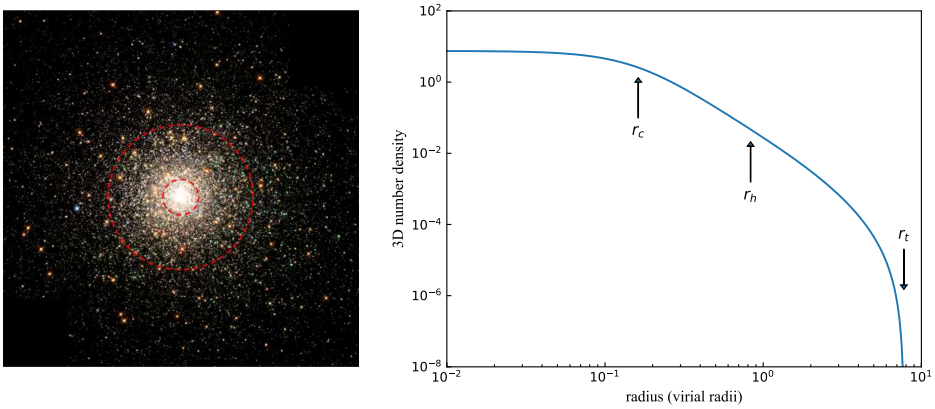


Figure 2.1. (a) The globular cluster M80 lies 10 kpc away and contains several hundred thousand stars. The cluster's core and half-mass radii are indicated by dashed red circles. The field of view is 3 arc min across. (b) The King model that best fits the cluster, with core, half-mass, and tidal radii marked. (Credit: F. R. Ferraro (ESO/Bologna Obs.), M. Shara (STScI/AMNH) et al., and the Hubble Heritage Team (AURA/STScI/NASA). Reproduced with permission.)

Figure 2.1(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 2.1(b) shows the density profile of the best-fitting King model (with $W_0 = 7.5$; see Section 1.4.3). 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.

2.1.3.2 Essential Dynamics

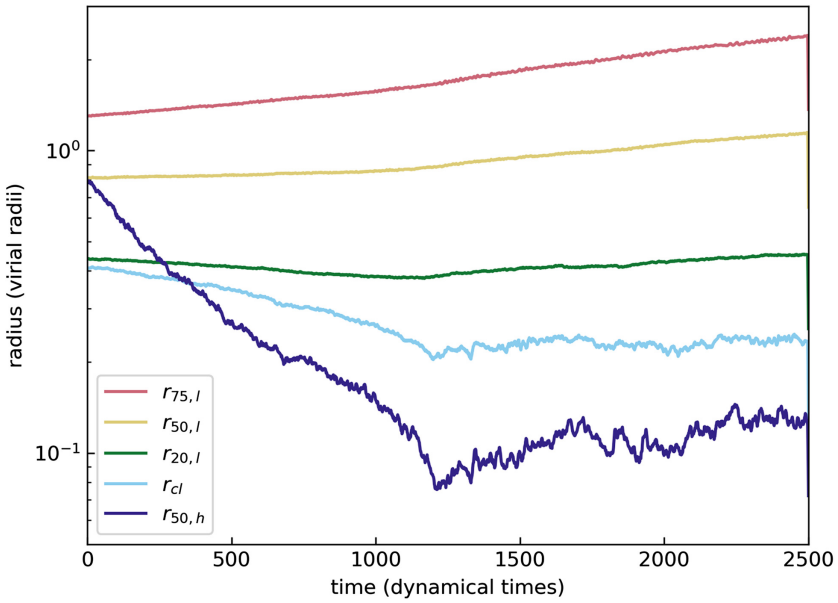
As we just saw, dynamical friction causes the most massive stars in the cluster to sink toward the center. The time scale can be very short, often less than the evolutionary lifetimes of the stars themselves (see Section 2.1.2.3). The most important consequences of this process are: (1) the surrounding 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 4.4.2 and 5.2.6), 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 (2.3)), 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 $t_{cc} \sim 15 t_{rh}$ 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 $t_{cc} \sim 0.2 t_{rh}$ (Portegies Zwart & McMillan 2002).

Figure 2.2 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



Q1

Figure 2.2. 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, as well as the core and 20%, 50%, and 75% Lagrangian radii of the light component are shown. The data have been smoothed using a square window with a total width of 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 ~1200 time units, binary formation and dynamical heating cause both the massive system and the light core to expand. (Data courtesy of M. Brewer.)

(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 (2.7), 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 2.6.3), 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 4.5.

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.

2.1.4 Physics of the Integrator

Evaluating Equation (2.1) 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 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 2.3.9, 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 (2.1) 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 7). 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.

2.1.4.1 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 results of *N*-body simulations are deterministic. As a consequence, 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. Deviations from the trajectory, and the eventual location of the final state, can be caused by external perturbations,

[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.

numerical round-off, and other sources of error. 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).

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 2.1.3)—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.

2.2 *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 2.2.2). Calculating the force in Equation (2.1) 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 $N \gtrsim 10^4$.

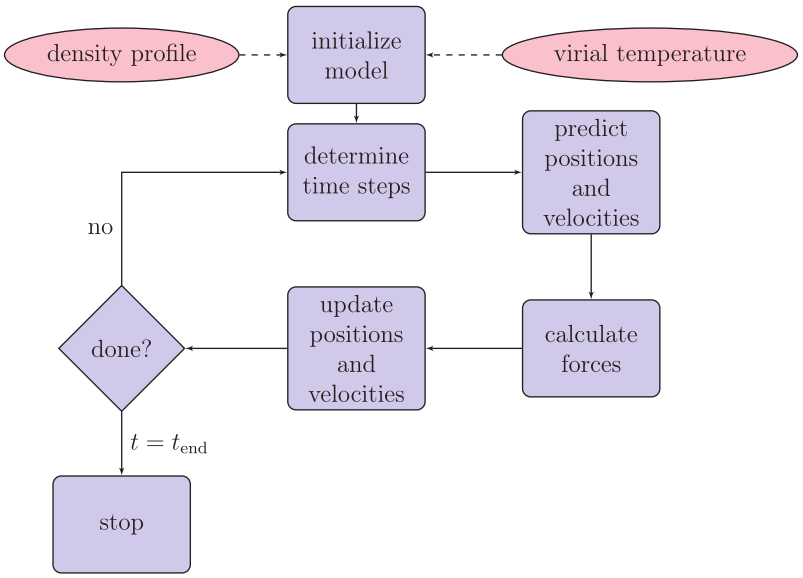


Figure 2.3. Workflow for a typical direct N-body code.

2.2.1 Global Structure of an N-body Code

Figure 2.3 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 2.3 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]

The heart of an N-body code is the integration scheme, the mathematical time-discretization used to solve Equation (2.1). An important consideration in choosing an integration scheme is its order, which tells us how the algorithmic error (per step) δE scales with time step δt. 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 2.3 uses a predictor–corrector scheme to advance the particles in time (see Section 2.2.3). 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.

2.2.2 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 are 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, pp. 377–418).
- 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, pp. 54–65). Another popular method is the fast multipole method (Greengard & Rokhlin 1987; Ambrosiano et al. 1988), which scales even more favorably—$\mathcal{O}(N)$.

2.2.3 Discretization Strategies in *N*-body Simulations

A system of differential equations, such as Equation (2.1), 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 stiffness 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 2.2.2). 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 $j \simeq (a^{\mathrm{new}} - a^{\mathrm{old}})/\delta t$.
- 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. Although of low order, its simplicity and the fact that it is symplectic (see Section 2.2.3.1) 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. 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. 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).

2.2.3.1 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

$$H = H_{\rm kep} + H_{\rm int}. \tag{2.15}$$

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 7, 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 described in Section 2.2.3. 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:

$$\begin{aligned} v' &= v_n + \tfrac{1}{2}a(x_n)\delta t \\ x_{n+1} &= x_n + v'\delta t \\ v_{n+1} &= v' + \tfrac{1}{2}a(x_{n+1})\delta t. \end{aligned} \tag{2.16}$$

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) kick–drift–kick scheme is

$$\begin{aligned} x' &= x_n + \tfrac{1}{2}v_n\delta t \\ v_{n+1} &= v_n + a(x')\delta t \\ x_{n+1} &= x' + \tfrac{1}{2}v_{n+1}\delta t. \end{aligned} \tag{2.17}$$

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 7. 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.

Table 2.1. Overview of the *N*-body Codes Incorporated in AMUSE

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

Notes. Here, TC Refers to tree code (see Section 2.2.2), and PN to the possibility of including post-Newtonian terms. 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. **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, pp. 377–418), Wang et al. (2015), (17) Barnes & Hut (1986), (18) Hamers & Portegies Zwart (2016).

2.2.3.2 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 2.2.2. 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.[6] 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 2.1 are parallel, GPU-accelerated, or both.

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

2.3 Gravity Solvers in AMUSE

A simple script to integrate Newton's equations of motion is presented in Listing 2.1. The script starts by including the Python `time` package and the required AMUSE packages from `amuse.lab`.

```
import time
from amuse.lab import Hermite
from amuse.lab import nbody_system
from amuse.lab import new_king_model
```

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 Section B.6. It also includes the

```
import time
from amuse.lab import Hermite
from amuse.lab import nbody_system
from amuse.lab import new_king_model

def gravity_minimal(N, W0, t_end):
    bodies = new_king_model(N, W0)
    bodies.scale_to_standard()

    gravity = Hermite()
    gravity.particles.add_particles(bodies)
    Etot_init = gravity.kinetic_energy + gravity.potential_energy

    start_time = time.time()
    gravity.evolve_model(t_end)
    dtime = time.time() - start_time

    Ekin = gravity.kinetic_energy
    Epot = gravity.potential_energy

    Etot = Ekin + Epot
    dE = (Etot_init-Etot)/Etot
    print "T =", gravity.get_time(), " CPU time:", dtime, "[s]"
    print "M =", bodies.mass.sum(), " E = ", Etot, " Q = ", -Ekin/Epot
    print "dE =", dE

    gravity.stop()

if __name__ in ("__main__"):
    N = 100
    W0 = 7.0
    t_end = 1 | nbody_system.time
    gravity_minimal(N, W0, t_end)
```

Listing 2.1: Minimal AMUSE code for solving a simple *N*-body problem. Source code is available in `${AMUSE_DIR}/examples/textbook/gravity_minimal.py`.

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 2.1 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.

The code in Listing 2.1 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. As we will see in later chapters, it can be expanded and used for far more complex simulations. This minimal script, like many of the following scripts, is set up as follows:

- generate a realization of the starting conditions (Section 2.3.1),
- specify and initialize the gravity solver to use (Section 2.3.2),
- evolve the model to a specific time (Section 2.3.5),
- print diagnostics,
- stop the community code,
- produce a figure of the results or analyze the data (Section 2.3.6).

After loading in the appropriate packages, the Python interpreter skips to the line

```
if __name__ in ("__main__"):
```

This rather intimidating looking piece of Python tells the interpreter that the following lines are the actual calling sequence when the script is run standalone (see Appendix C). Here, the initial conditions are declared and the main function called. In this example, the initial conditions are: number of particles `N`, some dimensionless structural parameter for the density profile `W0`, and the time when the simulation is supposed to end `t_end`.

```
    N = 100
    W0 = 7.0
    t_end = 1 | nbody_system.time
    main(N, W0, t_end)
```

The quantity `t_end` defined here has a unit, which is attached to the parameter using the "|" operator.[7] Gravitational *N*-body codes often use a dimensionless system of *N*-body units (sometimes called Hénon units; see Heggie & Mathieu 1986, p. 233).[8] In this system, the gravitational constant G, the total cluster mass M, and

[7] In the units module of the `astropy` package (see http://www.astropy.org/), this operator is replaced by the "*" symbol.

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

the cluster virial radius $R_{\rm vir}$ (see Section 2.1.2.1) each have a value of 1. For a cluster in virial equilibrium, the kinetic, potential, and total energies then are

$$T = \frac{1}{4}, \quad U = -\frac{1}{2}, \quad E = -\frac{1}{4}. \tag{2.18}$$

In this case, the unit of `t_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.

N-body units are handy when we are interested in the outcome of a pure gravitational problem. Their use is almost universal in N-body integrators, including most of the N-body codes listed in Table 2.1, as well as in N-body generation functions, such as `new_plummer_model`, `new_king_model`, etc. (see Section 1.4.3). 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 2.3.8). In Section 2.4 (Listing 2.4 in particular) we introduce the command-line option parser `optparse`, and present an example of a script in which we overload the option parser to include units (see Listing B.1 in Appendix B, Section B.4.2).

2.3.1 Generating Initial Conditions

The main routine in Listing 2.1, where the real work is done, starts with

```
def gravity_minimal(N, W0, t_end):
```

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

```
    bodies = new_king_model(N, W0)
```

generates the initial density and velocity profiles—in this case, a King (1966) model with N stars and a dimensionless central potential of $W_0 =$ `W0` (in practice, a real number between 1 and 16). The result is a list of `bodies`, which contain the identities, masses, positions and velocities of the N particles generated. You could inspect the content of the bodies at this point by adding an additional line

```
    print bodies
```

Note that, in Python3, this would be

```
    print(bodies)
```

Either way, this would result in something like:

```
      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
=========  ========  ========  ==========  ... =========  =========  =========
```

The output 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 because we have not specified any units explicitly.

We refer to this list as a *particle set*. We could construct an empty particle set with `bodies=Particles(N)` and subsequently set the masses—for example, as we did in the previous chapter (see Section 1.4.2):

```
bodies.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

```
bodies.mass = 0.2 | nbody_system.mass
```

In this way, we can also set the positions and velocities of all N bodies. Initializing the solar system was explicitly demonstrated in Listing 1.1 of Section 1.4.2.

Often, the velocities are scaled to virial equilibrium (Equation (2.3)). As we saw in Equation (2.18), in N-body units, the total energy is $E = -0.25$ and the virial ratio is $Q = -T/U = 0.5$. We can do this using a built-in `Particles` function:

```
bodies.scale_to_standard()
```

Scaling to N-body units is the default because no other units have been specified. Once scaled to standard units, the time scale is proportional to the cluster's dynamical time scale (see Equation (2.5)).

2.3.2 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 instantiation `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.

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.

For some codes, one can indicate the additional support of accelerating hardware or the use of multi-core architectures. For example, to run the `ph4` code on 32 cores, one would write

```
gravity = ph4(number_of_workers=32)
```

(This will still work even if your computer does not have 32 cores, but it may not speed up the code.) If you prefer to use the code with GPU support, you could write

```
gravity = ph4(mode="GPU")
```

If you are a power user with a GPU cluster, you might combine these invocations:

```
gravity = ph4(number_of_workers=32, mode="GPU")
```

2.3.3 Setting and Getting Parameters in a Community Code

Often, the functionality of a community code needs to be modified, e.g., 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 can provide help in these matters. The command

```
dir(ph4)
```

provides an 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. The comparable parameter in `BHTree` is called `timestep`. 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 ϵ (Aarseth 1963). Softening removes the singularity in the inverse-square force by replacing Equation (2.1) with

$$\mathbf{a}_i = G\sum_{j\neq i}^{N} m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{\left[(\mathbf{r}_j - \mathbf{r}_i)^2 + \epsilon^2\right]^{3/2}}. \tag{2.19}$$

This is done to accommodate integration schemes that can't handle point-mass potentials. Column 8 of Table 2.1 lists whether or not a code is (or can be) softened. In Section 4.5, 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, ϵ^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()`.

2.3.4 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 2.3.1, we initialized a particle set that we called `bodies`. We can send this particle set to the N-body code by adding the `bodies` to the code already running

```
gravity.particles.add_particles(bodies)
```

We now have two sets of particles—those in `bodies` and the set that now lives in the memory of the running code (a separate process that continues to wait for something to do). 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—`Hermite`, `BHTree`, or `ph4`, depending on which code `gravity` points to.

2.3.5 Evolving the Model to the Desired Time

The real work is done in

```
gravity.evolve_model(t_end)
```

Here, the *N*-body code runs to time `t_end`. Because this is the main working routine, we time its operation by enveloping it with timing diagnostics. After the code is done, we record the time spent and calculate the kinetic and potential energies.

To test the performance and accuracy of the integrator, we print the time spent in the integrator, 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 after the Python script ends.

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

```
%> python gravity_minimal.py
```

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

```
T= 1.00063050137 time CPU time: 0.310672998428 [s] M= 1.0 mass
E= -0.250000000148 mass * length**2 * time**-2 Q= -0.506407144201
dE= -5.92372817114e-10
/home/spz/amuse/src/amuse/support/literature.py:73: AmuseWarning:

You have used the following codes, which contain literature references:

      "HermiteInterface"
            Hut, P., Makino, J. & McMillan, S., *Astrophysical Journal
               Letters*, **443**, L93-L96 (1995)
      "AMUSE"
            Portegies Zwart S. et al., Multi-physics simulations using a
               hierarchical interchangeable software interface,
               Computer Physics Communications, 184(3), 456-468 (2013)
               Portegies Zwart S. et al., A multiphysics and multiscale
software
               environment for modeling astrophysical systems,
               New Astronomy*, 14, 369-378 (2009)
warnings.warn(prefix + self.all_literature_references_string(),
   exceptions.AmuseWarning)
```

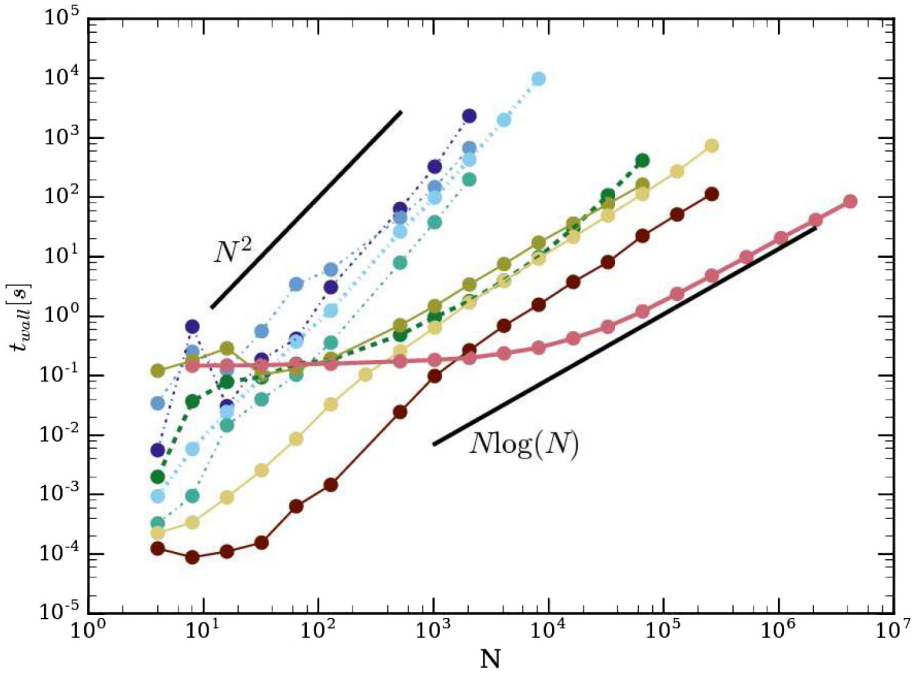


Figure 2.4. Wall-clock time as a function of N for a range of N-body solvers in AMUSE, integrating a Plummer sphere for 1 N-body time unit. The runs were performed for 10 N-body time units. A script to generate this figure can be found at `${AMUSE_DIR}/examples/textbook/plot_Nbody_performance.py`.

This final "warning" message is printed automatically at the end of every script. It provides the user with references that we hope he/she will cite if the code and script are run for production and/or publication purposes.[9] In rather old-fashioned syntax, the single stars indicate italic, whereas double stars indicate boldface—both typical reference styles. Most people, to greater or lesser extents, find this message annoying. This is deliberate—it forces the user to remember and (we hope) cite the sources on which their simulation rests.[10]

Figure 2.4 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 2.4. 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

[9] The AmuseWarning is a Python exception message produced because of the way we have chosen to implement the referencing system. Exceptions are normally meant to catch bugs rather than produce useful information about the usage of the code.
[10] The single stars are intended to indicate italics, double stars boldface.

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.

2.3.6 Retrieving Data from the *N*-body Code

The information stored in `bodies` (mass, position, velocity, etc.) are part of local memory and live in the user Python script but not in the *N*-body code. This local information is not automatically updated after the *N*-body code is run using `gravity.evolve_model()`. To access the new data, we could copy it directly from the remote code by referencing `gravity.particles`, but each access will cause AMUSE to open a one-time MPI or sockets connection to the running code where the remote data live, which can be slow in terms of compute time.

A more efficient way to communicate with a running code is realized by creating a *channel* to the remote data of the running worker code. A communication channel is declared using

```
channel_from_gravity_to_framework \
                = gravity.particles.new_channel_to(bodies)
```

To copy the particle data from the *N*-body code to the local Python framework, we say

```
channel_from_gravity_to_framework.copy()
```

Now the masses, positions, and velocities of the remote data in the worker module are duplicated in the local `bodies` particle set in the framework (Python) script. The local data (in `bodies`) and the remote data (in `gravity.particles`) are consistent. The `copy()` command copies all data, but to tune performance or to prevent overwriting some local data, one can choose to copy only part of the data. For example, to update only the stellar positions, but not their masses, velocities, or other attributes, we could write:

```
channel_from_gravity_to_framework.copy_attribute(["x","y","z"])
```

We could have arranged for the local data to be automatically synchronized with the worker data on each return from the worker code, but for the sake of user control and efficiency, we have opted instead for manual synchronization.

2.3.7 Storing and Recovering Data

We can improve the minimal gravity solver by adding two new features to the code. The first is to store the resulting data in a file for further analysis. It is advisable to

synchronize the local bodies with the code data (see Section 2.3.6) before writing them to a file. A snapshot can be saved to a file named `filename` as follows:

```
write_set_to_file(bodies, filename, "hdf5")
```

The `"hdf5"` argument indicates the format in which the snapshot if stored—in this case, the default, `"hdf5"`. This format is commonly used within the astronomical community; it provides a self-descriptive binary format with the potential of storing hierarchical data structures (The HDF Group 2000–2010). Other recognized types include `"starlab"` (the internal Starlab `dyn` and `star` formats Portegies Zwart et al. 1999; Hut et al. 2010), `"csv"` (comma-separated-values), and ASCII plain text (`"txt"`). Time is not an attribute of the particle set, but rather of the *N*-body integrator; it is not stored, by default. To save the time, use

```
write_set_to_file(bodies.savepoint(0 | model_time.unit), filename, "hdf5")
```

Here, the phrase `bodies.savepoint` ensures that the time is stored with the snapshot. A stored output file can be read back to memory by

```
bodies = read_set_from_file(filename, "hdf5")
```

If time was saved with the snapshot, it can be recovered using

```
model_time = bodies.get_timestamp()
```

Multiple snapshots are often stored in a single file. In that case, one can iterate over the individual snapshots using

```
many_snapshots = read_set_from_file(filename,"hdf5")
for snapshot in many_snapshots.history:
    print "The number of bodies in this snapshot is:", len(snapshot)
```

For runs with a large number of particles, or containing many snapshots, these files can become quite large. One solution is to split the output into smaller pieces and process them separately. Sometimes, however, this is not desirable. In such cases, one can keep the data on disk by keeping the file open while reading:

```
bodies = read_set_from_file(filename, "hdf5",
                            copy_history = False, close_file = False)
```

2.3.8 Using Other Units

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 "Earth's mass is:", (1|units.MEarth).in_(units.kg)
```

or

```
print "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 Section B.2.4.

2.3.9 Interrupting the *N*-body Integrator

Sometimes it is necessary to have a community code stop and hand control back to the framework, leaving subsequent decisions to the user. We call these events *stopping conditions* (see also Section B.3.3). There are many reasons why a community code should stop and return control to the user script, and the circumstances may be quite different for each physical domain. Stopping conditions are intrusive to the community code, and therefore are inconsistent with the AMUSE philosophy of keeping community codes pristine. Nevertheless, we have taken the step of implemented stopping conditions in most AMUSE community codes. The details vary significantly from one code to another. Barnes–Hut tree codes, for example, are generally not built to detect physical collisions between stars, while in regularized *N*-body codes, a "collision" may have an entirely different meaning. In all cases, checks for physical (or other) collisions can usually be incorporated into the interface code or at a high level in the community code without entailing sweeping changes in the source.

In an AMUSE script, one can initiate stopping conditions after initializing the worker code. In a gravity module, this is realized by

```
detect_collision = gravity.stopping_conditions.collision_detection
detect_collision.enable()
```

Many, but not all, AMUSE *N*-body codes have collision detection enabled. Those codes that do not will produce a warning message when the user code attempts to set a stopping condition, but subsequently they simply continue without the ability to stop, offering no other indication that stopping is disabled. In the time-evolution loop, you can check for a stopping condition on return from `evolve_model()` and take appropriate action:

```
        if detect_collision.is_set():
            for cp in range(len(detect_collision.particles(0))):
                particles_in_code = Particles(
                        particles=[detect_collision.particles(0)[cp],
                                   detect_collision.particles(1)[cp]])
                        local_colliding_particles = \
                             particles_in_code.get_intersecting_subset_in(bodies)
                merge_two_particles(gravity, local_colliding_particles)
                gravity.particles.synchronize_to(bodies)
```

The *N*-body integrator checks for the exception after each internal step and returns to the calling function when the event (in this case, a collision) is detected. The *N*-body system is synchronized before returning to the framework to allow the calling function to perform further tasks on the data sets without corrupting the integrity of the *N*-body data.

The flag `detect_collision.is_set()` returns `True` if the stopping condition is met, after which the code for resolving stopping conditions is invoked. Inside this code, the bodies that caused the community code to stop are returned to the Python framework and the user script must deal with the exception. In this example, the gravity solver stops when two bodies approach within a distance smaller than the sum of their radii. The resolution of the exception in this case is to delete the colliding particles and insert a new particle at their center of mass. The following routine accomplishes this as follows:

```
def merge_two_particles(gravity, particles_in_encounter):
    new_particle=Particles(1)
    new_particle.mass = particles_in_encounter.total_mass()
    new_particle.position = particles_in_encounter.center_of_mass()
    new_particle.velocity = particles_in_encounter.center_of_mass_velocity()
    new_particle.radius = particles_in_encounter.radius.sum()
    gravity.particles.add_particles(new_particle)
    gravity.particles.remove_particles(particles_in_encounter)
```

In this routine, a new particle is created and the center-of-mass data of the merging particles are copied to the new particle and added to the `gravity` instantiation. The two merging stars are not needed anymore, so they are removed from `gravity`. Best practice probably requires that we also flag the event, writing the merging particles to a file before removing them from the system (not shown here). Removal

is achieved by explicitly removing the particles from the local `bodies`, adding the center of mass, and explicitly synchronizing `gravity.particles` with the local data:

```
gravity.particles.synchronize_to(bodies)
```

Of course, for all this to happen, one must set the radii of the particles in advance. For example, you could write

```
bodies.radius = 0.001 | nbody_system.length
```

just after the particles have been initialized, but before they are sent to the community module. When working with physical units, that line might look as follows

```
bodies.radius = 10 | units.RSun
```

We return to the use of stopping conditions in Section 4.4, particularly when additional codes are needed to resolve the interrupt.

2.4 Examples

2.4.1 Integrating the Orbits of Venus and Earth

In Section 1.4.2, we presented a plot of the integrated orbits of the Sun, Venus, and Earth (Figure 1.8). Now we can improve our understanding of the scripts used to make this figure.

Listing 2.2 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 of mass—otherwise the Sun, rather than the solar system barycenter, would be placed at this location (although the difference is very small).

```
from amuse.lab import Particles, units

def sun_venus_and_earth():
    particles = Particles(3)
    sun = particles[0]
    sun.mass = 1.0 | units.MSun
    sun.radius = 1.0 | units.RSun
    sun.position = (855251, -804836, -3186) |units.km
    sun.velocity = (7.893, 11.894, 0.20642) | (units.m/units.s)

    venus = particles[1]
    venus.mass = 0.0025642 | units.MJupiter
    venus.radius = 3026.0 | units.km
    venus.position = (-0.3767, 0.60159, 0.03930) | units.AU
    venus.velocity = (-29.7725, -18.849, 0.795) | units.kms

    earth = particles[2]
    earth.mass = 1.0 | units.MEarth
    earth.radius = 1.0 | units.REarth
    earth.position = (-0.98561, 0.0762, -7.847e-5) | units.AU
    earth.velocity = (-2.927, -29.803, -0.0005327) | units.kms

    particles.move_to_center()
    return particles
```

Listing 2.2: Program to initialize the orbits of Venus and Earth around the Sun. The full script can be found in `${AMUSE_DIR}/examples/textbook/Sun_venus_earth.py`.

The integration routine presented in Listing 2.3 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.

2.4.1.1 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 2.4. If we did not choose to store the data, we could instead simply have

```
def integrate_solar_system(particles, end_time):
    from amuse.lab import Huayno, nbody_system
    convert_nbody = nbody_system.nbody_to_si(particles.mass.sum(),
                                             particles[1].position.length())

    gravity = Huayno(convert_nbody)
    gravity.particles.add_particles(particles)
    venus = gravity.particles[1]
    earth = gravity.particles[2]

    x_earth = [] | units.AU
    y_earth = [] | units.AU
    x_venus = [] | units.AU
    y_venus = [] | units.AU

    while gravity.model_time < end_time:
        gravity.evolve_model(gravity.model_time + (1 | units.day))
        x_earth.append(earth.x)
        y_earth.append(earth.y)
        x_venus.append(venus.x)
        y_venus.append(venus.y)
    gravity.stop()
    return x_earth, y_earth, x_venus, y_venus
```

Listing 2.3: Program to integrate the orbits of Venus and Earth around the Sun. (The full listing is in `${AMUSE_DIR}/examples/textbook/Sun_venus_earth.py`.)

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

def plot(x, y):

    pyplot.figure(figsize=(8,8))

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

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

    save_file = "plot_gravity.png"
    pyplot.savefig(save_file)
    print "\nSaved figure in file", save_file,"\n"
    pyplot.show()
```

Listing 2.4: Minimal routine to plot N-body simulation data. Full source code is available in `${AMUSE_DIR}/examples/textbook/plot_gravity.py`.

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

def animate(x, y):

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

    np = len(x)
    fig = pyplot.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 = ["yellow", "green", "blue"]
    size = [40, 20, 20]
    edgecolor = ["orange" ,"green", "blue"]

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

Listing 2.5: Minimal routine to animate *N*-body simulation data. Full source code is available in `${AMUSE_DIR}/examples/textbook/anim_gravity.py`.

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 2.5.

2.4.1.2 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 4.3.2.

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 2.4.1 (Listing 2.2), 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, 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 2.5. Instead of initializing the solar system from the simplified initial conditions given earlier in Listing 2.2, 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 2.6 presents the results of the numerical integration of the solar system over a much longer time scale. We will use this script in Section 2.4.1.3 to compare the variations in the eccentricity of Earth's orbit versus the temperature measurements at the Vostok station in Antarctica.

2.4.1.3 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).

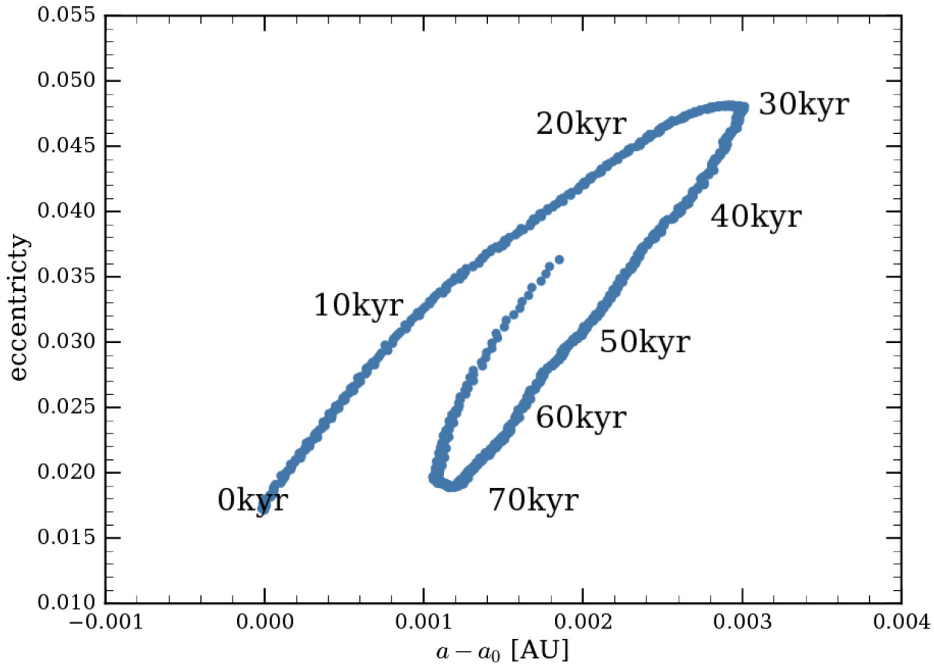


Figure 2.5. Earth's orbital semimajor axis (x-axis) and eccentricity (y-axis) as functions of time over a period of 90,000 yr. Initial conditions are taken from Table 1.1. The complete listing for the script that produced this figure is in `${AMUSE_DIR}/examples/textbook/earthorbitvariation.py`. It should take a minute or so to run on a laptop.

Figure 2.7 shows the measured changes in the temperature at Vostok station and 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.

2.4.2 Small Cluster with Stellar Collisions

In our next example, we shift gears and perform an experiment with a direct N-body code to study the effect of a collision runaway in a small cluster of stars. We opted for an $N = 1000$ body system initially distributed as a $W_0 = 7$ King model with a Salpeter mass distribution between 0.1 $M_\odot$ and 10 $M_\odot$ (see Section 1.4.4). Each star was given a radius of 0.001 N-body length units, which resulted in many interesting collisions. The source code is too long to reproduce here, but it is available in `${AMUSE_DIR}/examples/textbook/gravity_collision.py`.

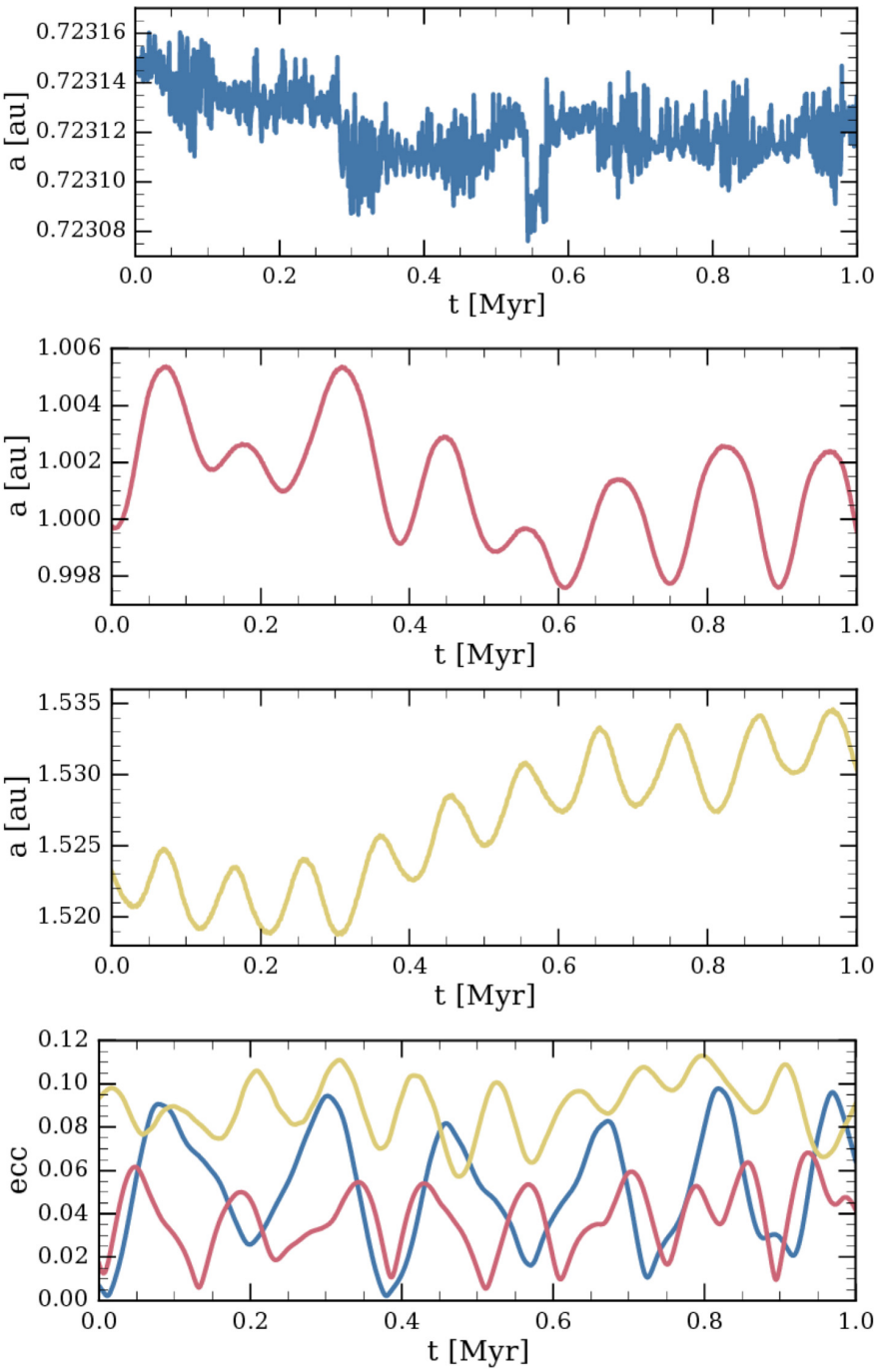


Figure 2.6. From top to bottom, the respective evolutions of the orbital semimajor axes of Venus (blue), Earth (red), and Mars (yellow) for the next million years. The bottom figure gives the orbital eccentricities of the three planets as functions of time, using the same color coding. The integration was performed using a script similar to that presented in Listing 1.2.

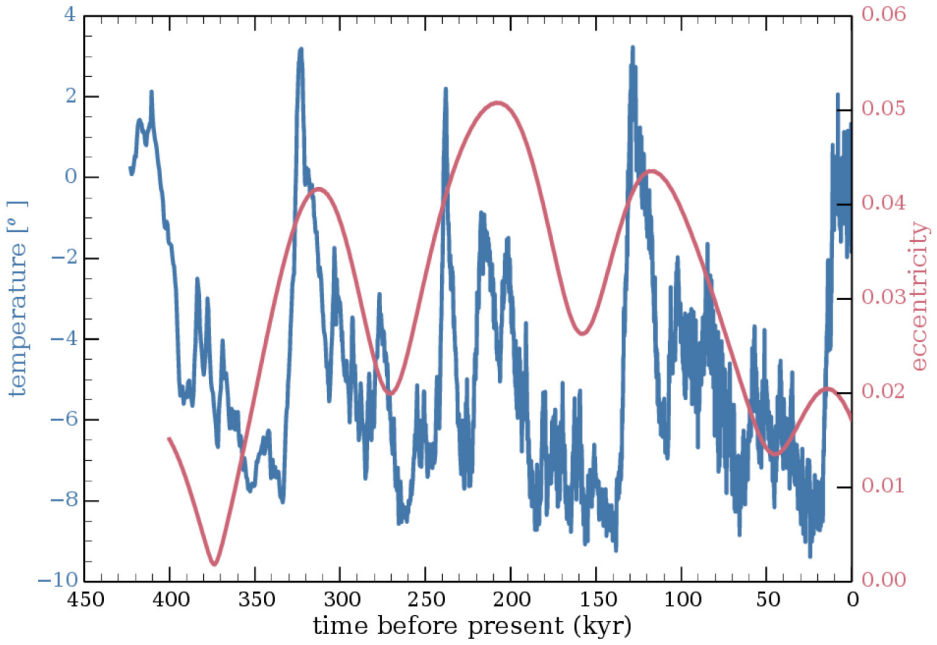


Figure 2.7. (Blue) Historical isotopic temperature record from the Vostok ice core (Jouzel et al. 1987; Petit et al. 1999). (Red) Orbital eccentricity evolution of the Earth over the last 400,000 yr. Integration was performed using `Huayno` with an accuracy parameter $\epsilon = 10^{-3}$. The full listing of the script that produced this figure is in `${AMUSE_DIR}/examples/textbook/Vostok1999temperature.py`. The Vostok ice-core temperature measurements are available at `${AMUSE_DIR}/examples/textbook/vostok1999temperature.data`.

We ran one simulation with virialized initial conditions. Collisions were resolved using the stopping conditions described in Section 2.3.9. For collision products, we assumed no mass loss and calculated a new radius such that the mean density remained constant. The integration was performed using the direct integrator `ph4`. Stellar evolution was not included (see Chapter 3). Figure 2.8 presents the evolution of the core and the half-mass radii of the N-body system; the moments at which collisions occur are indicated by bullets. The size of a bullet is proportional to the mass of the collision product.

2.4.3 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

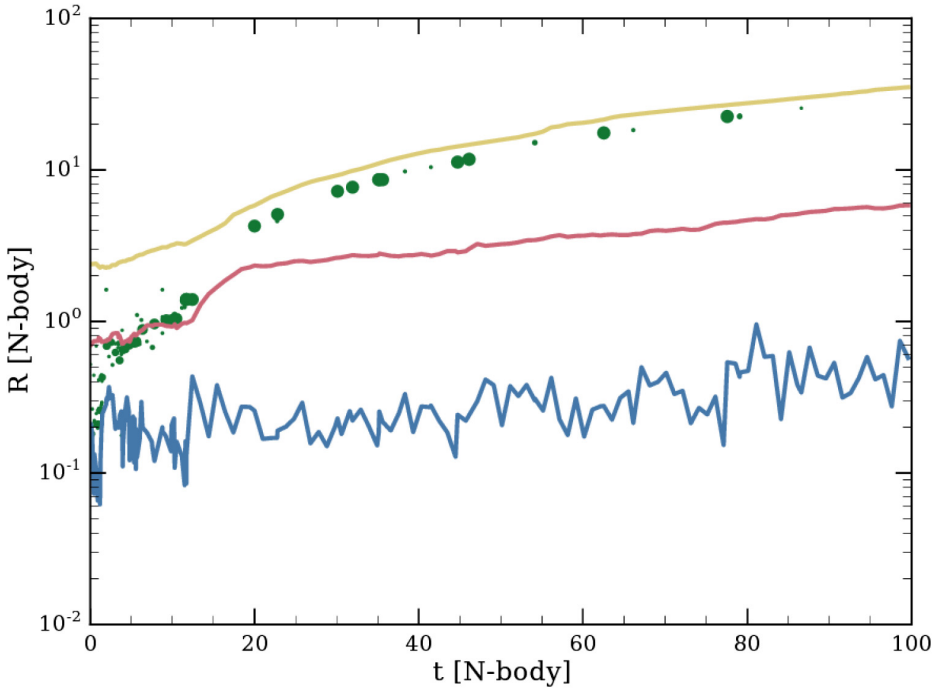


Figure 2.8. Direct N-body simulation (using ph4) of a small star cluster over 100 N-body time units. Initially, the cluster contained $N = 1000$ stars with a Salpeter mass distribution spanning two orders of magnitude in mass, distributed in a virialized King model with $W_0 = 7$. The initial radii of all stars were taken to be 0.001 N-body length unit. The bullets indicate the times and locations of stellar collisions; the size of a bullet indicates the mass of the collision product. We assume that no mass is lost and the mean stellar density remains constant. The three curves (bottom to top) illustrate the core radius (blue), the half-mass radius (red), and the 90% Lagrangian radius (yellow) of the cluster. The script that produced this figure is `${AMUSE_DIR}/examples/textbook/gravity_collision.py`. It should take a few minutes to run on a laptop.

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 `Sakura` (see Table 2.1 or Section B.6), 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 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

```
def initialize_multiple_system(N_bodies, masses, semimajor_axis, eccentricity,
    inclination, argument_of_pericenter, longitude_of_ascending_node):

    N_binaries = N_bodies-1
    particles = Particles(N_bodies+N_binaries)
    for index in range(N_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(N_binaries):
        particle = particles[index+N_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 2.6: 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_DIR}/examples/textbook/hierarchical_quadruple.py`.

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 2.6. 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 $i_\mathcal{B} = 80°$. 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 2.9 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.

2.4.4 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 1.4.3, 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.

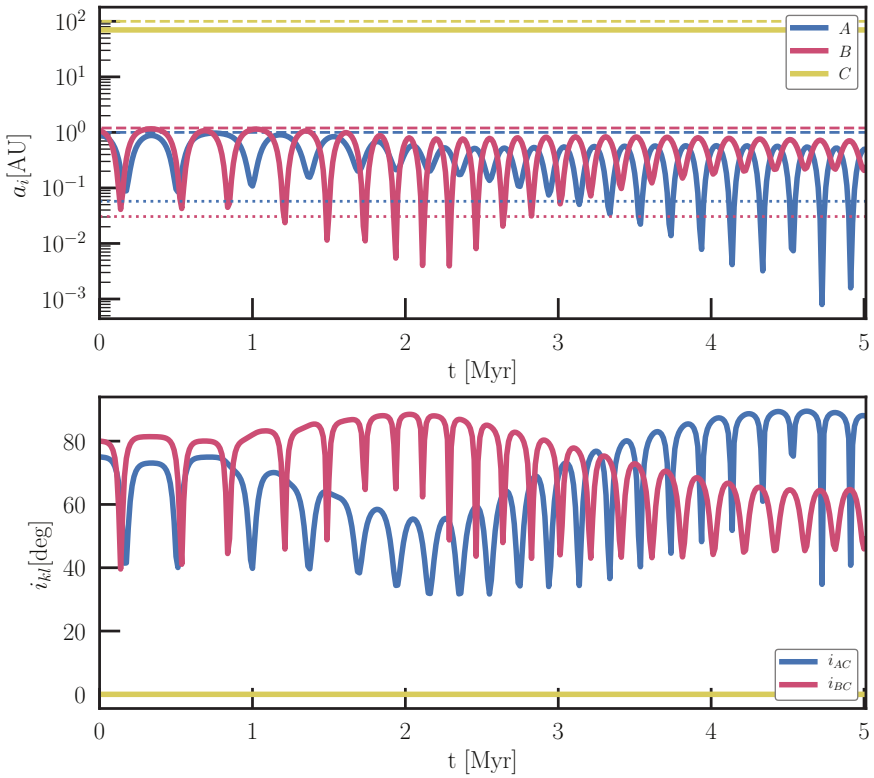


Figure 2.9. Example evolution of a 2 + 2 quadruple system. Top panel: the semimajor axes (dashed lines) and periapsis distances (solid curves) of orbits $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ as a function of time. Bottom panel: the inclinations of $\mathcal{A}$ and $\mathcal{B}$ relative to $\mathcal{C}$. 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}/examples/textbook/secularmultiples_example.py`, which takes about half a minute to run.

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:

```
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)
```

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.e12|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 2.10 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 2.10 (red), this is realized with

```
galaxy1.rotate(0., numpy.pi/2, numpy.pi/4)
```

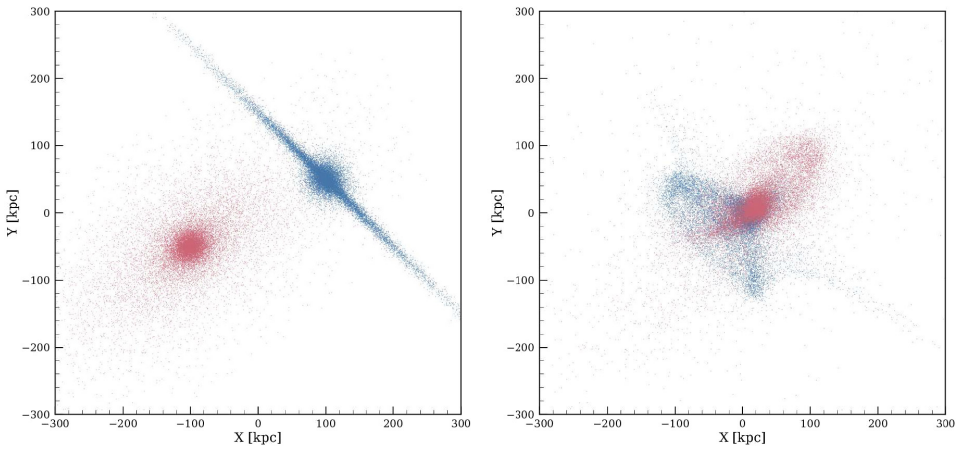


Figure 2.10. Collision between two galaxies, just before they interact (left) and 200 Myr later (right), when both galaxies have been distorted by the interaction. The calculation was performed using `Gadget2` (see Chapter 5; Springel 2000, 2005) as a gravity solver, without hydrodynamics. Both galaxies 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.

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.

2.5 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.

2.5.1 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 2.2.3) 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 2.1.4.1), 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.

2.5.1.1 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 2.1, 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 depends both on the algorithm used to calculate the forces and how the equations of motion are integrated. Figure 2.11 shows the relative energy error for two N-body solvers—a (fourth-order) direct-summation Hermite solver and a (second-order) Barnes–Hut tree code with a Leapfrog solver. Both solvers behave as expected until the time step drops below $\sim 10^{-2}$. For the Hermite scheme, the relative error approaches 10^{-14}, close to the machine precision of 10^{-16}. For these calculations, the round-off in the last significant digit becomes important and starts to dominate the error. The only way to achieve accuracy better than $dE/E \simeq 10^{-14}$ is by adopting a longer mantissa in the calculations. On the other hand, 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.

2.6 Assignments

2.6.1 Orbital Trajectories

Take the initial conditions for the solar system (Table 1.1) and integrate the planets' orbits for 100 yr with the integrator of your choice (Table 2.1). 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 2.6.

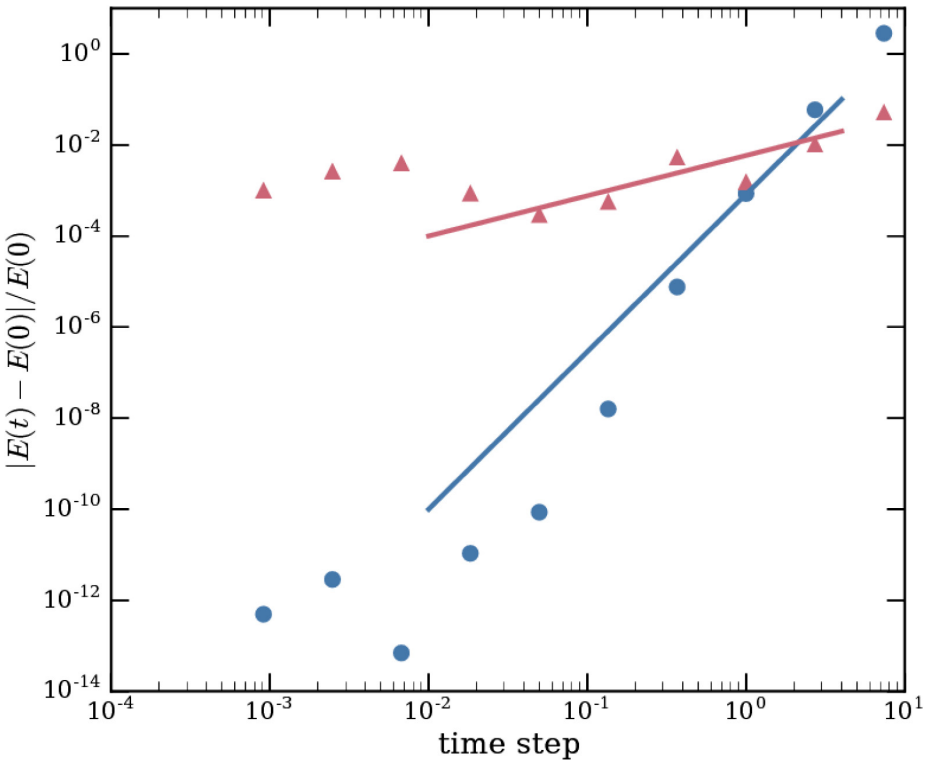


Figure 2.11. Energy error of the integration for one dynamical time of $N = 1000$ equal-mass particles initially distributed in a Plummer sphere. The integration was carried out using the direct N-body integrator `ph4` (blue bullets), and with the `BHTree` tree code (red triangles). The two solid lines show the global trends in energy conservation of the two N-body codes. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/plot_Nbody_precision.py`. It should take 5–15 min to run on a laptop, depending on the number of parallel worker processes used.

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 2.6. In what quantity are the differences most pronounced?

2.6.2 Vostok

In Section 2.4.1.3, 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 2.7 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.

2.6.3 Dynamical Binary Formation

(a) In cluster dynamics, it is common to define an energy scale, called kT (as in thermodynamics), by $K = 3/2\ NkT$, 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 2.1.3.2). Run an N-body simulation with $N = 100$ stars using a code of your choice (see Table 2.1), until the appearance of a binary with a binding energy exceeding 100 kT. You can take the code Listing 2.1 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 (1.1)) with $\alpha = -2.35$ (Salpeter 1955). Remember to rescale the system to virial equilibrium (see Section 2.3.1) 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?

2.6.4 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 2.12 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

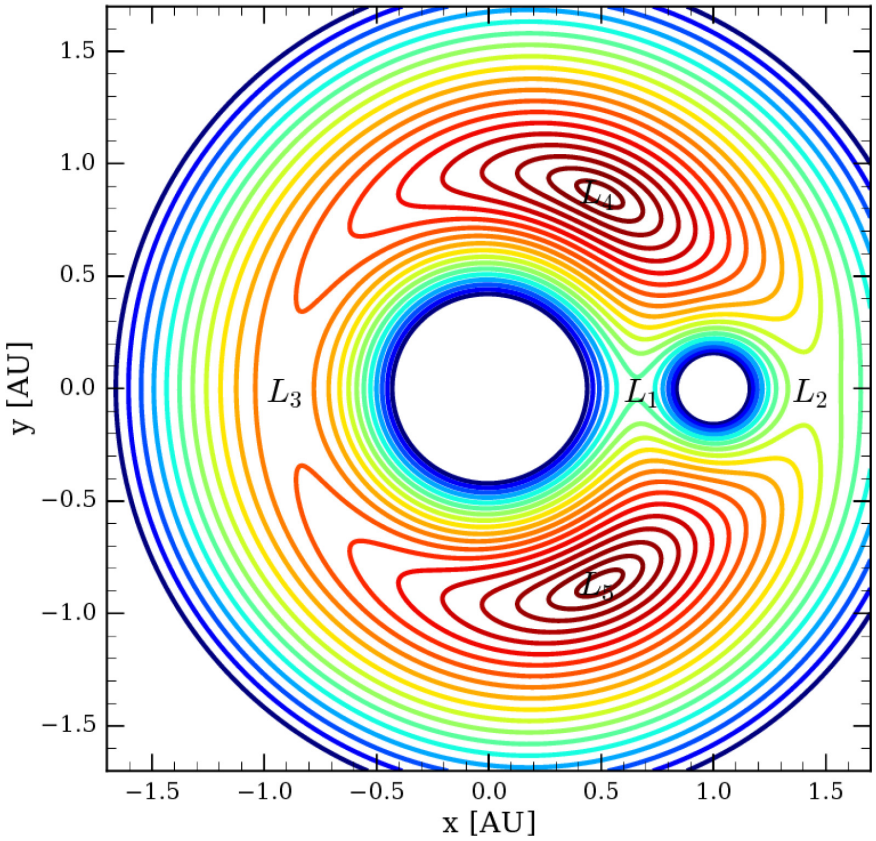


Figure 2.12. Equipotential surfaces for a two-body system. In this example, we adopted a primary mass of 1 $M_\odot$ with a 0.2 $M_\odot$ companion in a circular orbit at a distance of 1 au (to the right). The classical five Lagrange points are indicated by L_1–L_5. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/lagrange_points.py`.

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 2.12, 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.

(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.

2.6.5 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 2.3.1), 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 2.13 presents an example $Q(t)$ for three N-body codes, using 1000 stars of equal mass.

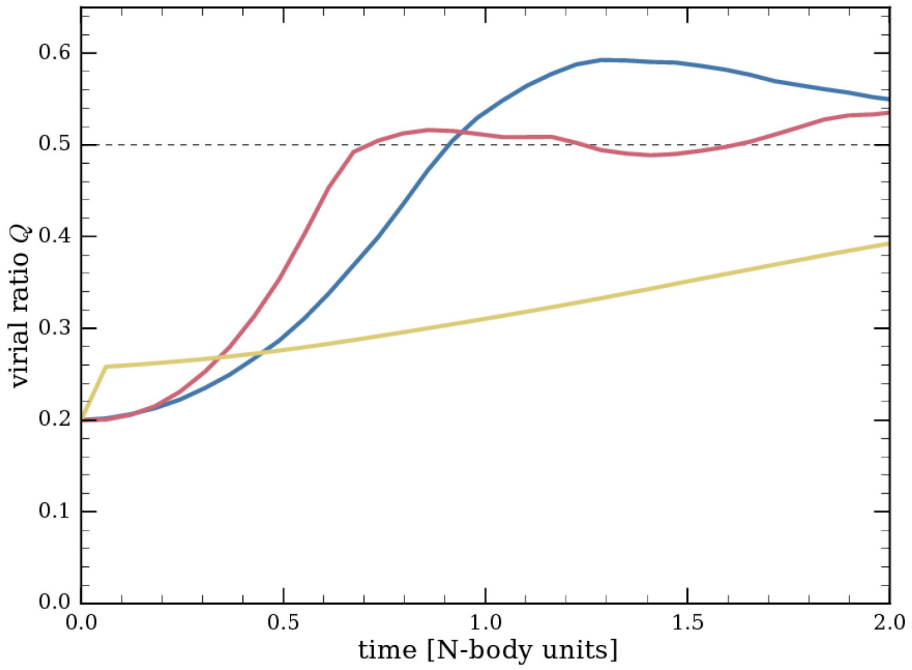


Figure 2.13. Virial ratio Q as a function of time for 1000 equal mass particles initially distributed in a Plummer sphere with $Q = 0.2$. The integrations were performed using ph4 (blue line), Huayno (red), and BHTree (yellow). The script that produced this figure is in `${AMUSE_DIR}/examples/textbook/gravity_to_virial.py`. It should run in less than a minute 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 one N-body time unit?

References

Aarseth S. A. 2003, Gravitational N-body Simulations (Cambridge: Cambridge Univ. Press)
Aarseth, S. J. 1963, MNRAS, 126, 223
Aarseth S. J. 1985, in Multiple Time Scales, ed. J. U. Brackbill,, & B. I. Cohen (New York: Academic)
Abbott, B. P., Abbott, R., Abbott, T. D., et al. 2016, PhRvL, 116, 241102
Ambrosiano, J., Greengard, L., & Rokhlin, V. 1988, CoPhC, 48, 117
Anders, P., Baumgardt, H., Gaburov, E., & Portegies Zwart, S. 2012, MNRAS, 421, 3557
Antonov, V. A. 1962, Solution of the Problem of Stability of Stellar System Emden's Density Law and the Spherical Distribution of Velocities (Vestnik Leningradskogo Universiteta, Leningrad: University)
Barnes, J., & Hut, P. 1986, Natur, 324, 446
Bédorf, J., Gaburov, E., & Portegies Zwart, S. 2012, JCoPh, 231, 2825
Bédorf, J., Gaburov, E., Fujii M. S., et al. 2014, in Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC '14, Galaxy with 18600 GPUs (Piscataway, NJ: IEEE Press)
Bettwieser, E., & Sugimoto, D. 1984, MNRAS, 208, 493
Binney, J., & Tremaine S. 2008, Galactic Dynamics (2nd ed.; Princeton, NJ: Princeton Univ. Press)
Boekholt, T., & Portegies Zwart, S. 2015, ComAC, 2, 2
Brouwer, D. 1937, AJ, 46, 149
Capuzzo-Dolcetta, R., Spera, M., & Punzo, D. 2013, JCoPh, 236, 580
Chambers, J. E. 1999, MNRAS, 304, 793
Chambers, J. E., & Migliorini, F. 1997, BAAS, 29, 1024
Chandrasekhar, S. 1943, ApJ, 97, 255
Chernoff, D. F., & Weinberg, M. D. 1990, ApJ, 351, 121
Cohn, H. 1980, ApJ, 242, 765
de Lagrange, J.-L. 1772, Chapitre II: Essai sur le Problème des Trois Corps (Gauthier-Villars: Paris), 6, 229
Dehnen, W. 2014, ComAC, 1, 1
Duncan, M. J., Levison, H. F., & Lee, M. H. 1998, AJ, 116, 2067
Einstein, A. 1916, AnP, 49, 769
Fellhauer, M., & Lin, D. N. C. 2007, MNRAS, 375, 604
Fukushige, T., & Heggie, D. C. 1995, MNRAS, 276, 206
Gaburov, E., Bédorf, J., & Portegies Zwart, S. 2010, OCTGRAV: Sparse Octree Gravitational N-body Code on Graphics Processing Units, Astrophysics Source Code Library, ascl:1010.048

Giersz, M., & Heggie, D. C. 1994, MNRAS, 268, 257
Giersz, M., & Heggie, D. C. 1996, MNRAS, 279, 1037
Gildor, H., & Tziperman, E. 2000, PalOc, 15, 605
Gnedin, N. Y., Glover, S. G. O., Klessen, R. S., & Springel V. 2015, Star Formation in Galaxy Evolution: Connecting Numerical Models to Reality (Berlin: Springer)
Greengard, L., & Rokhlin, V. 1987, JCoPh, 73, 325
Hamers, A. S., & Portegies Zwart, S. F. 2016, MNRAS, 459, 2827
Harfst, S., Gualandris, A., Merritt, D., et al. 2007, NewA, 12, 357
HDF Group 2000-2010, Hierarchical data format version 5, http://www.hdfgroup.org/HDF5, release version HDF5-1.8.20, HDF5-1.10.3
Heggie, D., & Hut P. 2003, in The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics, ed. D. Heggie,, & P. Hut (Cambridge: Cambridge Univ. Press)
Heggie, D. C. 1975, MNRAS, 173, 729
Heggie, D. C., & Mathieu R. D. 1986, The Use of Supercomputers in Stellar Dynamics, Lecture Notes in Physics, Vol. 267, ed. P. Hut,, & S. L. W. McMillan (Berlin: Springer)
Hénon M. 1975, in IAU Symp. 69, Dynamics of the Solar Systems, ed. A. Hayli (Cambridge: Cambridge Univ. Press)
Hills, J. G̃. 1975, AJ, 80, 809
Hockney, R. W., & Eastwood J. W. 1988, Computer Simulation Using Particles (Bristol: Hilger)
Hulse, R. A., & Taylor, J. H. 1975, ApJL, 195, L51
Hut, P., Makino, J., & McMillan, S. 1995, ApJL, 443, L93
Hut, P., McMillan, S., Makino, J., & Portegies Zwart, S. 2010, Starlab: A Software Environment for Collisional Stellar Dynamics, Astrophysics Source Code Library, ascl:1010.076
Inagaki, S., & Saslaw, W. C. 1985, ApJ, 292, 339
Iwasawa, M., Portegies Zwart, S., & Makino, J. 2015, ComAC, 2, 6
Jänes, J., Pelupessy, I., & Portegies Zwart, S. 2014, A&A, 570, A20
Jouzel, J., Lorius, C., Petit, J. R., et al. 1987, Natur, 329, 403
King, I. R. 1966, AJ, 71, 64
Kozai, Y. 1962, AJ, 67, 591
Kuijken, K., & Dubinski, J. 1995, MNRAS, 277, 1341
Laskar, J., & Robutel, P. 2001, CeMDA, 80, 39
Lee, H. M., & Ostriker, J. P. 1987, ApJ, 322, 123
Lidov, M. L. 1962, P&SS, 9, 719
Liu, L., Wu, X., Huang, G., & Liu, F. 2016, MNRAS, 459, 1968
Makino, J. 1991, ApJ, 369, 200
Makino, J. 1996, ApJ, 471, 796
Makino, J., & Aarseth, S. J. 1992, PASJ, 44, 141
Mardling, R. A., & Aarseth, S. J. 2001, MNRAS, 321, 398
McMillan, S. L. W., & Aarseth, S. J. 1993, ApJ, 414, 200
Mikkola, S. 1983, MNRAS, 203, 1107
Mikkola, S., & Merritt, D. 2008, AJ, 135, 2398
Milankovitch, M. 1998, Canon of Insolation and the Ice-Age Problem (1st ed., Agency for Textbooks). Originally published as Zavod za Udzbenike i Nastavna Sredstva: Kanon der Erdbestrahlung und seine Andwendung auf das Eiszeiten-problem (Belgrade: Mihaila Ćurčića, 1941)
Milgrom, M. 1983, ApJ, 270, 365

Newton, I. 1687, Philosophiae Naturalis Principia Mathematica, 1, 13
Nitadori, K., & Makino, J. 2008, NewA, 13, 498
Oshino, S., Funato, Y., & Makino, J. 2011, PASJ, 63, 881
Peters, P. C. 1964, PhRv, 136, 1224
Petit, J. R., Jouzel, J., Raynaud, D., et al. 1999, Natur, 399, 429
Plummer, H. C. 1911, MNRAS, 71, 460
Portegies Zwart, S. F., & Boekholt, 2018, CNSNS, 61, 160
Portegies Zwart, S. F., & McMillan, S. L. W. 2002, ApJ, 576, 899
Portegies Zwart, S. F., Makino, J., McMillan, S. L. W., & Hut, P. 1999, A&A, 348, 117
Portegies Zwart, S. F., Belleman, R. G., & Geldof, P. M. 2007, NewA, 12, 641
Rauch, K. P., & Tremaine, S. 1996, NewA, 1, 149
Rein, H., & Liu, S.-F. 2011, REBOUND: Multi-purpose N-body code for collisional dynamics, Astrophysics Source Code Library, ascl:1110.016
Rein, H., & Liu, S.-F. 2012, A&A, 537, A128
Rein, H., & Spiegel, D. S. 2015, MNRAS, 446, 1424
Rodriguez, C. L., Morscher, M., Wang, L., et al. 2016, MNRAS, 463, 2109
Saha, P., & Tremaine, S. 1992, AJ, 104, 1633
Salpeter, E. E. 1955, ApJ, 121, 161
Spera, M., Capuzzo Dolcetta, R., & Punzo, D. 2012, HiGPUs: Hermite's N-body integrator running on Graphic Processing Units, Astrophysics Source Code Library, ascl:1207.002
Spinnato, P. F., Fellhauer, M., & Portegies Zwart, S. F. 2003, MNRAS, 344, 22
Spitzer L. 1987, Dynamical Evolution of Globular Clusters (Princeton, NJ: Princeton Univ. Press)
Spitzer, L. Jr. 1969, ApJL, 158, L139
Springel, V. 2000, GADGET-2: A Code for Cosmological Simulations of Structure Formation, Astrophysics Source Code Library, ascl:0003.001
Springel, V. 2005, MNRAS, 364, 1105
Takahashi, K., & Portegies Zwart, S. F. 1998, ApJL, 503, L49
Tremaine, S. 2015, ApJ, 807, 157
Trenti, M., & van der Marel, R. 2013, MNRAS, 435, 3272
Tricarico, P. 2012, ORSA: Orbit Reconstruction, Simulation and Analysis, Astrophysics Source Code Library, ascl:1204.013
Verlet, L. 1967, PhRv, 159, 98
Verlinde, E. P. 2017, ScPP, 2, 016
Wang, L., Spurzem, R., Aarseth, S., et al. 2015, MNRAS, 450, 4070
Whitehead, A. J., McMillan, S. L. W., Vesperini, E., & Portegies Zwart, S. 2013, ApJ, 778, 118
Wisdom, J., & Holman, M. 1991, AJ, 102, 1528
Yoshida, H. 1990, PhLA, 150, 262
Yoshida, H. 1993, CeMDA, 56, 27
Zemp, M. 2014, Halogen: Multimass spherical structure models for N-body simulations, Astrophysics Source Code Library, ascl:1407.020
Zemp, M., Moore, B., Stadel, J., Carollo, C. M., & Madau, P. 2008, MNRAS, 386, 1543