In Figure 9.2 we present part of the analysis as explored in Saz Ulibarrena & Portegies Zwart (2025), in which a reinforcement algorithm was used to find the optimum bridge time step. the two axes represent integration (wall-clock) time and energy error. The curve shows the trade-off between the two if we reduce the bridge time step.

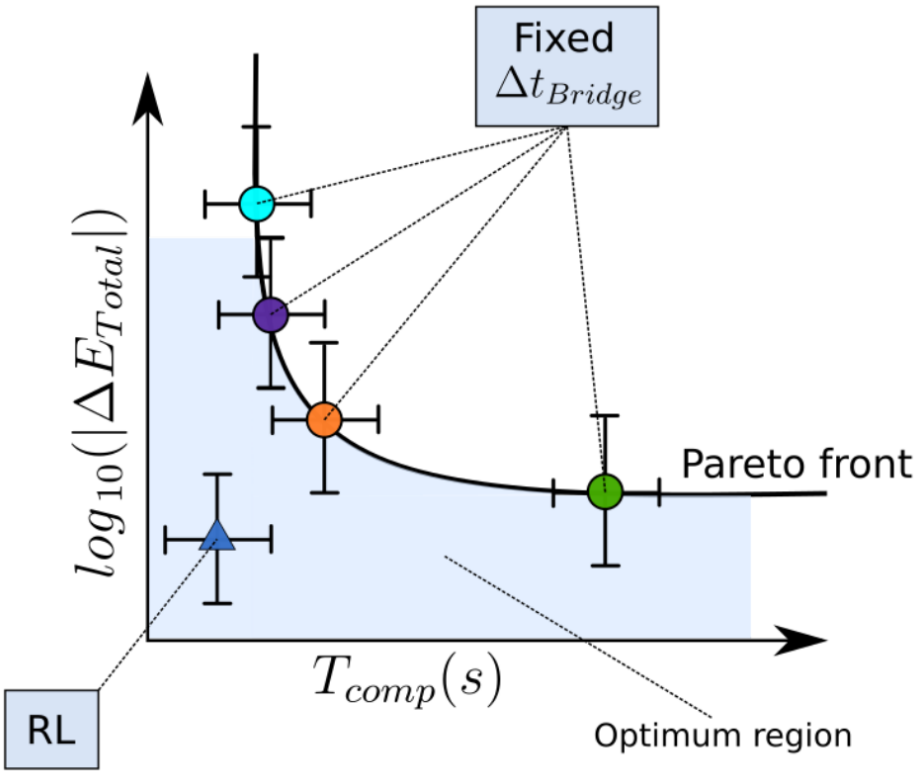

Figure 9.2: Perato curve for the bridge time-step when tuned using reinforcement learning. The bullet points represent fixed bridge time steps, whereas the reinforcement algorithm allows bridge time steps to change with time.

The automated routine to determine the bridge time step, using reinforcement learning, is interesting and in principle powerful. If fully operational, it would have two major advantages: 1) it would not require any expert knowledge to choose a bridge time step, and 2) the calculation would be faster with a smaller overall energy error. In practice, however, it still turns to be hard to find an optimal choice. For now, we are not really encouraging using reinforcement learning to determine your bridge time step, but in the near future we hope that this method can be made into a workable module.

9.2 Using Bridge

Fortunately, you don't have to code all the intricate details of the Bridge method yourself. AMUSE has a `bridge` class that takes care of the details of implementing the classic and higher-order Bridge schemes just described. We illustrate this by developing a simple model of a star cluster moving in an analytic model of the potential near the

Galactic center (McMillan & Portegies Zwart, 2003; Harfst *et al.*, 2010). Then we show how this model can be extended easily to reproduce the seminal results of Fujii *et al.* (2007), which represented the first fully self-consistent treatment of star clusters near the Galactic center.

9.2.1 Star Cluster in a Static Galactic Potential

Code example 9.1 presents a simple class that provides a power-law analytic approximation to the mass distribution and gravitational field within a few hundred parsecs of the Galactic center:

$$\begin{aligned} M(r) &= M_{\rm gal}\left(\frac{r}{R_{\rm gal}}\right)^{\alpha} \\ a(r) &= -\frac{GM(r)}{r^2}, \end{aligned} \tag{9.14}$$

where $M(r)$ is mass interior to radius r and we take $M_{\rm gal} = 1.6\times10^{10}\,{\rm M}_\odot$ within radius $R_{\rm gal} = 1\,{\rm kpc}$, and $\alpha = 1.2$ (Mezger *et al.*, 1996). We initialize the class with

```
galactic_center = GalacticCenterGravityCode(Rgal, Mgal, alpha)
```

```
26 class GalacticCenterGravityCode(object):
27     def __init__(self,R, M, alpha):
28         self.radius=R
29         self.mass=M
30         self.alpha=alpha
31 
32     def get_gravity_at_point(self,eps,x,y,z):
33         r2=x**2+y**2+z**2
34         r=r2**0.5
35         m=self.mass*(r/self.radius)**self.alpha
36         fr=constants.G*m/r2
37         ax=-fr*x/r
38         ay=-fr*y/r
39         az=-fr*z/r
40         return ax,ay,az
41 
42     def circular_velocity(self,r):
43         m=self.mass*(r/self.radius)**self.alpha
44         vc=(constants.G*m/r)**0.5
45         return vc
```

Code example 9.1: Semi-analytic power-law description of the gravitational field in the inner few hundred parsecs of the Galactic center. The quantity `mass` is the mass within distance `radius` of the center, and `alpha` is the slope of the power-law mass distribution.

Key to this (and any) bridge class is the function `get_gravity_at_point()`, which takes a position in space and a softening length as arguments, then returns the gravitational acceleration at the specified location. This function allows any gravity code to be bridged with any other in AMUSE. (In this particular case, the softening length

is ignored, but the argument is required by bridge because it is used by other codes for which softening may be necessary.)

Next, we initialize the star cluster by starting an N-body code and generating a realization of particle masses, positions and velocities. In the example in Code example 9.2, we initialize a code called `nbodycode`, which could be either a direct N-body code, a tree code, or a gravity-enabled SPH code. The cluster is generated from a King (1966) model with `N` particles, total mass `Mcluster`, virial radius `Rcluster`, and a dimensionless potential depth `W0` (see also Section 4.1.3.1). Additional parameters for the N-body code can be passed using the argument `parameters`. In this case, we provide the softening parameter as an argument because we will run this script with a tree code, which requires softening.

```
    Rgal = 1. | units.kpc
    Mgal = 1.6e10 | units.MSun
    alpha = 1.2
    galaxy_code = GalacticCenterGravityCode(Rgal, Mgal, alpha)

    m=galaxy_code.mass*((100|units.pc)/galaxy_code.radius)**galaxy_code.alpha
    cluster_code = make_king_model_cluster(BHTree, N, W0, M, R,
                                           parameters=[("epsilon_squared",
                                                        (0.01 | units.parsec)**2)])

    stars = cluster_code.particles.copy()
    stars.x += Rinit
    stars.vy = 0.8*galaxy_code.circular_velocity(Rinit)
    channel = {"from_framework": stars.new_channel_to(cluster_code.particles),
               "to_framework": cluster_code.particles.new_channel_to(stars)}
    channel["from_framework"].copy_attributes(["x","y","z","vx","vy","vz"])

    plot_cluster(f, stars.x.value_in(units.pc), stars.y.value_in(units.pc), c='r')

    system = bridge(verbose=False)
    system.add_system(cluster_code, (galaxy_code,))

    times = quantities.arange(0|units.Myr, tend, timestep)
    xcom = [] | units.pc
    ycom = [] | units.pc
    for i,t in enumerate(times):
        system.evolve_model(t,timestep=timestep)
        channel["to_framework"].copy_attributes(["x","y","z","vx","vy","vz"])
        com = stars.center_of_mass()
        xcom.append(com[0])
        ycom.append(com[1])

    plt.plot(xcom.value_in(units.pc), ycom.value_in(units.pc), lw=1, c='k')

    x = system.particles.x.value_in(units.parsec)
    y = system.particles.y.value_in(units.parsec)
    cluster_code.stop()
```

Code example 9.2: Initializing the gravity code to be coupled via bridge to a background potential.

The bridge is created by constructing a new `gravity` solver that combines the N-body code `cluster_code` for the star cluster and the analytic `galaxy_code` for the galactic background. The bridge is declared by

```
gravity = bridge.Bridge()
```

and subsequently we identify the codes to be bridged with

```
gravity.add_system(cluster_code, (galaxy_code,))
```

This line adds `cluster_code` as a code to be bridged, and indicates that it interacts with `galaxy_code`. In general, the second argument is a list of codes with which the first code interacts.[1]

In this example, the bridge is unidirectional: the dynamical evolution of the particles in `cluster_code` code is affected by `galaxy_code`, but not the other way around. Multiple codes can be added to a bridge, and their interactions tailored to the problem at hand. This can be very effective if, for example, our galaxy code is composed of several separate parts—for the bulge, bar, and disk, say—and we wanted to model several clusters in the bridge:

```
gravity.add_system(cluster1, (cluster2, cluster3, bulge, bar, disk))
gravity.add_system(cluster2, (cluster1, cluster3, bulge, bar, disk))
gravity.add_system(cluster3, (cluster1, cluster2, bulge, bar, disk))
```

and so on.

Once the bridge is initialized and the integrators added, we set the bridge time step:

```
gravity.timestep = 0.1 | units.Myr
```

It often requires some experimentation to determine what this time step should be. Too long a time step results in inaccurate inclusion of the coupling, whereas too short a time step may make the calculation too expensive. It is possible to change the bridge time step at run time (see Section 10.1.3), but there is currently no ready algorithm to determine its optimum value.

In the event loop over `time`, we now call `evolve_model` from the bridged `gravity` solver

```
gravity.evolve_model(time)
```

Synchronization between the internal code particles and those in the Python script is realized via channels. The event loop is presented in Code example 9.1.

Figure 9.3 shows the results of a simulation of the orbital evolution of the stars in the Arches cluster in the field of the Galactic center. The 50,000 $M_\odot$ star cluster was represented by 1024 equal-mass particles (not a good approximation to the observed mass function of the Arches!) initially represented by a virialized King model (King,

[1] A Python aside: the trailing comma after `galaxy_code` turns the single argument into a tuple that can be evaluated. Forgetting the trailing comma will lead to an error with the inscrutable and somewhat intimidating message `"object is not iterable"`. It is only needed when there is only one item in the list.

```
def make_king_model_cluster(nbodycode, N, W0, Mcluster,
                            Rcluster, parameters = []):

    converter=nbody_system.nbody_to_si(Mcluster,Rcluster)
    bodies=new_king_model(N,W0,convert_nbody=converter)

    code=nbodycode(converter)
    for name,value in parameters:
        setattr(code.parameters, name, value)
    code.particles.add_particles(bodies)
    return code
```

Code example 9.3: Coupling a gravity code to a background potential using a bridge. The last three lines store the stars x and y positions and terminate the code. In the example we adopted the following parameters: `number_of_particles = 1024`, `king_w0 = 3`, `distance_initial = 50.0 | units.pc`, `mass_cluster = 5.0e4 | units.MSun`, and `radius_cluster = 0.8 | units.pc`.

1966) with a radius of 0.8 pc. The cluster had a non-circular orbit in the Galactic field, starting at a distance of 50 pc. We adopted a Barnes–Hut tree code with a rather large softening of 0.1 pc, to allow us to quickly integrate the motion of the stars in the cluster without getting too distracted by the details of the internal cluster dynamics.

9.2.2 Adding extra functionality with a kick to a bridge

Sometimes one would like to add a functionality to the bridge operator in order to kick the system code. A simple example can be considered with the dynamical friction on a black hole of intermediate mass due to its orbit through a galactic nucleus. We discussed a similar issue in Section 4.1.2.3, but without addressing how to add such a term to the integration of an N-body system.

In principle, this process does not work much different than the bridging of two codes, except that in this case one of the codes has no `particle` attribute and doesn't need to be kicked nor drifting itself. In the example, below, we initiate the `Hermite` integrator two intermediate mass black holes that orbit in a background potential. This example is not very different than what we already discussed in Section 9.2.1.

The difference is that we would like to exert a force on the black holes due to the effect of dynamical friction excreted on the black holes. We simply start the N-body code (here `Hermite`), and add the black holes. Then we start the `DynamicalFriction` code, and bridge them together.

```
code1 = Hermite()
code1.particles.add_particles(cluster)
code2 = DynamicalFriction(Coulomb_logarithm=3.7)
system = Bridge(timestep=0.1 | units.Myr)
system.add_system(code1)
system.add_code(code2)
system.evolve_model( 10 | units.Myr )
```

The dynamical friction solver comprises of two parts, a Plummer potential, and a dynamical friction term. We have created a small class to achieve this, in Figure 9.4.

Since the `Hermite` code is 4th order, we use the symbol S_4 (for stars) for the

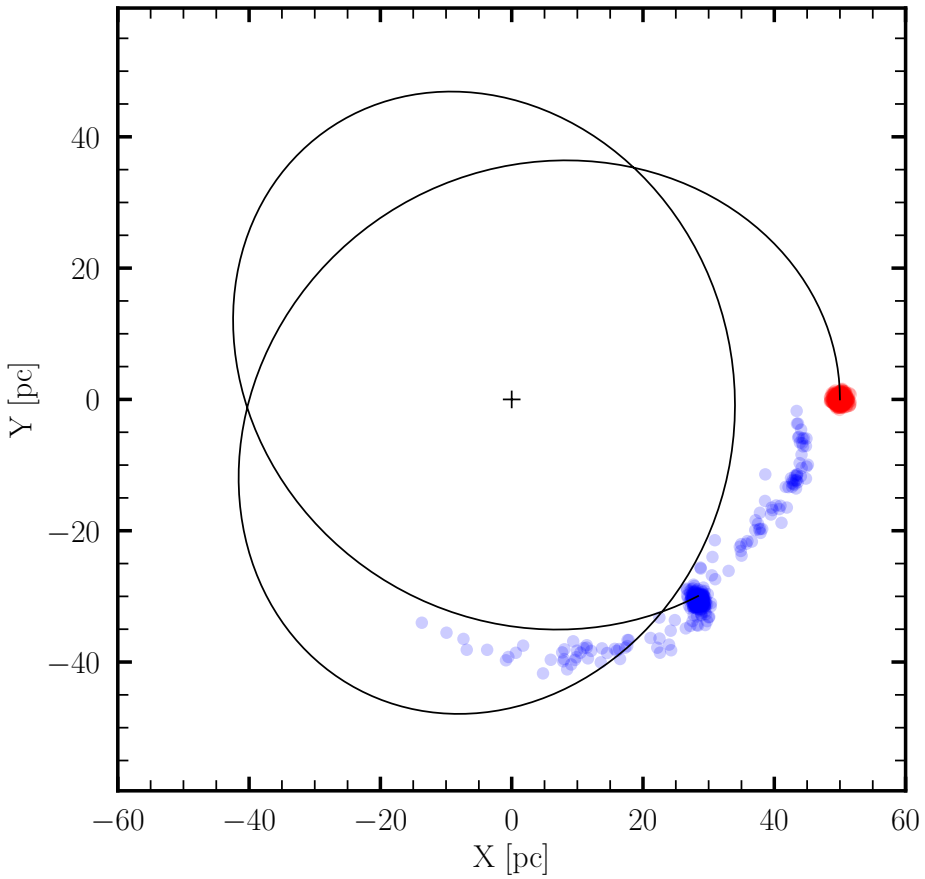

Figure 9.3: Snapshots of an Arches-like cluster orbiting near the Galactic center. Initially (in red) the cluster was composed of 5000 particles taken from a Salpeter mass function over two orders of magnitude with a mean mass of $10\,M_\odot$ leading to a total mass of $50{,}000\,M_\odot$, distributed as a King model in virial equilibrium with $W_0 = 3$ and a virial radius of 0.8 pc. The center of mass of the cluster was placed 50 pc from the Galactic center (along the x-direction) with 80% of the circular velocity (in the y-direction). A softening of 0.1 pc was introduced for all particles in the cluster. The integration was performed using the background galactic potential presented in Code example 9.1. The initialization and main event loop are shown in Code example 9.3 and Code example 9.2, respectively. The equations of motion of the cluster are integrated using the `BHtree` code. The black curve follows the cluster's center of mass, ending in the final shapshot (in blue). The source code for the figure is `amuse.examples.gravity_potential`.

sub-system. The potential is indicated with the letter G (for Galaxy). The bridge is indicated with the left-pointed arrow, and the augmentation is indicated with the letter f (for function) below the bridge arrow. The notation for the augmented bridge in which a 4th order direct N-body code is bridged with a potential including an additional function for (in this case for the dynamical friction term) then has the following notation: $[S_4 \underset{f}{\leftarrow} G]$. this may look somewhat arcane, at first, but once augmented bridge become hierarchical, multi-layers or more complex in other means, this notation captures these complexities nicely.

A star cluster in the Galactic center will feel a frictional force due to the accumulation of stars in its wake. McMillan & Portegies Zwart (2003) used a direct integration technique to study the time scale on which the Arches star cluster sinks from its birth location at about 30 pc from the Galactic center towards the middle of the Milky Way.

It turns out that the time scale depends quite sensitively on the Coulomb parameter $\ln\Lambda$. In their Fig. 6 they present the results of several calculations for the distance to the Galactic center as a function of time for a black hole (or star cluster) of $50\,000\,\mathrm{M}_\odot$. In Figure 9.4 we repeat their calculation but using an external function included in the bridge step

$$[S_4 \underset{f}{\leftarrow} G] \tag{9.15}$$

The equation to solve in $f = f(S, G)$ we calculate the acceleration of the individual particles in the N-body code due to dynamical friction (See Equation 7 of McMillan & Portegies Zwart (2003)):

$$\vec{a}_f = -4\pi \ln(\Lambda) G^2 \rho m \frac{\vec{v}_c}{v_c^3} \chi. \tag{9.16}$$

Here G is the gravitational constant, ρ the local stellar density, with mass m, $\chi \simeq v_c/\sqrt{2}\sigma$ (with σ the local 1-dimensional velocity dispersion), and $\ln(\Lambda) \simeq \ln(r)/\ln(R)$ is the Coulomb logarithm. Here r is the distance of the stars to the potential center (which has size R), and $\vec{v}_c$ is the particle's velocity vector. The acceleration ($\vec{a}_f$) is negative, because dynamical friction results in a drag force leading to deceleration.

The class, performing this calculation together with hovering the stars on the background potential is presented in Figure 9.4.

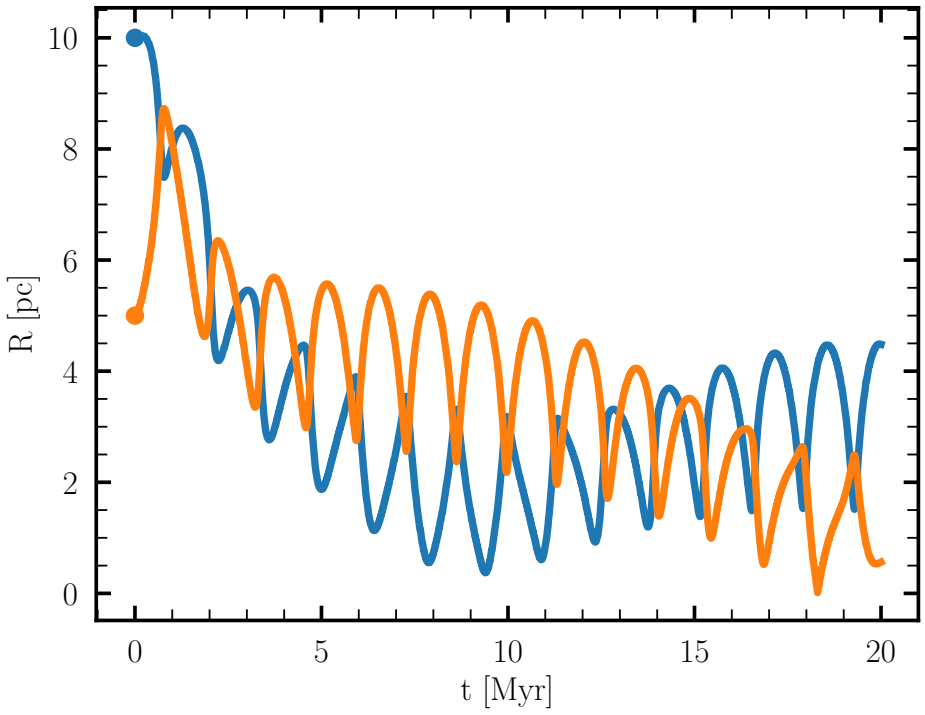

Figure 9.4: Time evolution of Galactocentric distance R of two massive compact objects of $50\,000\,\mathrm{M}_\odot$ each from a distance of 10 pc (blue) and 5 pc (orange). Both curves were integrated using a bridge between `ph4` for integrating the black holes and a background potential with dynamical friction using $\ln(\Lambda) = 3.7$. The sinusoidal variations along the curves are the result of the two black holes not being in a perfect circular orbit. The interference between the two black holes is a consequence of their mutual gravitational interactions. The source code, takes a few seconds to run, can be found at `amuse.examples.imbhs_with_friction`

In this example, we included the dynamical friction as an additional drag force in

the kick of the bridging step. Alternatively, we could include a term in the drift-part of the bridge step. Including additional forces to a bridge can be realized in a similar way by adding an extra term to the kick operator. In this way solving constructing an N-body solver that includes the YORP- and Yarkovski-effects[2] , dynamical friction, tidal evolution, gas-drag, or post-Newtonian dynamics becomes relatively straightforward.

```
13 class CodeWithFriction(bridge.GravityCodeInField):
14
15     def kick_with_field_code(self, particles, field_code, dt):
16         self.LnL = 3.7
17
18         R = particles.position.length()
19         vx = particles.vx.mean()
20         vy = particles.vy.mean()
21         vz = particles.vz.mean()
22         rho = field_code.density(R)
23         vc = field_code.circular_velocity(R)
24         X = 0.34
25         m = particles.mass.sum()
26         ax = -4*np.pi*self.LnL*constants.G**2 * rho*m*(vx/vc**3)*X
27         ay = -4*np.pi*self.LnL*constants.G**2 * rho*m*(vy/vc**3)*X
28         az = -4*np.pi*self.LnL*constants.G**2 * rho*m*(vz/vc**3)*X
29         self.update_velocities(particles, dt, ax, ay, az)
30
31     def drift(self, tend):
32         pass
33
34 def get_system_state_relative_to_SMBH(time, black_holes):
35     t = [] | units.Myr
36     r = [] | units.pc
37     SMBH = black_holes[black_holes.name=="SMBH"]
38     for bi in black_holes-SMBH:
39         t.append(time)
40         r.append((bi.position-SMBH.position).length())
41     return t, r
42
43 def get_system_state(time, black_holes):
44     t = [] | units.Myr
45     r = [] | units.pc
46     for bi in black_holes:
47         t.append(time)
48         r.append(bi.position.length())
49     return t, r
```

Code example 9.4: Example class of a code for adding an extra functionality to a bridge. In this case for calculating the dynamical friction term. Source code is available in `amuse.examples.imbhs_with_friction`.

[2]Named after Ivan Osipovich Yarkovsky (18441902), see also Whipple (1950); Bottke *et al.* (2002).

```
friction_code = CodeWithFriction(
    cluster_gravity,
    (potential,),
    do_sync=True,
    verbose=False,
    radius_is_eps=False,
    h_smooth_is_eps=False,
    zero_smoothing=False,
)
gravity = bridge.Bridge(use_threading=False)
gravity.add_system(cluster_gravity, (potential,) )
gravity.add_code(friction_code)
gravity.timestep = dt_bridge
```

9.2.3 The Classic Bridge

Now that we've seen some of the issues involved in setting up and using a simple Bridge system, we can quite easily create a bridge to reproduce the results of Fujii *et al.* (2007) on the dynamical evolution of a star cluster orbiting the Galactic center. We start by creating a classic Bridge combining a fourth-order Hermite predictor–corrector scheme for the cluster and a Barnes–Hut tree code for the Galaxy. After declaring the codes to be bridged—`ph4` and `bhtree`—we create the bridge solver and add the codes.

```
code1 = Ph4()
code2 = Bhtree()
combined_gravity = bridge.Bridge()
combined_gravity.add_system(code1, (code2,))
combined_gravity.add_system(code2, (code1,))
```

In this case, we have created a bidirectional solver, in which each code interacts with the other.

The simulation is then performed much as before:

```
combined_gravity.timestep = 0.1 | units.Myr
combined_gravity.evolve_model(10 | units.Myr)
```

Figure 9.5 presents an example calculation of the Arches cluster. It is similar to the calculation in Figure 9.3, but now we have adopted a particle representation for the Galactic center region and solved the system using the tree code `BHtree`, in much the same way as presented in Fujii *et al.* (2007). The Arches cluster was initialized with a King model with a dimensionless depth of $W_0 = 3$ and a virial radius of 0.8 pc, as in Figure 9.3. The cluster was placed in a almost circular orbit at a distance of 50 pc from the Galactic center. The latter was represented by a Plummer (1911) sphere with a total mass of 1.6×10^{10} $M_\odot$ and a scale radius of 100 pc (see Section 3.2).

9.2.3.1 Bridging with a Drifter Code

In many cases, it is overkill to run a direct N-body code or tree code in a background potential, particularly if the particles hardly interact, as is the case with planetesimals

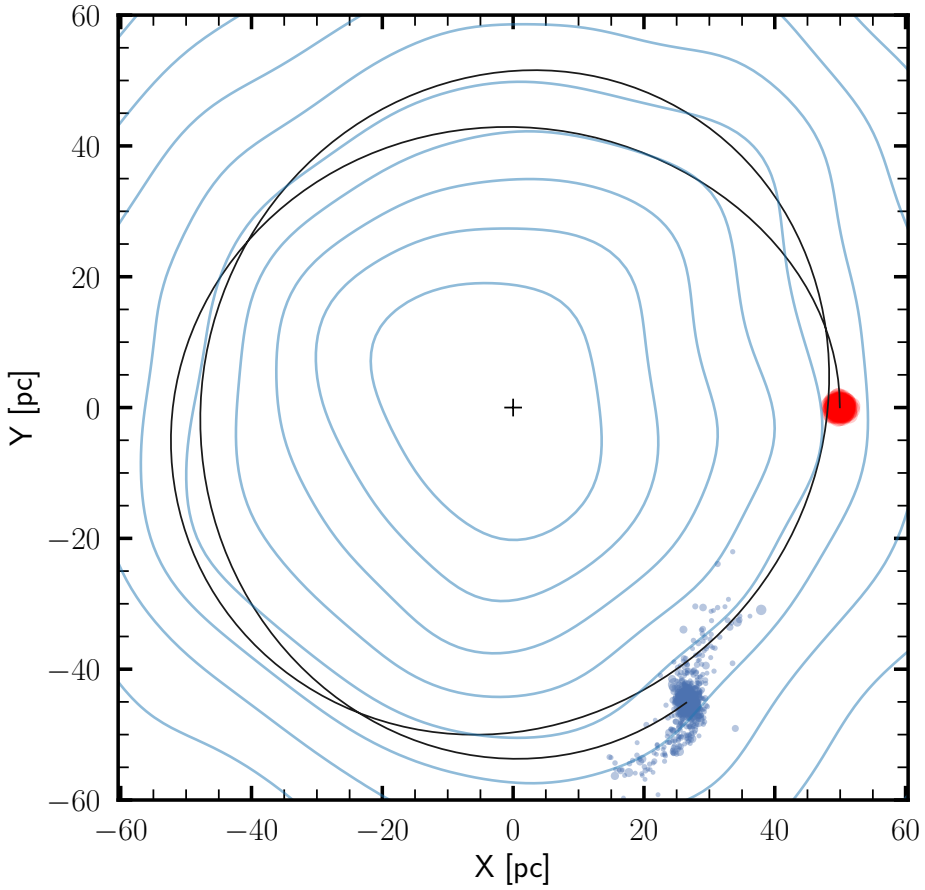

Figure 9.5: Calculation of a star cluster (using `ph4`) in a galaxy (using `BHTree`) for initial conditions comparable to those in Figure 9.3, but now we actively evolve the Galaxy using 10^4 equal mass particles. The star cluster was represented by 1000 point masses drawn from a Salpeter mass function between 0.1 and 100 $M_\odot$. This star cluster, as in Figure 9.3, was born at a distance of 50 pc from the Galactic center (along the X-axis) and with a velocity of 80% of the circular velocity (in the Y-direction). Evolving to an age of 2.5 Myr took about 3 mintes. The source code for the figure is `amuse.examples.gravity_gravity`.

in a planetary system or individual stars in a background galactic potential. In such cases, it is preferable to use a small Python script that just drifts the particles in the background potential, rather than calculating the (negligible) interparticle forces and integrating the orbits. In Code example 9.5 we present an example of such a simple drift code, which we use here to integrate a low-mass star cluster in a semi-analytic Galactic potential. The drift code itself can be used in the same way as any other gravity code. Note that we use the Python `@property` built-in function which allows our small class to remain compatible with past and future versions (see Appendix C.3).

```
class drift_without_gravity:
    def __init__(self, convert_nbody, time=0 | units.Myr):
        self.model_time = time
        self.convert_nbody = convert_nbody
        self.particles = Particles()

    def evolve_model(self, t_end):
        dt = t_end - self.model_time
        self.particles.position += self.particles.velocity * dt
        self.model_time = t_end

    @property
    def potential_energy(self):
        return quantities.zero

    @property
    def kinetic_energy(self):
        return (
            0.5 * self.particles.mass * self.particles.velocity.lengths() ** 2
        ).sum()

    def stop(self):
        pass
```

Code example 9.5: Simple drift code to replace the N-body code in a background potential where interactions between particles are unimportant. Source code is available in `amuse.examples.gravity_drifter`.

The example in Code example 9.5 is far from optimal, and the drifting can easily be done in parallel (for an example, see Appendix A.7.4) or using a graphical processing unit. This would require somewhat more elaborate functions, and you might want to explore the possibility of using

```
from amuse.community.fastkick import Fastkick
```

which is a more efficient and optimized implementation of the kick operation using a GPU. An example of the use of `FastKick` is given in Section 9.3.2.1.

Although the script (which is referenced in Code example 9.5) uses a different model for the background Galactic centrer potential adopted for producing Figure 9.3, the results are also different due to the lack of self-gravity among the stars. As a consequence, the stars stay together longer and the cluster disperses more slowly. We do not recommend using this script for simulating star clusters in a background potential, but it is a handy tool for integrating the orbits of individual (non-self-interacting) particles in the galactic potential.

9.3 Bridging Other Codes

Bridging high-precision gravity with low-precision gravity, or gravity with a background field, is a very effective coupling strategy. Bridge is guaranteed to work in these cases because we can write down a common Hamiltonian for the coupled systems, but Bridge is much more versatile than this. Coupled systems for which we cannot write down a

coupled (or any) Hamiltonian can also be integrated using the classic bridge scheme. We could just as easily have replaced the analytic calculation by another N-body code or a hydrodynamical solver, or bridged one hydrodynamical solver with another, or included radiative transfer. One needs a little courage to do this, because we are unaware of any formal proof that this procedure results in a unique or "correct" solution to the coupled problem. On the other hand, resolution and convergence tests indicate that the solutions obtained in this way do appear to be reliable and accurate, and therefore suitable for scientific interpretation. We leave a complete proof to the brave reader, and should one be forthcoming we will be happy to include it in future editions!

The range of applications of Bridge is up to the imagination of the researcher. In the next section, we present a few bridge topologies you might want to experiment with.

9.3.1 Bridge Hierarchies and Hierarchical Bridges

The simplest possible bridge can be realized by integrating two single particles, such as the Sun and the Earth, and following each—not with a regular integrator, but using Bridge directly. Classic Bridge works as a second-order Verlet–leapfrog integrator, which is perfectly adequate for integrating the solar system, or (almost) any N-body system. An example of how to integrate such a two-body system is given in Code example 9.6.

```
23      star_gravity = Ph4(converter)
24      star_gravity.particles.add_particle(ss[0])
25
26      planet_gravity = Ph4(converter)
27      planet_gravity.particles.add_particle(ss[1:])
28
29      channel_from_star_to_framework = star_gravity.particles.new_channel_to(ss)
30      channel_from_planet_to_framework = planet_gravity.particles.new_channel_to(ss)
31
32      gravity = bridge.Bridge(use_threading=False)
33      gravity.add_system(star_gravity, (planet_gravity,))
34      gravity.add_system(planet_gravity, (star_gravity,))
```

Code example 9.6: Bridge construction for integrating the Sun–Earth system.

In this example, we declare a bridge with two components: one for the Sun, which is affected by Earth's gravity, and one for Earth, which is affected by the gravity of the Sun. The eventual integration is carried out in a loop over time:

```
while time < time_end:
    time += time_step
    gravity.evolve_system(time)
```

in exactly the same way as a more usual integrator or N-body code would be used within the framework. It may seem strange to start an integrator (in this case `ph4`) and have it integrate a single particle (and indeed, some integrators won't even let you do this), but here the integrator simply performs the drift step and defines `get_gravity_at_point()`,

letting the bridge complete the integration of the equations of motion. It probably makes far more sense to use the drifter code in Code example 9.5 instead of the direct N-body code.

We could make this example even stranger by adding the Moon as a third particle and constructing a bridge for all three elements (Sun, Earth, and Moon) as in Code example 9.7, a simple code to integrate a three-body system using second-order bridge. As you might imagine, this is not a very efficient way to integrate the Sun–Earth–Moon system. In fact, for comparable overall accuracy, our three-body bridge is almost 600 times slower than a direct integration using `ph4`, mainly because it is all done in Python, whereas `ph4` is compiled C++ (Stroustrup, 2013).

```
    star_gravity = Ph4(converter)
    star_gravity.particles.add_particle(star)

    planet_gravity = Ph4(converter)
    planet_gravity.particles.add_particle(planet)

    moon_gravity = Ph4(converter)
    moon_gravity.particles.add_particle(moon)

    channel_from_star_to_framework = star_gravity.particles.new_channel_to(ss)
    channel_from_planet_to_framework = planet_gravity.particles.new_channel_to(
        ss)
    channel_from_moon_to_framework = moon_gravity.particles.new_channel_to(ss)

    time = 0 | units.yr
    write_set_to_file(ss, filename,
                      timestamp=time, overwrite_file=True)

    gravity = bridge.Bridge()
    gravity.add_system(star_gravity, (planet_gravity, moon_gravity))
    gravity.add_system(planet_gravity, (star_gravity, moon_gravity))
    gravity.add_system(moon_gravity, (star_gravity, planet_gravity))
```

Code example 9.7: Bridge construction for integrating the Sun–Earth–Moon system. We call this method the multi-bridge; its topology is at the bottom left of Figure 9.6.

We could extend the list of arguments indefinitely, even hierarchically, making the system quite complex. In Code example 9.8, we demonstrate how a hierarchy of level two can be achieved. The bridge for realizing the interaction between Sun and Earth is again bridged with the Moon, leading to a single composite bridge that integrates the entire system. In this case, the `ph4` instantiations take care of the `get_gravity_at_point()` used in the various bridges. This implementation is even slower than the example in Code example 9.6, but no matter—its purpose is to illustrate the method.

The relative performance of the various Bridge implementations just described is presented in Table 9.1. The three bridged methods use `ph4` in various ways, in combination with a bridge. The hierarchical bridge method uses three nested bridges.

Performance may be much better for other bridge topologies or different problems, but the key message here is that Bridge allows considerable flexibility that may com-

```
37      sp_gravity = bridge.Bridge()
38      sp_gravity.add_system(moon_gravity, (planet_gravity,))
39      sp_gravity.add_system(planet_gravity, (moon_gravity,))
40
41      gravity = bridge.Bridge()
42      gravity.add_system(sp_gravity, (star_gravity,))
43      gravity.add_system(star_gravity, (sp_gravity,))
```

Code example 9.8: Hierarchical bridge for integrating the Sun–Earth–Moon system. The majority of the code is identical to Code example 9.7 except that the last four lines there should be replaced with these lines to initialize the hierarchical bridge. This hierarchical-bridge topology is depicted on the bottom right of Figure 9.6.

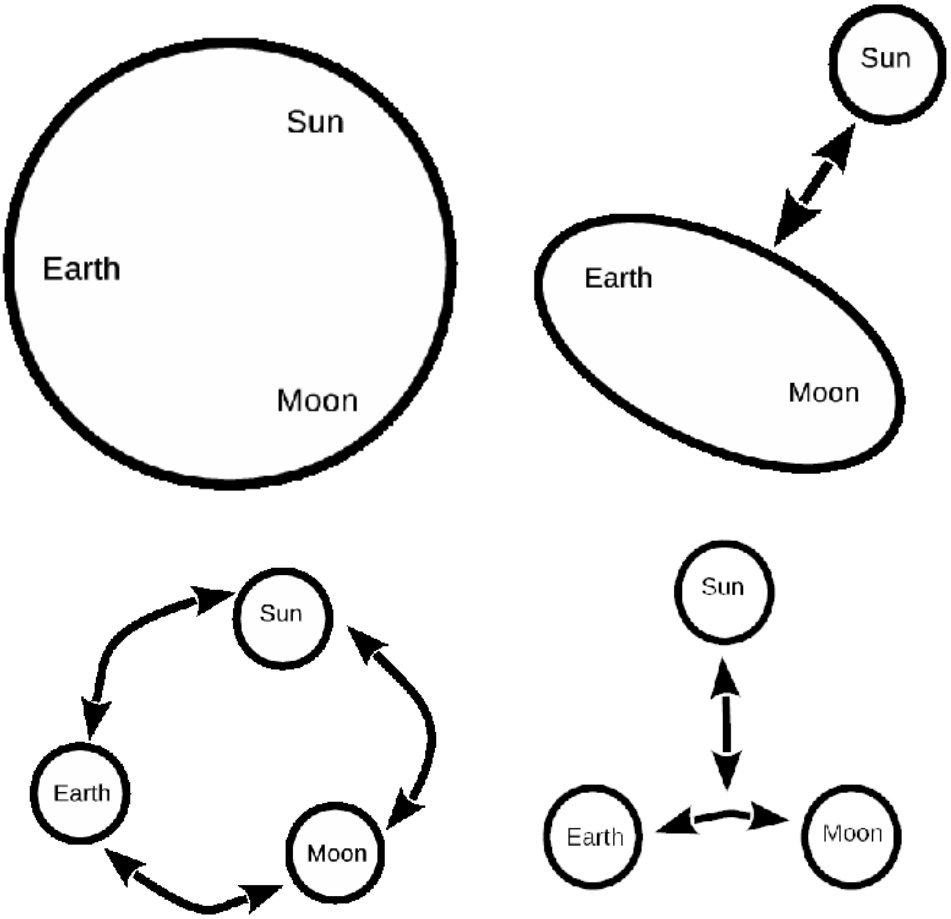

Figure 9.6: The various `Bridge` and `ph4` instantiations in Table 9.1. Top left indicates the use of a single N-body integrator for all three particles. At top right we present a two-way bridge (indicated by the two-pointed arrow) between an N-body realization for only the Sun, and separately for Earth and the Moon. At bottom left we show a three-body bridge, in which each particle is separately handled by an N-body integrator, and each is separately added to a bridge, see Code example 9.7. The bottom right depicts the hierarchical bridge, see Code example 9.8.

pensate for (some) loss of performance in many circumstances. Of course, we don't want performance to drop by a factor of 600, but in practical situations, when running computer-intensive production-quality simulations, the overhead of such complex bridging strategies is actually rather small—generally not more than a few percent.

code	code example	100 yr integration		
		$t_{\rm wall}$ [s]	$\mathrm{d}\Delta\mathrm{E}_{\rm max}$	$\Delta\mathrm{E}_{\rm tot}$
ph4		4.40	4.21×10^{-5}	1.61×10^{-6}
ph4 normalized	4.1	1	1	1
Bridge with ph4	9.6	4.0	0.52	40
Three-body bridge	9.7	8.4	0.57	36
Hierarchical bridge	9.8	12.5	0.57	36
TBB 4th order	A.4	13	1.13	62
TBB 6th order	A.4	4.4	0.52	31
TBB 8th order	A.4	40	0.78	47

Table 9.1: Performance of various Bridge implementations of the Sun–Earth–Moon problem. See also Figure 9.7. Values are relative to `ph4` as a standard, using a `timestep_parameter` of 0.15 (first line), which gives 4.40 s for the wall clock time, $\Delta\mathrm{E}_{\rm max} = 4.21 \times 10^{-5}$ for the energy error, and $\Delta\mathrm{E}_{\rm tot} = -1.61 \times 10^{-6}$ for the maximum energy error per output time step (0.1 yr) and the total energy error for integration over 20 years, respectively. In the following lines we normalize the performance of various bridged integrators (using `ph4`), including higher order bridges. The latter are generally more expensive, but have a favorable energy error per bridge time step. The various scripts are in `amuse.examples` and are called `no_bridge_ph4_integrator`, `three_body_bridge` (in the table named TBB), `three_body_bridge_hierachical`, and `three_body_bridge_n_order`.

Figure 9.7 compares the evolution of the energy error $(E - E_0)/E_0$ of `ph4` with those of the three bridge topologies listed in Table 9.1. To make the comparison more honest, we degraded the `ph4` time step by using a value of 0.6 for `timestep_parameter`. (This parameter is normally set to 0.1, which would have given a relative energy error better than 10^{-12} per step.) It is interesting to note that the energy error in the non-symplectic fourth-order Hermite integrator `ph4` accumulates, whereas the energy errors in the symplectic bridged systems show no secular evolution. The performance of the fourth- and sixth-order bridge methods is quite poor. At present, these are implemented in Python, which is rather inefficient. Porting these to compiled C++ or FORTRAN has not been a priority because, normally, little time is spent in the bridge, except for these somewhat contrived examples.

The example of a bridge coupling or a hierarchical bridge may appear a bit artificial for integrating the Sun–Earth–Moon system, but for other applications, the strict separation of codes, while combining them either sequentially or hierarchically, creates many interesting possibilities. In that sense, the timings listed in Table 9.1 are not representative of the ultimate performance of a more complex simulation in which each of the individual modules is more compute intensive. We encourage the reader to experiment with bridge until bridging is as familiar an operation as eating chicken pie (Cooder, 1972).

If a higher-order bridge is needed, it can be easily implemented by

```
from amuse.couple import bridge
import amuse.ext.composition_methods as CM
gravity = bridge.Bridge(use_threading=False, method=CM.SPLIT_4TH_S_M4)
```

Here, `composition_methods` provides a library for different bridge-coupling strategies (see Appendix A.4.2). The first free parameter in such complex bridge couplings is the number of evaluations of the same bridge step (m). For an example, see Equa-

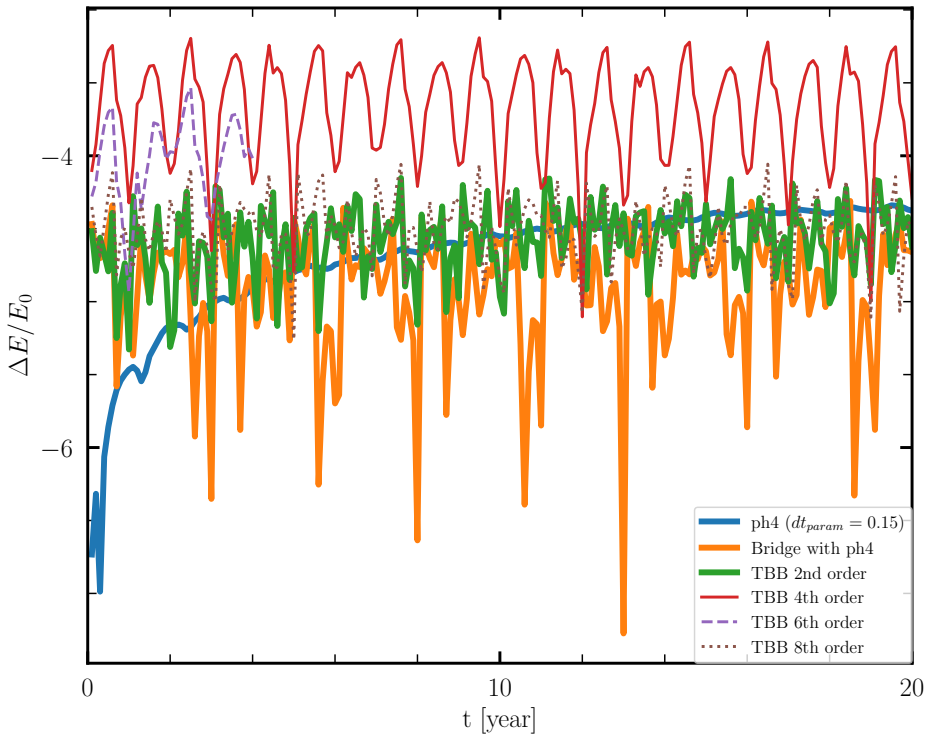

Figure 9.7: Evolution of the energy error while integrating the Sun–Earth–Moon system using a variety of bridge integrators. We use ph4 with a time-step parameter $\eta = 0.15$ (blue curve) as a reference, which clearly shows a secular increase of the energy error in the calculation. The orange curve gives Earth-moon in ph4 but subsequently bridged with the Sun. The other curves are calculated with all particles individually bridged with a 2nd, 4th, 6th and 8th order bridge. Interestingly, the 2nd-order bridge performs best for this case. The source code for reproducing this figure is `amuse.examples.plot_sun_and_earth_and_moon`, but the worker codes are presented in Figure 9.6.

tion (A.10), where we describe a sixth-order bridge with $m = 11$ force evaluations per step. The other parameter is the number of symplectic operations (S, see also Appendix A.4). The argument `method` in the `Bridge` declaration indicates the specific bridge scheme to use, in this case a fourth-order method with $m = 4$; for a sixth-order method one might instead adopt `SPLIT_6TH_SS_M13` (see Table A.16 in Appendix A.4.2).

An example of a higher-order bridge is given in `amuse.examples.three_body_bridge_order4m4`. The default `method` is set to `LEAPFROG`, the classic second-order bridge. Note that, as discussed in Appendix A.4, for codes to be bridgeable to an order higher than second, they must be able to run with negative time steps (stepping backward in time). Not all codes in AMUSE comply with this requirement: those that cannot accommodate negative time steps, such as the Barnes–Hut tree code, can only be bridged to second order.

9.3.2 Bridging Gravity with Hydrodynamics

Many hydrodynamics solvers allow the possibility of adding dark matter particles (called `dm_particles` in AMUSE, as supposed to `gas_particles`; see Section 6.3.3). The `dm_particles` have a gravitational effect on the rest of the system, but no hy-

drodynamical interactions. But the numerical solver used to integrate the equations of motion for `dm_particles` is generally identical to the hydrodynamics solver, which typically is a second-order Verlet integrator with a hierarchical tree to evaluate the forces. These integrators are not very accurate, and they are generally not used for collisional N-body systems. Collisional gravitational problems need more accurate N-body integrators, of which many are available in AMUSE. In principle, if we could couple the high-order gravity modules with the hydrodynamics modules, we could have the best of both worlds: the high accuracy of a fourth-order predictor–corrector Hermite integrator (for example `ph4`), along with the hydrodynamics solver of choice (such as `Fi`). In AMUSE we can do this using Bridge.

Bridging gravity with hydrodynamics is not much different than bridging gravity with gravity, as discussed in Section 9.1. Essentially the same bridge technique can be adopted, and the script will not look much different from one in which two or more gravitational dynamics codes are coupled. We could replace `ph4` in Code example 9.1 with any of the SPH codes in AMUSE without any further changes. Code example 9.9 presents an example of a coupling between an SPH code and a high-order direct N-body code. Here, we generalize the initialization of the hydro and the gravity codes by making them into classes. (For a brief overview of Python classes, we refer to Appendix C.3.7.) This requires a base class, as presented in Code example 9.10, and two specific classes for the gravity and hydrodynamics solvers. These derived classes are presented in Code examples 9.11 and 9.12. Section 9.4.3 presents an example in which we integrate the equations of motion of three stars using an N-body code, while we address the gas dynamics with an SPH code.

```
def gravity_hydro_bridge(Mprim, Msec, a, ecc, t_end, n_steps,
                         Rgas, Mgas, Ngas):

    stars = new_binary_from_orbital_elements(Mprim, Msec, a, ecc,
                                             G=constants.G)
    eps = 1 | units.RSun
    gravity = Gravity(ph4, stars, eps)

    converter = nbody_system.nbody_to_si(1.0|units.MSun, Rgas)
    ism = new_plummer_gas_model(Ngas, convert_nbody=converter)
    ism.move_to_center()
    ism = ism.select(lambda r: r.length()<2*a,["position"])
    hydro = Hydro(Fi, ism, eps)
    model_time = 0 | units.Myr
    filename = "gravhydro.hdf5"
    write_set_to_file(stars.savepoint(model_time), filename, 'amuse')
    write_set_to_file(ism, filename, 'amuse', append_to_file=True)

    gravhydro = bridge.Bridge(use_threading=False)
    gravhydro.add_system(gravity, (hydro,))
    gravhydro.add_system(hydro, (gravity,))
    gravhydro.timestep = 2*hydro.get_timestep()

    while model_time < t_end:
        orbit = orbital_elements_from_binary(stars, G=constants.G)
        dE_gravity = gravity.initial_total_energy/gravity.total_energy
        dE_hydro = hydro.initial_total_energy/hydro.total_energy
        print(("Time:", model_time.in_(units.yr), \
               "ae=", orbit[2].in_(units.AU), orbit[3], \
               "dE=", dE_gravity, dE_hydro))

        model_time += 10*gravhydro.timestep
        gravhydro.evolve_model(model_time)
        gravity.copy_to_framework()
        hydro.copy_to_framework()
        write_set_to_file(stars.savepoint(model_time), filename, 'amuse',
    append_to_file=True)
        write_set_to_file(ism, filename, 'amuse', append_to_file=True)
        print("P=", model_time.in_(units.yr), gravity.particles.x.in_(units.au))
    gravity.stop()
    hydro.stop()
```

Code example 9.9: Combined solver for gravity and hydrodynamics using the generic class structure presented in Code example 9.10 (see `amuse.examples.gravity_hydro` for the full code example).

```
class BaseCode:
    def __init__(self, code, particles, eps=0|units.RSun):

        self.local_particles = particles
        m = self.local_particles.mass.sum()
        l = self.local_particles.position.length()
        self.converter = nbody_system.nbody_to_si(m, l)
        self.code = code(self.converter)
        self.code.parameters.epsilon_squared = eps**2

    def evolve_model(self, time):
        self.code.evolve_model(time)
    def copy_to_framework(self):
        self.channel_to_framework.copy()
    def get_gravity_at_point(self, r, x, y, z):
        return self.code.get_gravity_at_point(r, x, y, z)
    def get_potential_at_point(self, r, x, y, z):
        return self.code.get_potential_at_point(r, x, y, z)
    def get_timestep(self):
        return self.code.parameters.timestep
    @property
    def model_time(self):
        return self.code.model_time
    @property
    def particles(self):
        return self.code.particles
    @property
    def total_energy(self):
        return self.code.kinetic_energy + self.code.potential_energy
    @property
    def stop(self):
        return self.code.stop
```

Code example 9.10: Generic base class for a single code.

```
class Gravity(BaseCode):
    def __init__(self, code, particles, eps=0|units.RSun):
        BaseCode.__init__(self, code, particles, eps)
        self.code.particles.add_particles(self.local_particles)
        self.channel_to_framework \
            = self.code.particles.new_channel_to(self.local_particles)
        self.channel_from_framework \
            = self.local_particles.new_channel_to(self.code.particles)
        self.initial_total_energy = self.total_energy
```

Code example 9.11: Derived class for gravity includes channels.

```
class Hydro(BaseCode):
    def __init__(self, code, particles, eps=0|units.RSun,
                 dt=None, Rbound=None):
        BaseCode.__init__(self, code, particles, eps)
        self.channel_to_framework \
            = self.code.gas_particles.new_channel_to(self.local_particles)
        self.channel_from_framework \
```