Chapter 2

Initial conditions

īn nīz bogzarad
This too shall pass.

Solomon – Israel Folklore Archive #126

Any simulation requires a starting point. We call these the initial conditions. Some initial conditions are simple, some are extraordinary complex. The simplest ones are often the most interesting, and lead to the clearest understanding of the complex phenomena that follow through calculation. Many final conditions are eventually turn into initial conditions. In the end, it all boils down to the appropriate initial conditions. In astrophysics, the ultimate initial conditions are probably the beginning of time and the Universe, whatever that is.

Every simulation in AMUSE requires initial conditions to start a calculation. Typically these entail choosing dynamical quantities such as particle masses, positions, and velocities, stellar properties such as radii and luminosities, or gas properties such as density, temperature, and composition. These quantities are often the result of some prior calculation, or represent a restart from a previously stored state. In principle, any mathematically consistent set of initial choices is a valid way to start a simulation. In practice, however, we generally want to use some reasonable, physically motivated starting point for further study.

Some initial conditions are realistic, but most are not - and most of the time that is okay. Realistic initial conditions are the goal of (almost) all researchers, but they are hard to find and generally even harder to explain. Most studies start with somewhat unrealistic initial conditions, with the expectation (or at least the hope) that they capture the fundamental physics necessary to study the desired phenomenon, and that the details of the simulation will rapidly drive the system into some physically more meaningful configuration. In follow-up studies, we can then improve on the earlier choice of initial conditions, say by covering a larger part of parameter space or by zooming in on a region of interest. In some cases, for example core collapse in an idealized equal-mass star cluster (see Section 3.1.3.2), the evolution is quite insensitive to the details of the initial conditions. In others however, such as star formation in an interstellar molecular cloud (Chapter 5), the outcome depends crucially on the initial state.

A simple example of unrealistic but widely accepted initial conditions are those used in gravitational N-body simulations of star clusters. Such simulations generally start by setting the number of stars N, and distributing them in space with specified radial (spherically symmetric) density and (truncated Maxwellian) velocity profiles. Most modern studies of star clusters incorporate some sort of (parametrized) stellar evolution, so the initial conditions must then also include stellar masses and radii. The entire system is then scaled (usually by adjusting the velocities) to virial equilibrium (see Section 3.1.2), before high-precision N-body integrators are used to study the dynamical evolution of the system. From an astronomical point of view, these initial conditions are not correct, but they can still be very useful in the study of gravitational dynamics.

More realistic initial conditions for this problem might start with an interstellar gas cloud, which collapses and fragments to form stars. Once massive stars begin to shine, they produce winds and radiation that cause the remaining gas to be blown away, terminating further star formation and locally leaving behind a star cluster. In this case, no ad hoc specifications of the positions and masses of cluster stars are required. But of course, this only shifts the problem back in time, as the initial conditions for the gas simulation now must specify the (clumpy) density distribution and the (turbulent) motion of the cloud. Those "initial conditions" in turn depend on the large-scale flow of gas in the galaxy, and so on. The fact is that there are no "true" initial conditions for this problem, just convenient milestones in a continuous process. Most generally adopted initial conditions have problems, but doing much better can turns garbage into science.

AMUSE contains an extensive (and growing) collection of codes to generate initial conditions. Very often one code provides the conditions from which another code continues. For example, when two stars collide, a stellar evolution code might provide the initial density and temperature profiles for the two colliding stars, while a gravitational dynamics code provides their relative positions and velocities. In this way two codes are used to generate the conditions for starting a third code that simulates the hydrodynamics of the collision. In the next subsections we discuss some common initial conditions used in astrophysical studies.

2.1 Creating particle distributions in space

Data handling is a critical aspect of AMUSE. It is used to transfer data from one code to another, and for eventual data processing and analysis. The most common data archiving method is in the form in a particle set. This is basically a list (or array) of adjustable data containers. The fundamental ingredient in a particle set is the `key`.

```
from amuse.datamodel import Particles
particles = Particles(N)
print(particles)
```

```
                  key
                    -
====================
 5512314219721684995
 8353633730176500107
 9653245888285946456
 9457276936209163492
14505666692805646583
14003077150553218257
====================
```

2.1.1 Initial condition for gravitational dynamics

In large systems, initial conditions are not specified exactly, and we must start with a random realization of a distribution function. In self-gravitating simulations of star clusters, the Plummer (1911) model is often adopted as an initial density profile. In AMUSE, a particle set representing a Plummer sphere of N stars can be generated with

```
particles = new_plummer_model(N)
```

The Plummer model represents a self-consistent equilibrium solution to Poisson's equation, and so is a valid description of an N-body system. But aside from the fact that it is simple and has no free dimensionless parameters in its description, it has no special claim to fame as a cluster model. Nevertheless, it is a very common choice.

```
9  from matplotlib.pyplot import show, xlim, ylim, figure
10 from amuse.plot import scatter, xlabel, ylabel
11 from amuse.ic.plummer import new_plummer_model
12
13
14 def plot_plummer_model(number_of_particles=1000):
15     figure(figsize=(5, 5))
16     bodies = new_plummer_model(number_of_particles)
17     scatter(bodies.x, bodies.y)
18     xlim(-1, 1)
19     ylim(-1, 1)
20     xlabel("X")
21     ylabel("Y")
22     show()
```

Listing 2.1: Routine to plot a Plummer sphere. Code fragment from `amuse.examples.plot_plummer`.

Some initial condition generators are not considered standard, and require a community code to be built/installed before usage. For example, for "fractal" initial conditions (Goodwin & Whitworth, 2004), we must have the `amuse.community.fractalcluster`

package available. If this is not the case, using the `new_fractal_cluster_model` function will result in an error:

```
ModuleNotFoundError: No module named 'amuse.community.fractalcluster'
```

Figure 2.1 presents a small collection of various initial realizations common in N-body simulations: the Plummer sphere, a King model (with structure parameter $W_0 = 9$; King, 1966), the top view of a galactic disk plus bulge model (for which we adopted `Halogen` with parameters $\alpha = 1$, $\beta = 5$, $\gamma = 0.5$; Zemp *et al.*, 2008; Zemp, 2014), and a fractal distribution (with fractal dimension 1.6; Goodwin & Whitworth, 2004).

The various models are generated with calls similar to those discussed in Section 2.1. For the King model we write:

```
particles = new_king_model(N, W0=9.0)
```

The Galaxy disk model is constructed with

```
particles = new_halogen_model(N, alpha=1.0, beta=5.0, gamma=0.5)
```

The fractal distribution is generated with

```
particles = new_fractal_cluster_model(N=N, fractal_dimension=1.6)
```

For the latter (and for some other special initial condition generators) one would have to include the appropriate file, which can be loaded with

```
from amuse.ic.fractalcluster import new_fractal_cluster_model
```

A circum-stellar disk can be generated using

```
from amuse.ext.protodisk import ProtoPlanetaryDisk
model = ProtoPlanetaryDisk(
    N,
    densitypower=1.5, Rmin=0.1, Rmax=1,
    q_out=1.0, discfraction=0.1,
).result
model.rotate(0., np.pi/6, np.pi/6)
```

Here the arguments include the number of particles (N), the slope of the power-law projected density profile, the inner and outer disk radius (if defined in units these values are using the converter), the Toomre Q-parameter, and the mass fraction (here called disk-fraction, indicating that the disk is 10% of the star's mass, which is unity in dimensionless units). The routine `ProtoPlanetaryDisk` returns a class whose parameter `result` is an SPH particle set. After the disk is generated we rotate it over two angles with $\pi.6$ radians to result in the bottom left panel in Figure 2.1.

The composite King model can be generated by first making a King model for the center of masses of the stars, and subsequently make as many King models as center of masses were generated. Here is an example of 5 King models with /5th of the total number of stars each, as presented in the bottom right panel in Figure 2.1.

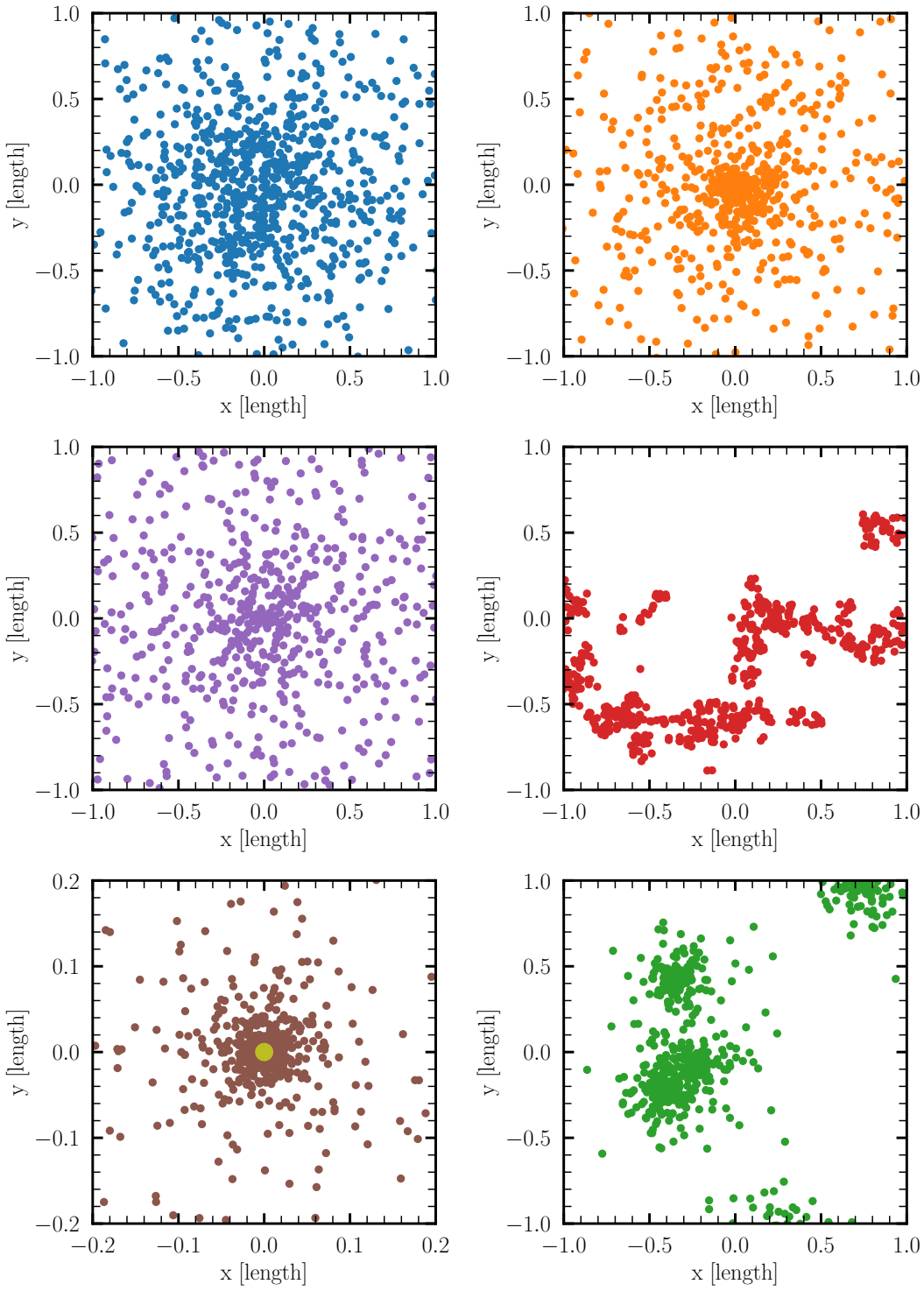

Figure 2.1: The inner parts, spanning (1 by 1 Hénon units (Hénon, 1971; Heggie & Mathieu, 1986)), of projected distributions drawn from Plummer model (top left), a King model with $W_0 = 9$ (top right), Halogen Galaxy model with $\alpha = 1$, $\beta = 5$, $\gamma = 0.5$ (middle left), a cluster with fractal dimension 1.6 (middle right), an Oort cloud with a central star (bottom left), and a hierarchical King model, each for $N = 1000$ particles. The source for making the Plummer distribution is presented in Listing 2.1. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/plot_density_distributions.py`.

```
model_com = new_king_model(5, W)
model_com.position *= 0.7
for i in range(5):
    model = new_king_model(int(N/5), W)
    model.position *= 0.1
    model.position += model_com[i].position
```

2.1.2 Initial condition for hydrodynamics

Like particle distributions for gravitational dynamical simulations, hydrodynamical simulations also require input density and other distributions. AMUSE provides most support for smoothed particle hydrodynamics (SPH) codes (see Chapter 5), and contains a variety of distribution-generating functions. Listing 2.2 illustrates the conversion of a 1-dimensional stellar evolution model to a 3-dimensional hydrodynamical particle distribution.

```
20 def stellar_model(number_of_particles, mass, time=0.0 | units.Myr):
21     star = Particle(mass=mass)
22     stellar_evolution = Evtwin()
23     se_star = stellar_evolution.particles.add_particle(star)
24     print(f"Evolving {star.mass} to t= {time.in_(units.Myr)}")
25     stellar_evolution.evolve_model(time)
26     print(f"Stellar type: {stellar_evolution.particles.stellar_type.number}")
27     print("Creating SPH particles from the (1D) stellar evolution model")
28     sph_particles = convert_stellar_model_to_sph(
29         se_star, number_of_particles
30     ).gas_particles
31     stellar_evolution.stop()
32     return sph_particles
```

Listing 2.2: Converting an 1–D stellar evolution model to a 3–D SPH particle representation. Code fragment from `amuse.examples.plot_hydro_density_distributions`.

In Listing 2.2, in the lines 25-27 the parameter `gas_particles` is returned from the function `convert_stellar_model_to_sph`. This is common practice when the called function returns a class, in this case a particle set. In this case, the return values is a class containing the particle set `core_particle`, its radius (`core_radius`), and a `gas_particle` set. The resulting hydrodynamical models are plotted in Figure 2.2 (left panel); Listing 2.3 shows a portion of the plotting code.

```
def plot_zams_stellar_model(number_of_particles, mass):
    sph_particles = stellar_model(number_of_particles, mass)
    figure = plt.figure(figsize=(6, 6))
    ax = figure.add_subplot(111)
    sph_particles_plot(
        sph_particles,
        min_size=500,
        max_size=500,
        alpha=0.01,
        view=(-2, 2, -2, 2) | units.RSun,
    )
    ax.set_facecolor("white")
    ax.set_xlabel(r"x [R$_\odot$]")
    ax.set_ylabel(r"y [R$_\odot$]")

    save_file = "stellar_2MSunZAMS_projected.pdf"
    plt.savefig(save_file)
    print(f"Saved figure in file {save_file}\n")
    plt.show()
```

Listing 2.3: Plotting the 3–D particle representation using the AMUSE `native_plotting` module. Code fragment from `amuse.examples.plot_hydro_density_distributions`.

```
def gmc_model(number_of_particles, mass, radius):
    converter = nbody_system.nbody_to_si(mass, radius)
    sph_particles = molecular_cloud(
        targetN=number_of_particles, convert_nbody=converter
    ).result
    sph = Fi(converter)
    sph.gas_particles.add_particle(sph_particles)
    sph.evolve_model(1 | units.day)
    channel = sph.gas_particles.new_channel_to(sph_particles)
    channel.copy()
    sph.stop()
    return sph_particles
```

Listing 2.4: Converting a molecular cloud model to a 3–D SPH particle representation. Code fragment from `amuse.examples.plot_hydro_density_distributions`.

We don't have to start from a stellar model, of course. Listing 2.4 illustrates the creation of a 3-dimensional hydrodynamical particle distribution describing a turbulent molecular cloud. As discussed in Section 2.1.1 in relation to the function `ProtoPlanetaryDisk()`, `molecular_cloud()` also returns a class whose parameter result contains the SPH particle set:

```
from amuse.ic.molecular_cloud import new_molecular_cloud
sph_particles = new_molecular_cloud(
    targetN=number_of_particles, convert_nbody=converter,
)
```

Both routines can use the argument `converter_nbody` converter to scale between dimension-less N-body units and physical units. The result is a molecular cloud represented by N particles. The particles will have all have the same mass (`mass`) and internal energy (`u`), but position and velocity consistent with a homogeneous gas sphere with a turbulent velocity field. (The system is integrated for a short time after creation

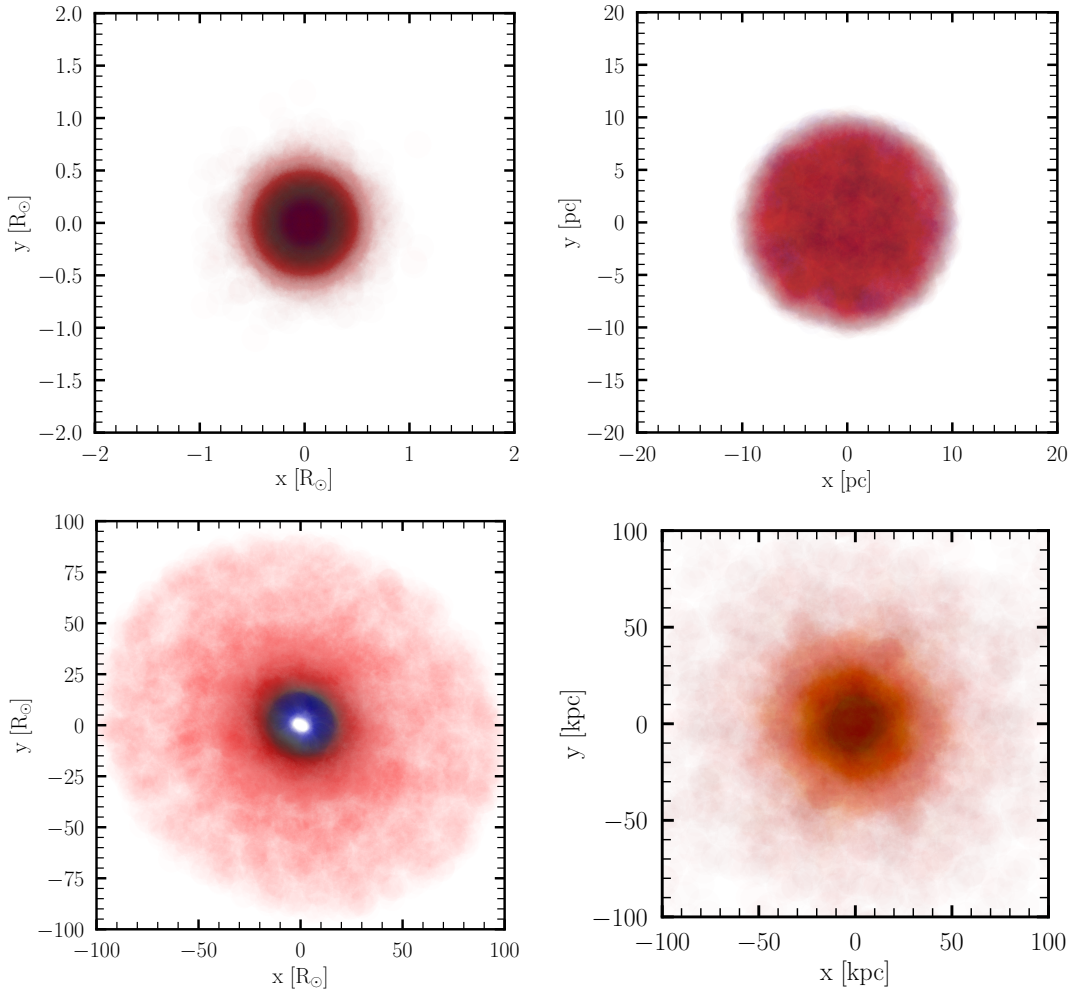

Figure 2.2: **[interactive switch for temperature/density/velocity/... – Steven]** Some initial particle distributions for SPH codes. The top-row left panel shows a 2 solar mass star, the right a homogeneous spherical distribution with a turbulent velocity field, as is often used in simulations of molecular clouds. Particles are colored by their internal energy (for the star, from red for 0.048 $(\mathrm{km/s})^2$ and blue for 1.82 $(\mathrm{km/s})^2$). The bottom row shows a slightly tilted circum-stellar disk with a total mass of 0.1% of the star's mass (left), and a $10^{10}\,\mathrm{M}_\odot$ galaxy model. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/plot_hydro_density_distributions.py`.

to reduce random initial noise in the gas distribution.) The resulting density distribution is presented in Figure 2.2 (right panel). We will explore such models in more depth in Chapter 5.

We will not dwell on more advanced code coupling strategies here, and similarly defer discussion of radiative transport to Chapter. We postpone these advanced topics because they require multiple codes to be operational at the same time. They will be discussed toward the end of this book, in Chapters 6 and 8.

2.2 Creating mass distributions

The default for `new_plummer_model` and `new_fractal_cluster_model` is to create stars of equal mass, but usually we want stars to have a range of masses, drawn from some distribution. This distribution is known as the initial mass function, and is one of the most important quantities in stellar astrophysics.

The initial mass function is often represented as a power-law, meaning that the number of stars dN with masses between m and $m + dm$ is given by

$$\frac{dN}{dm} = A\, m^{\alpha} \tag{2.1}$$

for $M_{\rm min} < M < M_{\rm max}$. The choice $\alpha = -2.35$ (Salpeter, 1955) is so common that it is has its own standard function call in AMUSE:

```
mass_zams = new_salpeter_mass_distribution(N, mass_min=Mmin, mass_max=Mmax)
```

returns an array of N masses randomly selected between `Mmin` and `Mmax`, distributed according to a Salpeter distribution, a power-law distribution with exponent $\alpha = -2.35$. The resulting mass function is presented in Figure 2.3. The naming of this mass `mass_zams` seems somewhat arcane, but refers to the mass of a star at the moment of hydrogen ignition (generally called the birth of the star) which is generally referred to as the zero-age main-sequence (see also Appendix E.1).

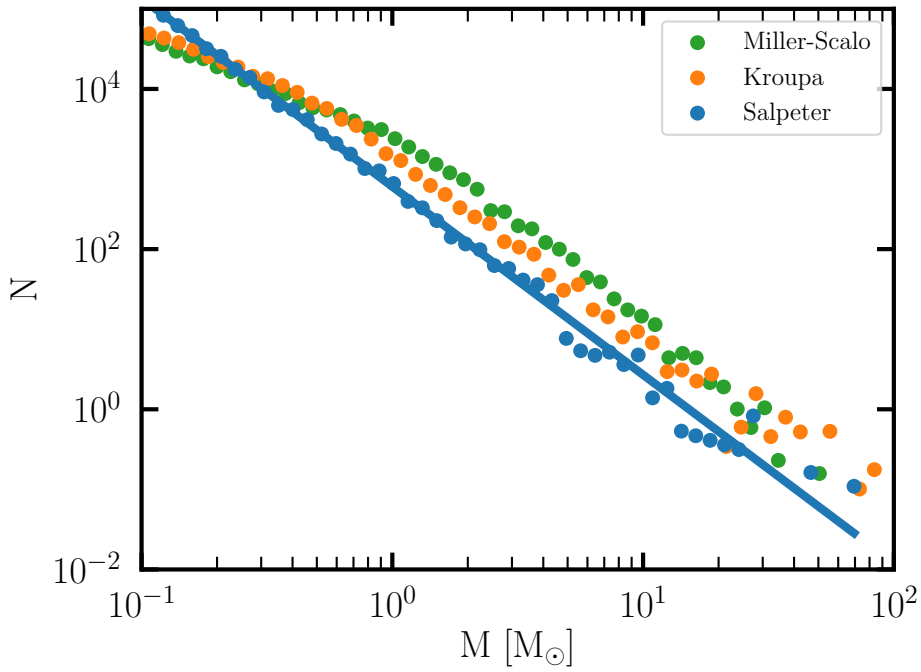

Figure 2.3: Initial mass function for 10^4 stars between $0.1\,M_\odot$ and $100\,M_\odot$. The Salpeter mass function is overplotted with an analytic power-law with a slope of -2.35 (blue). Two other often used mass functions are shows in green and orange. The latter two are described by Miller & Scalo (1979) (in green) and Kroupa (2001) (orange). An example of how this is plotted is presented in Listing 2.5. The script to generate this figure can be found in `amuse.examples.salpeter`.

```
def plot_mass_function(masses, ximf, color, label, n_bins=51):
    "Plots a histogram of the masses provided, with a power law added if ximf > 0"
    mass_min = masses.min()
    mass_max = masses.max()
    log_min = np.log10(mass_min.value_in(units.MSun))
    log_max = np.log10(mass_max.value_in(units.MSun))
    bins = 10 ** np.linspace(log_min, log_max, n_bins)
    print(bins)
    bin_number, bin_edges = np.histogram(masses.value_in(units.MSun), bins=bins)
    y = bin_number / (bin_edges[1:] - bin_edges[:-1])
    x = (bin_edges[1:] + bin_edges[:-1]) / 2.0
    for yi in y:
        yi = max(yi, 1.0e-10)

    plt.scatter(x, y, s=50, c=color, lw=0, label=label)

    if ximf < 0:
        c = (
            (mass_max.value_in(units.MSun) ** (ximf + 1))
            - (mass_min.value_in(units.MSun) ** (ximf + 1))
        ) / (ximf + 1)
        plt.plot(x, len(masses) / c * (x**ximf), c=color)
```

Listing 2.5: Routine to plot the generated Salpeter mass function. Code fragment from `amuse.examples.salpeter`.

If you need a power-law mass function with a different slope, you can change the exponent by using a keyword argument, but we recommend that you use the more general version of the mass function generator:

```
mass_zams = new_powerlaw_mass_distribution(
    1000,
    mass_min=0.1 | units.MSun,
    mass_max=100 | units.MSun,
    alpha=-2.0,
)
```

which will return a list of 1000 stars with masses between 0.1 and 100 $M_\odot$, distributed according to a power law with $\alpha = -2$.

The masses can now be easily assigned to the previously generated Plummer distribution, by

```
particles.mass = mass_zams
```

or one can generate a new particle set:

```
particles = Particles(mass=mass_zams)
```

The particles can subsequently be assigned additional attributes, as already discussed. Note that changing the masses causes the system to have a different total mass and a different ratio of potential to kinetic energy, as anticipated in the function used to generate the positions and velocities, and the system generally needs to be re-scaled (see Section 3.3.1).

2.3 Creating systems of multiple stars and planets

2.3.1 The Solar system

The initial conditions of the solar system are unknown. But if we are interested in the future dynamics of the solar system we could start with the masses, positions, and velocities of the Sun and eight planets as they are observed at some moment in time, and then integrate the equations of motion forward in time. Table 2.1 presents such conditions from which such a simulation can be started.

name	mass	position			velocity		
	[M_{Jup}]	[au]			[km/s]		
Sun	1047.517	0.005717	−0.00538	-2.130×10^{-5}	0.007893	0.01189	0.0002064
Mercury	0.000174	−0.31419	0.14376	0.035135	−30.729	−41.93	−2.659
Venus	0.002564	−0.3767	0.60159	0.03930	−29.7725	−18.849	0.795
Earth	0.003185	−0.98561	0.0762	-7.847×10^{-5}	−2.927	−29.803	−0.000533
Mars	0.000338	−1.2895	−0.9199	−0.048494	14.900	−17.721	0.2979
Jupiter	1.000000	−4.9829	2.062	−0.10990	−5.158	−11.454	−0.13558
Saturn	0.299470	−2.075	8.7812	0.3273	−9.9109	−2.236	−0.2398
Uranus	0.045737	−12.0872	−14.1917	0.184214	5.1377	−4.7387	−0.06108
Neptune	0.053962	3.1652	29.54882	0.476391	−5.443317	0.61054	−0.144172

Table 2.1: A realization of the Solar System, up to the last known significant digit, calculated for 5 April 2063 at 00:00 hours—Julian date 2474649.5 days (corresponding to first contact: http://www.startrek.com/article/origin-of-first-contact-day-explained)—using the ephemerides from http://ssd.jpl.nasa.gov/txt/p_elem_t2.txt, estimating the uncertainty in the positions and velocities using the dispersion given by Folkner (2011).

The orbital parameters from this table are coded-up in the routine `new_solar_system.py`. The parameters for moons, planetesimals, Trojans, Greeks, plutinos, the classic Kuiper belt, and the Oort cloud are not included in this table. For the moons we have a separate function that generates initial conditions for all the Solar system's moons, it is called `solar_system_moons.py`. For the asteroids and other minor bodies, you will have to download the appropriate data, and convert the ephemerides to Cartesian coordinates using your own routine (see the assignment in Section 2.5.2).

The source code to initialize the Sun, Venus, and Earth is presented in Listing 2.6. It starts by including two basic AMUSE functionalities—particles and units:

```
10 from amuse.datamodel import Particles
11 from amuse.units import units
```

A particle set is then declared in line 14:

```
14     particles = Particles(3)
```

Attribute values are then set. The first particle in the array (`particle[0]`) is assigned to the Sun and the other two particles to Venus and Earth. (Note the details of setting three vector coordinates and units in a single line.) Then, writing

```
print(sun.position)
```

will print out the Cartesian coordinates of the Sun. After initialization, the particles

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

Listing 2.6: Code to generate initial conditions for Venus and Earth orbiting the Sun. Code fragment from `amuse.examples.sun_venus_earth`.

are moved to the center of mass of the 3-body system, using the (class-defined) function `move_to_center()` mentioned in the previous subsection. At the end of the function, the particle set is returned to the main script with

```
34     return particles
```

A potentially easier way to achieve the initialization of the Sun, Venus and Earth, would be to use the following few lines:

```
from amuse.ic.solarsystem import new_solar_system
solar_system = new_solar_system()
SunVenusEarth = solar_system[0] + solar_system[2] + solar_system[3]
SunVenusEarth.move_to_center()
```

The third particle in this array actually represents the center of mass of the Earth-Moon system. To separate these two objects, one could use this (somewhat verbose and slow, but more precise) version:

```
from amuse.ic.solar_system_moons import new_lunar_system

solar_system = new_lunar_system()
Sun = solar_system[solar_system.name=="sun"]
Venus = solar_system[solar_system.name=="Venus"]
Earth = solar_system[solar_system.name=="Earth"]
SunVenusEarth = Sun + Venus + Earth
SunVenusEarth.move_to_center()
```

In this latter script one actually uses the mass, position and velocity of Earth, rather than of the Earth-Moon system.

The integration of the orbits can be carried out using the script in Listing 3.4 (see Section 3.4.1). Listing 2.7 presents a snippet of code to plot the planets' positions as they orbit the Sun. The results of the calculation are shown in Figure 2.4. We use `matplotlib.pyplot` (imported via convention as `plt`) to present our results, and later will show how to use an AMUSE-tailored version for plotting with units and hydrodynamical models. Note that here and throughout we provide references to the scripts that produced the figures and, for convenience, we save the plot in a file (`plt.savefile(...)`) in the current directory, and also display it on the screen (`plt.show()`).

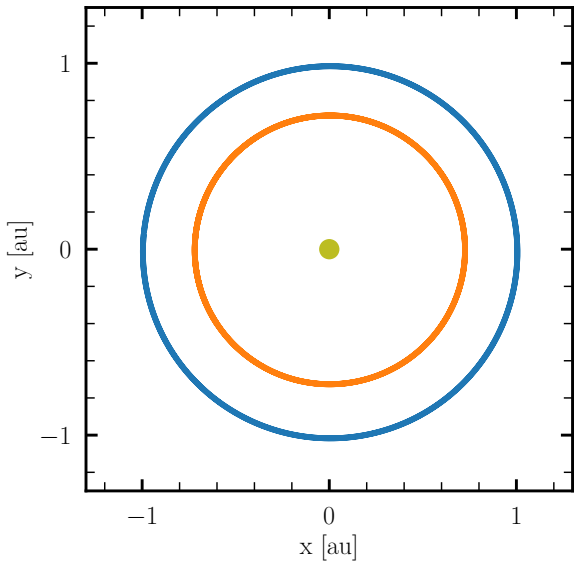

Figure 2.4: Orbital integration of the Sun, Venus and Earth for 1 yr using the script in Listing 3.4. The script to generate this figure can be found in `${AMUSE_DIR}/examples/textbook/sun_venus_earth.py`.

```
81 def plot_track(particles, output_filename):
82     figure = plt.figure(figsize=(10, 10))
83     plt.rcParams.update({"font.size": 30})
84     ax = figure.add_subplot(1, 1, 1)
85     ax.minorticks_on()
86     ax.locator_params(nbins=3)
87 
88     x_label = "x [au]"
89     y_label = "y [au]"
90     plt.xlabel(x_label)
91     plt.ylabel(y_label)
92 
93     ax.scatter([0.0], [0.0], color="#CCCC00", lw=8)
94     for planet in particles:
95         if planet.name.lower() in ["venus", "earth"]:
96             ax.plot(
97                 planet.get_timeline_of_attribute_as_vector("x")[1].value_in(units.au
    ),
98                 planet.get_timeline_of_attribute_as_vector("y")[1].value_in(units.au
    ),
99                 color=planet.color,
100             )
101     ax.set_xlim(-1.3, 1.3)
102     ax.set_ylim(-1.3, 1.3)
103 
104     plt.savefig(output_filename)
105     print(f"\nSaved figure in file {output_filename}\n")
```

Listing 2.7: Plotting the orbits of Venus and Earth around the Sun using pyplot. This tells you more about `pyplot` than about AMUSE, but we present it here for completeness. Code fragment from `amuse.examples.sun_venus_earth`.

2.3.2 Other planetary systems

In principle, any planetary system can be initialized in the above described method, by providing masses, positions and velocities of the planets. Following planet formation through Oligarchic growth Kokubo & Ida (2002)

```
from amuse.ic import make_planets_oligarch
from amuse.units import units
mass_star = 1 | units.MSun
radius_star = 1 | units.RSun
inner_radius_disk = 10 * radius_star
outer_radius_disk = 42 | units.au
mass_disk = 0.01 * mass_star
planetary_system = make_planets_oligarch.new_system(
    mass_star,
    radius_star,
    inner_radius_disk,
    outer_radius_disk,
    mass_disk,
)
print(planetary_system.planets)
```

and to print the planetary data

```
print(planetary_system.planets[0])
```

Integrating such a system can be achieved by

```
gravity = Ph4()
gravity.particles.add_particles(planetary_system)
gravity.particles.add_particles(planetary_system.planets[0])
```

The implementation of the oligarchic planet formation model was designed by Tremaine (2015) and we adopt the implementation by Hansen & Murray (2013).

```
from amuse.ic import make_planets_oligarch
mass_star = 1 | units.MSun
radius_star = 1 | units.RSun
inner_radius_disk = 10 * radius_star
outer_radius_disk = 42 | units.au
mass_disk = 0.01*mass_star
planetary_system = make_planets_oligarch.new_system(
    mass_star,
    radius_star,
    inner_radius_disk,
    outer_radius_disk,
    mass_disk,
)
```

2.3.3 Binaries and hierarchical multiples

Generating binaries and hierarchical multiple systems may be somewhat elaborate, but if this is done systematically it is not hard. Best is to start with the inner most (hardest) binary (or more than one if the system is highly hierarchical).

Let's start by making a simple binary, as we do in Listing 2.8. This could be a planet with moon, or a star with planet, or a galaxy with a dwarf companion. We assume that both objects are point masses, and that they have a Kepler orbit.

The Kepler solver in this routine is presented in Listing 2.9.

After the binary is generated, one would like to see if it really turned out as expected. A simple way to achieve this is presented in Listing 2.10, where we calculate the semi-major axis and eccentricity from the binary's Cartesian coordinates.

A higher order hierarchical multiple can be generated by iteratively using the generated binary as one of the stars in the binary generation routines.

2.4 Often returning initial conditions

For practical purposes we have generated a series of often appearing initial conditions. These are collected in a separate Python package with a number of generation functions. These functions take the form of `make_`, `add_`, `conv_` and `move_` for generating a new

```
def new_binary_orbit(stars, semimajor_axis, eccentricity=0, kep=None):
    """
    Takes a Particles object of length 2 and returns a binary particle from the
    stars, with specified orbital parameters if `kep` is `None` or using the
    values from `kep`
    """
    if len(stars) != 2:
        raise ValueError("'stars' must consist of exactly two particles")
    if kep is not None:
        rel_position = as_vector_quantity(kep.get_separation_vector())
        rel_velocity = as_vector_quantity(kep.get_velocity_vector())
    else:
        # FIXME
        orbital_period = semimajor_axis_to_orbital_period(
            semimajor_axis, stars.total_mass()
        )
        rel_position = semimajor_axis
        rel_velocity = eccentricity * 2 * np.pi * semimajor_axis / orbital_period
        # rel_velocity = ...
    mu = stars[0].mass / stars.mass.sum()
    binary = Particle()
    binary.child1 = stars[0]
    binary.child2 = stars[1]
    binary.child1.position = mu * rel_position
    binary.child2.position = -(1 - mu) * rel_position
    binary.child1.velocity = -(1 - mu) * rel_velocity
    binary.child2.velocity = mu * rel_velocity
    return binary
```

Listing 2.8: Source for generating a binary of two objects with masses *mass*1 and *mass*2, semi-major axis and eccentricity. Other orbital parameters are easily set in this routine. Source code is available in `amuse.examples.make_binary_star`.

```
def new_kepler(converter):
    """Starts and returns a new Kepler integrator with given converter"""
    kepler = Kepler(converter)
    kepler.initialize_code()
    kepler.set_longitudinal_unit_vector(1.0, 0.0, 0.0)
    kepler.set_transverse_unit_vector(0.0, 1.0, 0)
    return kepler
```

Listing 2.9: initializing a Kepler object for system in the x-y plane. Source code is available in `amuse.examples.make_binary_star`.

particle set, adding some attribute or feature, converting the particle set and moving it around. A complete list is presented in Appendix B.2.2.

These utilities can read in an existing particle set from file, and write the extended or otherwise affected particle set to file, after which it can be read in by one of the other utilities. In this way it is possible to generate rather elaborate particle sets.

For example, a Plummer distribution of 100 stars in space of which 10 receive a system of 8 planets following the Oligarchic growth model on the Sun's location in the Galaxy can be realized as follows.

```
64 def calculate_orbital_elements(bi, kepler):
65     """
66     Takes a binary particle and a kepler code, returns the semimajor axis and
67     eccentricity
68     """
69     comp1 = bi.child1
70     comp2 = bi.child2
71     mass = comp1.mass + comp2.mass
72     pos = comp2.position - comp1.position
73     vel = comp2.velocity - comp1.velocity
74     kepler.initialize_from_dyn(
75         mass,
76         pos[0],
77         pos[1],
78         pos[2],
79         vel[0],
80         vel[1],
81         vel[2],
82     )
83     a, e = kepler.get_elements()
84     return a, e
```

Listing 2.10: Code snippet for verifying the binary's orbital parameters. Source code is available in `amuse.examples.make_binary_star`.

```
python make_Plummer_model.py -N 100 -F plummer.amuse
python name_stars.py -f plummer.amuse --name PlanetarySystem \
  --nstars 10 -F plummerNamed.amuse
python add_planet_Oligarch.py -f PlummerNamed.amuse --name PlanetarySystem \
  --Nplanets 8 -F plummerWithPlanets.amuse
python move_all_particles.py -f plummerWithPlanets.amuse --pos -8300 0.0 27 \
  --vel 11.1 240 7.25 -F plummerWithPlanetsInGalaxy.amuse
```

This procedure results in a number of files `plummer.amuse`, `plummerNamed.amuse`, `plummerWithPlanets.amuse` and the final snapshot named `plummerWithPlanetsInGalaxy`. this latter snapshot can subsequently be run with a gravity code including a Galactic potential to study the evolution of the stellar system with planets.

By combining various routines it is possible to make a wide diversity of particle sets which can be used as initial conditions. Of course, not all variety of initial conditions will be available, but we expect the list of initial condition routines to grow steadily with time.

2.5 Assignments

2.5.1 Create your own initial conditions generator or manipulator

The more routines we have for generating initial conditions, the more versatile AMUSE will become. For this assignment requires some idea of how these generators and manipulators are constructed. Make a generator, constructor, manipulator for an appli-

cation that is not yet available in AMUSE. For example, one to generate a gaseous circumstellar disk around a single star. By the time you read this, such a routine probably has already been added to the repository, so find another topic for which you can make a generator. Test the code, and submit it to the git repository at https://github.com/amusecode/initial-conditions-generator.

2.5.2 Solar system minor bodies

Download the ephemerides of the Solar system minor bodies (https://www.minorplanetcenter.net/iau/mpc.html), and generate a routine to convert the Kepler elements to Cartesian coordinates.

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

1. Generate a snapshot for the Solar system at the time of first contact.

2. Check the planet positions and velocities from Table 2.1.

Bibliography

Folkner, W. M. 2011 (Oct.). Uncertainties in the JPL planetary ephemeris. *Pages 43–48 of:* Capitaine, Nicole (ed), *Journées Systèmes de Référence Spatio-temporels 2010*.

Goodwin, S. P., & Whitworth, A. P. 2004. The dynamical evolution of fractal star clusters: The survival of substructure. *A&A* , **413**(Jan.), 929–937.

Hansen, Brad M. S., & Murray, Norm. 2013. Testing in Situ Assembly with the Kepler Planet Candidate Sample. *ApJ* , **775**(1), 53.

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

Hénon, M. H. 1971. The Monte Carlo Method (Papers appear in the Proceedings of IAU Colloquium No. 10 Gravitational N-Body Problem (ed. by Myron Lecar), R. Reidel Publ. Co. , Dordrecht-Holland.). *Ap&SS* , **14**(1), 151–167.

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

Kokubo, Eiichiro, & Ida, Shigeru. 2002. Formation of Protoplanet Systems and Diversity of Planetary Systems. *The Astrophysical Journal*, **581**(1), 666.

Kroupa, Pavel. 2001. On the variation of the initial mass function. *MNRAS* , **322**(2), 231–246.

Miller, G. E., & Scalo, J. M. 1979. The initial mass function and stellar birthrate in the solar neighborhood. *ApJS* , **41**(Nov.), 513–547.

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

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

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

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

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

Part I

Fundamental principles