Chapter 1

What is Computational Astrophysics?

Informatica gaat net zo min over computers als astronomie over telescopen.
Informatics is as little about computers as astronomy is about telescopes.

Edsger W. Dijkstra

Computational astrophysics has come a long way since the early days when astronomers began to use support devices for their calculations. In this chapter we review some key developments in this field, embrace the relatively new concept of multiscale and multiphysics simulations, and introduce the Python programming environment which we will use for the rest of this book. We do this by presenting a series of examples which may look arcane now, but will (we hope) become clearer as we progress through the text.

1.1 Computational astrophysics

Computational astrophysics bridges observation and theory. The improved quality of observations and the need for deeper understanding of the underlying physics has driven demand for higher resolution and increased physical realism in numerical simulations. With exponentially growing hardware performance, we expect that the importance of large-scale simulations will only increase, but the software must also improve to keep pace with observations and theory.

1.1.1 Origin of this book

The idea for this book originated from writing the first version of the Astrophysical MUltipurpose Software Environment (AMUSE) in 2009, which at the time we called the MUltiphysics Software Environment (MUSE Portegies Zwart *et al.*, 2009). Since then, we have taught numerous courses in computational astrophysics based on our software, and supervised many students in research projects based on AMUSE. The notes from these lectures and the example scripts written by our students and ourselves formed the first version of what we called the AMUSE Primer. Later the idea gelled into a book, which you are reading right now.

Writing this book was fun. Writing the software was even more fun. Writing the example scripts was the hardest thing to do. The simplest looking codes often caused us most work. Our objective was to make these codes as short and crisp as possible. Making scripts educational without losing their main purpose was our primary challenge.

1.1.2 Hands on *is* hands on

In our experience, the best way to learn is by doing. And the best way to start doing is to take an example that resembles your objective to some degree, and build on it. We therefore include a large number of examples in a separate repository within the AMUSE code distribution, with many code snippets incorporated into this book.

In practice we prefer to work as follows:

- start with a scientific question,
- identify the minimal physics needed to address the question,
- find an example script that resembles the target problem as closely as possible,
- rewrite that script to make it solve our problem.

Making figures and animations can be time consuming but also very rewarding. Making a publishable figure can take as much as a day. In our examples we adopt `matplotlib`[1] (Hunter, 2007) as our graphics library. There are many alternatives to `matplotlib` for making animations or figures, but its principal advantage is its transparent interface with Python, making it easy to incorporate into AMUSE.

A simple script can be written in a few hours; a more complicated script may take days to get right. But then the long painstaking process from test script to production code starts. This generally entails a number of stages, as illustrated in Figure 1.1. The main steps include adjusting the script to one's needs, then refactoring the script, followed by profiling and optimization. We include the refactor step to stress the importance of preserving readability of source code, which so often leads to reproducible results (see Beck & Andres, 2004, for more about refactoring).

1.1.3 What about the math?

As Albert Einstein wrote on January 7, 1943 to nine-year-old Barbara Lee Wilson (Calaprice, 2002):

> Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

[1]Matplotlib is well documented and publicly available as open source from http://matplotlib.org /api/pyplot_api.html

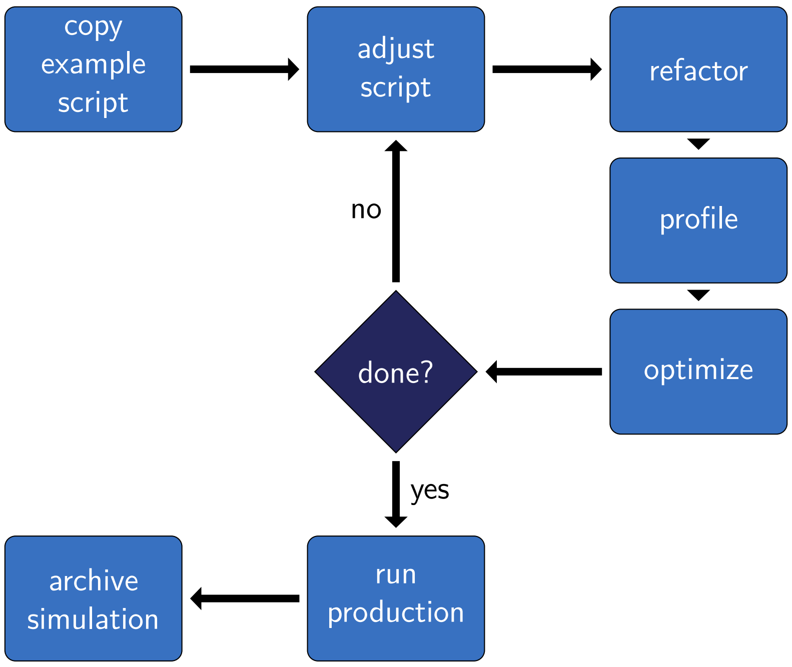

Figure 1.1: Workflow for designing a production script.

A solid background in math is nice, and surely helpful, but you can probably read and understand most of this book (up to part II) with only rudimentary mathematics and calculus. Similarly, a Ph.D. or other strong background in astrophysics or computer science will certainly make some of the material easier to digest, but we have successfully used AMUSE and this text with students at all levels, from freshman undergraduate to Masters to Ph.D. We even have users with a high-school background; In 2022, Dave Kroos graduate from Walburg College Zwijndrecht by performing calculations for his research project on *The Newtonian and Relativistic Chaotic Behaviour of the S-Star Cluster*; he received the highest possible grade.

1.1.4 Objective of this book

This book is about how to perform simulations of astrophysical phenomena. These simulations have the goal of mimicking nature, or at least certain aspects of it, in order to better understand the universe. By the end of this book you will be able to perform many high-end simulations yourself. You will have learned how to set up initial conditions, how to design and develop a simulation environment to address your astrophysical question, how to perform the runs, and how to analyze the resulting data.

You can read this book in several ways. You can simply start at the beginning and work through to the end, or you may prefer to pick out the pieces you need in order to bootstrap your own research project. Either way, we recommend that you read the first couple of chapters, to understand how AMUSE works and some of its basic programming ideas. In Appendix E, we describe how to teach AMUSE as part of a MSc educational track.

1.1.5 What is missing from this book

This book is far from exhaustive. Many topics, such as data mining, high-performance and distributed computing, programming paradigms for graphics processing units, operating systems, and the effects of limited numerical precision are entirely separate disciplines that we can only direct you to. AMUSE contains far more functionality than we can describe. We will only briefly address the algorithms underlying many of the phenomena discussed. Our discussion of the underlying physics is also necessarily limited. This is not an astrophysics text—rather, our goal is to present just enough astrophysics to allow you to harness the modules we describe.

Of the many astrophysical phenomena we could include, we have decided to limit our scope to gravitational dynamics, stellar evolution, hydrodynamics, radiative transport, and chemistry. We neglect a lot of physics, including asteroseismology, astrobiology, planetary geology, cosmology, YORP[2], and tidal dissipative effects. These are not unimportant, but the book is finite, and new modules addressing such areas are constantly being added to the AMUSE framework.

1.1.6 Outline of the book

This book is divided into three parts. In Part I, Chapters 3 to 6, we discuss scripts designed to address single physical processes, for which only one numerical solver is required. A *minimal solver* is the simplest possible script needed to run a particular solver. In AMUSE, the low-level solvers—usually written in high-performance languages and often optimized for parallel architectures—are called *community codes.* When we include the AMUSE Python interface structure we generally use the more generic term *community module.* In practice, we often need additional modules for generating initial conditions, boundary values, or post-production analysis, but the main work in a minimal code is performed by a single community module. Minimal solvers run a particular code and stop after producing just enough information to gauge the quality of the solution. The solvers presented here should be viewed as examples to illustrate the AMUSE programming style and module interfaces, but they can also be used as a basis for further work.

The next level of complexity is a *simple solver*, which is an extended version of a minimal solver. Simple solvers are meant to illustrate more complex and practical mono-physics applications. Within the AMUSE framework we can run a single code as a standalone program. Since AMUSE contains multiple solvers for each physical domain it addresses, this allows for detailed and instructive comparison between them. The ease with which initial conditions can be generated, and with which the user can change solvers and analyze the results makes running multiple solvers on the same problem a useful exercise. In this regard, the main advantage of using AMUSE is the unified way in which the input is parsed and delivered to, and results returned from, the solver. This enables the possibility of using exactly the same initial conditions for each of a set of numerical solvers, and analyzing the resulting data in a uniform way.

[2][**YORP should probably be explained – Steven**]

In Part II, Chapters 7 and 8 we discuss elementary and hierarchical code coupling strategies, followed by Chapter 9, in which we present a few example cases for multi-scale and multi-physics simulations.

In the last part of the book, Part III, we dwell on expert knowledge and the discuss the subtleties of multi-scale simulation (Chapter 10) and multi-physics simulations (Chapter 11). In the last chapter (Chapter 12), we discuss possible expansion and other packages that were derived from AMUSE.

The chapters are followed by several appendices. The installation of AMUSE is discussed in Appendix A, followed by Appendix B on AMUSE fundamentals and internal details. Important, but often not considered multi-scale or multi-physics are the generation of initial conditions, which we discuss in Chapter 2. How to add your own code to AMUSE is discussed in Appendix D. The last two appendices include a brief Python primer (Appendix F) and how to organize a Masters level course of 6 European Credits (EC) (Appendix E).

We have adopted this presentation order for its steady increase in physical complexity and its systematic introduction of more and more sophisticated AMUSE capabilities.

1.2 A brief history of simulations in astrophysics

The first recordings of theoretical astronomy date back to the Egyptian 18th dynastic (1550–1292 BCE) calculations by pharaoh Hatshepsut's architect Senmut; portions of the murals found in his tomb at Deir el-Bahri can be found at http://www.metmuseum.org (Dorman, 2006). Figure 1.2 presents part of this mural. Such calculations enabled astronomers to recognize structure and describe patterns in the heavens, an aspect of astronomy that is still relevant today.

Although the Egyptian civilization did not show much academic interest in the heavens, this changed dramatically when the Babylonian civilization emerged. Around 350 to 50 BCE, they already showed a keen interest in mathematics and algorithms. This is demonstrated by in several clay tablets with cuneiform writing, such as the one presented in Figure 1.3, which shows a calculation of Jupiter's orbital integration on the sky. The complexity of this calculation, due to the epycycles the planet makes when its orbit is folded in with the Earth's and subsequently projected on the sky, requires a second order integration. The method depicted in Figure 1.3 resembles a Leap-frog or Verlet integrator. Although we do not know who wrote the tablet, or who designed the algorithm, it is clear that the Babylonians were already aware of 2$^{\text{nd}}$ order error-reducing algorithms (Ossendrijver, 2016). It would be better to call this the Babylonian integrator.

Some time between year 350 and 370 the Egyptian astronomer, Hypatia was born in Alexandria current Egypt (Pellò, 2022). She constructed Astrolabes, hydrometers and was a keen philosopher, until she was murdered in March 415, probably by religious extremists (Belenkiy, 2010).

Many centuries later, modern astronomy started with the discovery of the first refracting telescope, built by Hans Lippershey in 1608, which had a collecting area of about 1 cm^2. Galileo Galilei reported the first astronomical applications in 1609, using

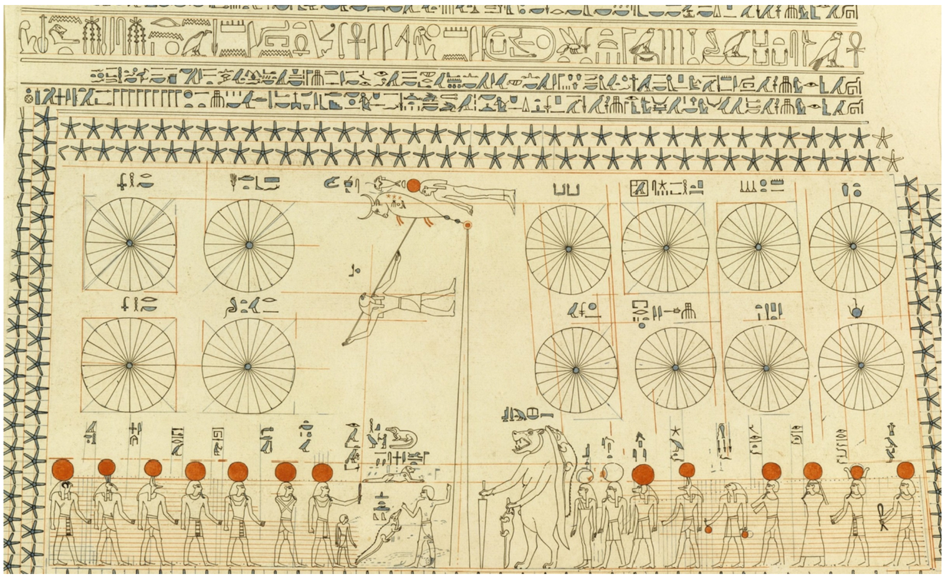

Figure 1.2: Reproduction of the lower part of the 2nd tomb of Senenmut's stela, dated around 1479-1458 BCE (accession number MMT:48.105.52 (TT353), Figure 27 of Wilkinson, 1979, GNU General Public License, Rogers Fund, 1948). This unique mural shows, apart from all the Egyptian gods, the most important constellations and a much debated conjunction between the Sun and the Pleiades star cluster. The original photograph was presented in Dorman (2006, 1991).

a telescope of similar design. In the 400 years since that momentous first application of new technology, the collecting areas of the largest research telescopes have increased by a factor of about 10^6, becoming the 10-m class telescopes in widespread use today.

Digital computers were introduced in the late 1930s (Zuse, 1967, 1993), and the first commercial programmable model, ENIAC, achieved in 1945 a computational speed equivalent to about 200 modern floating point (double precision) operations per second (flops). (We say "equivalent" because ENIAC was essentially a single precision machine.) Today the fastest supercomputers achieve speeds exceeding 10^{17} flops, a speedup of a factor of roughly 10^{15} in raw performance in just eight decades. Thus, while observing technology has grown impressively but relatively slowly over the past four centuries, computer scientists have witnessed a technological explosion in living memory.

The ongoing revolution in the availability of digital computers has led to an entirely new branch of research in which key facilities are not confined to high mountain tops in Hawaii or the Chilean Andes, but more often in the room next door. Astronomers quickly realized that they could use computers to archive, process, analyze, and mine the copious amounts of data taken by observing campaigns. However, arguably the biggest impact on the way astronomers pursue their scientific questions has been in numerical simulations of astrophysical phenomena.

Astronomical research lends itself naturally to computation because it is characterized by large ranges in temporal and spatial scales, complex and coupled nonlinear processes, and the extreme conditions found in interstellar space. With the appearance of digital computers it became possible to study processes in the intergalactic vacuum, hot plasmas at stellar surfaces, billion-year time scale processes, and black-hole physics,

Figure 1.3: Photo of text A (lines 1 to 7) of the Babylonian clay tablet from 350 to 50 BCE with the mathematical tables to predict Jupiter's orbit on the sky. The original photograph was presented as figure 1 in Ossendrijver (2016). The tables is about 4×4 cm.

none of which can be studied in Earth-based laboratories. The major current limitation to studying the universe by means of computation lies in software, and in particular our limited understanding of algorithms that can solve multiscale, multiphysics problems. AMUSE is an attempt to address these issues.

1.2.1 The first simulation experiments

One of the first astrophysical simulations was carried out in 1941 by Erik Holmberg, a graduate student at Lund University who had access to a number of photocells. Using the fact that the intensity of radiation drops off as the inverse square of the distance, like the force of gravity, he devised an experiment to simulate two colliding galaxies. The initial setup for a single "galaxy" in Holmberg's experiment is presented in Figure 1.4. The progress of his experiment is shown in Figure 1.5. The arrows indicate the directions in which the galaxies move with respect to one another, and how they rotate. This experiment made Holmberg one of the founders of what is now an immense body of astronomical research in gravitational dynamics, some of it reviewed by Binney & Tremaine (1987).

Holmberg had no access to a computer, so all of his calculations were performed by hand. His article (Holmberg, 1941) is interesting in many respects, but unclear on exactly how the forces (photon counts in his case) were used to update the positions of

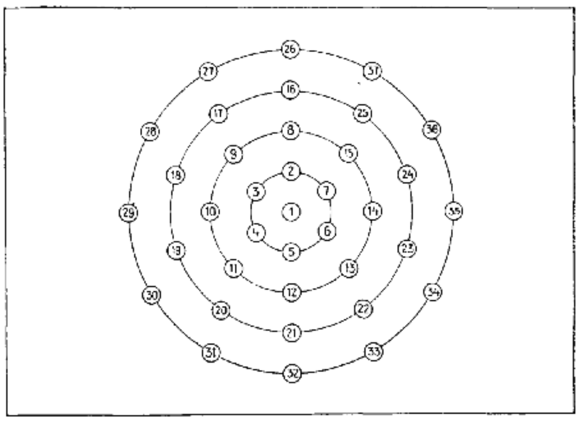

Figure 1.4: Copy of Figure 3 from Holmberg (1941), presenting the initial conditions for a single galaxy in Holmberg's simulation of two colliding galaxies. (Copyright AAS. Reproduced with permission.)

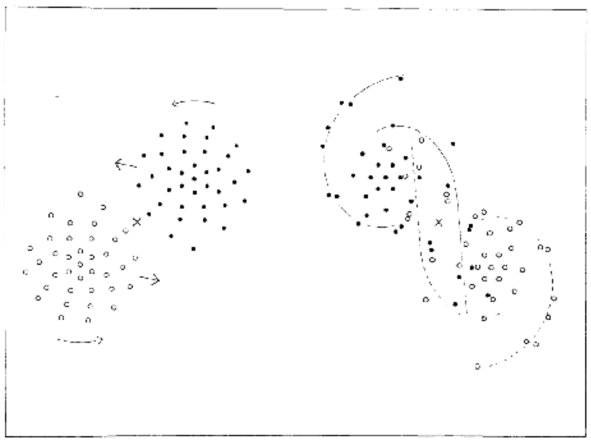

Figure 1.5: Copy of Figure 4a from Holmberg (1941), presenting two snapshots of the results from his calculations. Both snapshots ware taken at the moment the two galaxies were separated by twice their radii, one before the interaction (left) and one after the first close approach (right). (Copyright AAS. Reproduced with permission.)

the particles (the light bulbs). Today, in a comparable paper, we might state that the integration was performed using (say) a 4th-order Hermite predictor-corrector scheme with direct force evaluation and zero softening (see Makino & Aarseth, 1992). The choice of algorithm (in this case the Hermite scheme) is an important part of the solution. Although this level of detail might sound excessive, you will better appreciate such considerations by the time you have read Chapter 3.

1.3 Software used in this book

1.3.1 Motivation for a homogeneous software environment

Developing high-quality simulation software is relatively straightforward when the physical scope of the problem is limited. The task becomes considerably more difficult once the application spans multiple physical domains, or if the dynamic range in temporal or spatial scales exceeds the precision of the hardware. Successful examples of complex multiphysics software in computational astrophysics are `NBODY7` (Aarseth, 1999, 2011), `FLASH` (Fryxell *et al.*, 2000, 2010), and `GADGET` (Springel *et al.*, 2001; Springel, 2000), which have broad user bases and can be applied to a wide variety of problems. (Note that, where possible, we provide two references for each software implementation, one to the publication in which the package was first described, and one to the Astrophysics Source Code Library at http://ascl.net/ or ZENODO https://zenodo.org/) Some of these codes address relatively narrow ranges of physics and are limited by the specific numerical algorithms adopted to solve the underlying equations, but others, like `FLASH`, are highly modular in their internal organization, in some ways similar to AMUSE.

Figure 1.6 presents typical temporal and spatial scales for a number of astronomical phenomena. The ellipses indicate specific areas of interest. The colors indicate physical domains: gravity (blue), stellar evolution (red), hydrodynamics (yellow), and radiative processes (green). Several domains are identified by name, such as the collapse of a giant molecular cloud (top right in yellow), and the range of habitable planet orbits (blue circle at lower left). Gravity and hydrodynamics span a broad range of temporal and spatial scales, whereas radiative processes and stellar evolution exhibit much narrower ranges. Note that both spatial and temporal scales span a range of about 14 orders of magnitude, from galactic orbits at one extreme to stellar mergers or intricate orbital dynamics at the other.

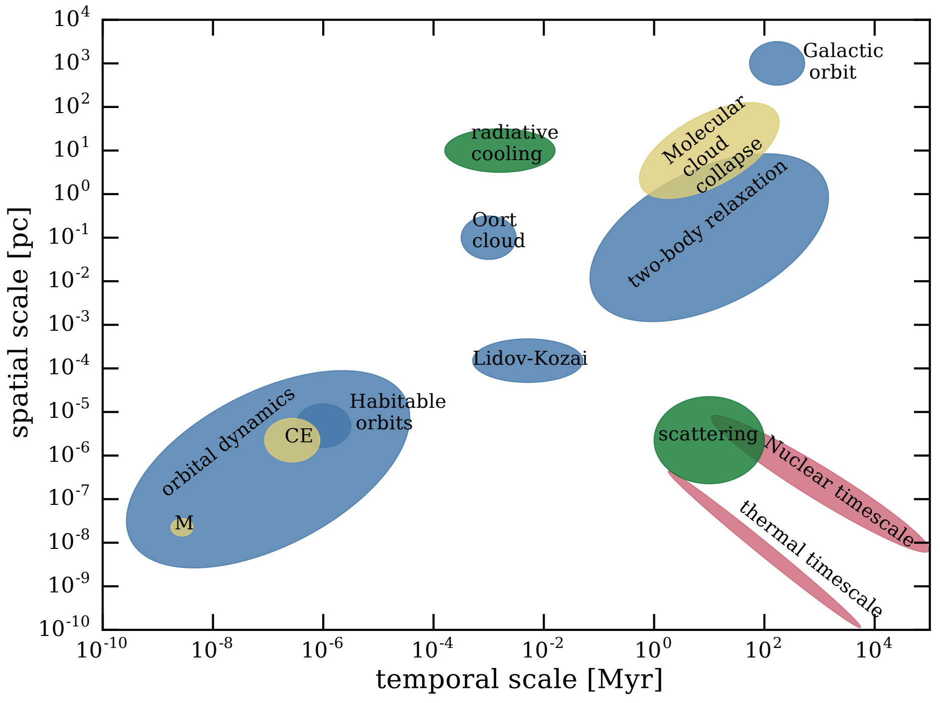

Figure 1.6: A number of typical scales, for gravitational dynamics (blue), stellar evolution (red), hydrodynamics (yellow), and radiative transfer (green), are indicated at several locations in the plane of temporal and spatial scales. Here several small scales (mergers indicated by M) and common envelope evolution (indicated by CE) are embedded within the orbital dynamics scale ellipse.

Many existing simulation codes provide excellent solvers within their specific physical domains, but astrophysical problems often require solvers for a wide variety of domains. For example, understanding the ejection of primordial gas from an embedded star cluster, born during a galaxy merger, requires simultaneous treatments of collisional stellar dynamics (the protostellar cluster), collisionless stellar dynamics (the galaxy merger), hydrodynamics (interstellar gas), stellar evolution (embedded protostars and stars), and radiative processes (feedback between stars and the interstellar gas). All these processes occur on comparable temporal and spatial scales.

Instead of designing new algorithms to solve all of these problems simultaneously, we adopt the philosophy that each of the physical domains can be solved separately, then combined into a consistent solution at discrete intervals in time and space. Validation of such a simulation could subsequently be accomplished by recomputing the model with a different solver for each of the physical domains. In this way we can study the convergence (or the absence thereof) of the solution with a sequence of numerical implementations. The Astrophysical Multipurpose Software Environment (AMUSE; Portegies Zwart, 2011; Portegies Zwart *et al.*, 2013) allows a user to combine existing solvers to build new applications that can be combined again to study increasingly complex problems. This enables the growth of multiphysics and multiscale application software in a hierarchical fashion, and allows testing of each intermediate step as the complexity of the software increases.

AMUSE has a hierarchical component architecture that encapsulates dynamically shared libraries for simulating stellar evolution, gravitational dynamics, (magneto)-hydrodynamics, radiative processes, and also has limited support for astrochemistry. Applications are able to use some or all of these resources simultaneously, and can exchange data in a unique and deterministic way. With this hybrid simulation environment it becomes possible to study multiphysics processes operating on a broad range of length and time scales.

1.3.2 Choice of programming languages

We have adopted Python (van Rossum, 1995a,b,c,d) as the implementation language for the AMUSE framework and high-level management functions, including bindings to the communication interface. The earlier version of this book used Python2.7, but we have since migrated to Python3 Van Rossum & Drake (2009) (as of writing 3.7 or higher). The choice of Python is motivated by its flexibility, broad acceptance in the scientific community, object oriented design, and ability to allow rapid prototyping, which shortens the software development cycle and enables easy access to the community code in the community module, albeit at the cost of slightly reduced performance. However, the entire AMUSE framework is organized in such a way that relatively little computer time is actually spent in the framework itself—most time is (or should be) spent in the community code(s). The overhead from Python compared to a compiled high-level language is typically $\lesssim 10\%$, and often much less (Pelupessy *et al.*, 2013).

As discussed further in Paragraph D.1.1.2, our implementation of the communication layer in AMUSE uses the standard Message Passing Interface protocol (MPI; L. *et al.*, 1996) and SmartSockets (Maassen & Bal, 2007). Normally MPI and SmartSockets are used for communication between compute nodes on parallel distributed-memory systems or within computational grids. However, in AMUSE they are used as communication channels between *all* processes, whether or not they reside on the same node as the control script, and whether or not they are themselves parallel.

AMUSE community codes are written in a variety of languages, reflecting the programming backgrounds and preferences of their authors. As a result, for each community code there are two interface functions. The one on the AMUSE side is written in Python and is called the proxy. The interface on the community code side is called the partner and is written in the native language of that code. These two interfaces communicate via one of the message-passing protocols just described. Thus our only real restriction on supported languages is the requirement that the community code is written in a language with MPI or Smart Sockets bindings, such as C (Ritchie, 1993b,a), C++, (Stroustrup, 2013), Java (Gosling *et al.*, 1996, 2014), and members of the FORTRAN family (Backus, 1978).

We emphasize that, from the user perspective, the details of the interface and communication protocols are invisible. These details are central to the AMUSE design, but most high-level users are blissfully unaware of them. To the user, a Python AMUSE script looks just like a traditional serial program.

1.3.3 Design of AMUSE

The AMUSE project goal stipulated three basic functionalities:

(1) a homogeneous, physically motivated interface for existing astronomical simulation codes;

(2) the incorporation of multiple community codes from four fundamental domains (stellar evolution, gravitational dynamics, hydrodynamics, and radiative transfer); and

(3) the ability to design new simulation experiments by combining one or more of the community codes in various ways.

In this way researchers can concentrate on physically relevant details, such as the calling sequence for the various physical domains, without having to worry about the many technical details.

Each of the existing community codes in AMUSE is embedded in an interface layer that realizes communication between the AMUSE framework and a community code. One important advantage of this strict separation is that it explicitly prohibits short-cut communication between independent subroutines, greatly diminishing the opportunities to create "spaghetti code". In addition, it allows us to manipulate data as it moves between the community code and the framework. One such data manipulation is the conversion of units, as discussed in Section 2.3.1. The handling of community-code dependent parameters are largely managed by the framework, as is the acquisition and mining of simulation data. Figure 1.7 illustrates schematically the philosophy of AMUSE, with the central AMUSE framework surrounded by its four fundamental physical solvers.

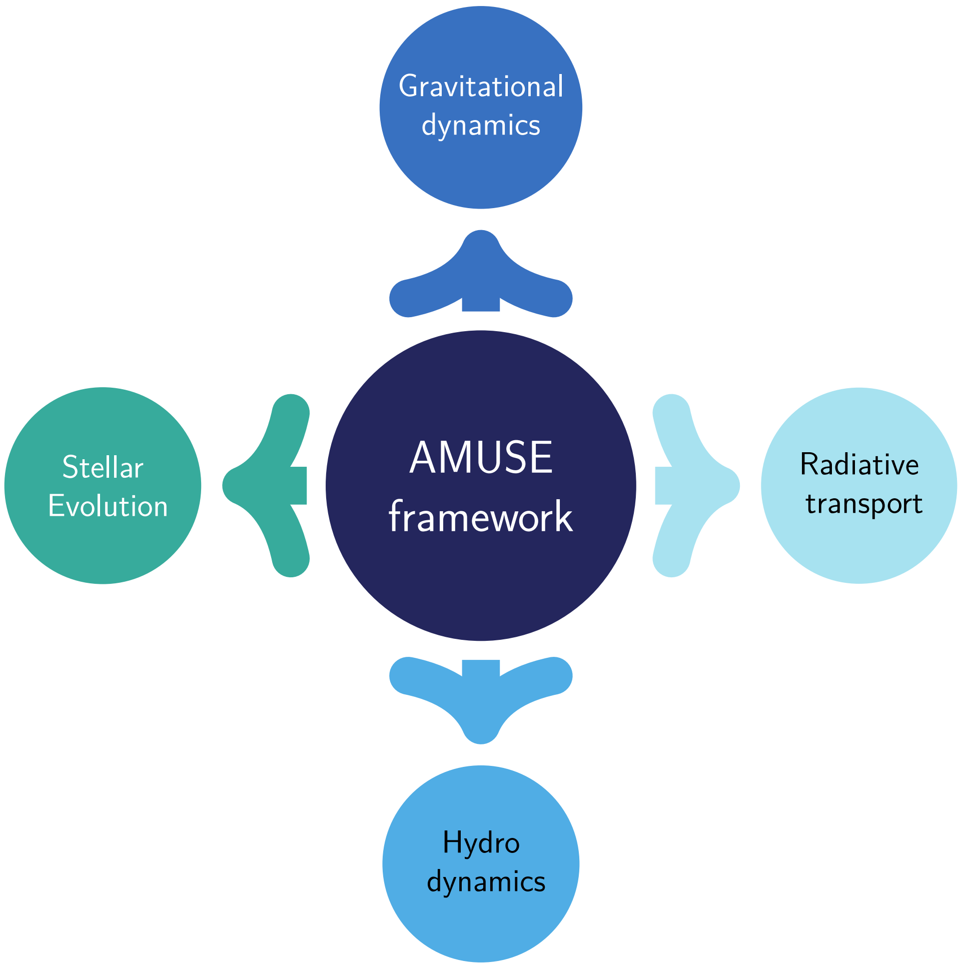

Figure 1.7: Multiscale and multiphysics codes are connected disciplines, as depicted here in the domain–application plane.

New hybrid codes can be written in AMUSE by combining community codes using Python. An AMUSE Python script for solving a particular problem prescribes the order in which events are handled and specifies the numerical solver that should be used to resolve each event. Once initial conditions are generated and the simulation is completed, the resulting data can be stored or further analyzed using the tools available in AMUSE, or developed by the user. The technical problems associated with coupling codes, setting boundary conditions, specifying run-time parameters, and guaranteeing that the various elements communicate in the proper units are handled entirely by AMUSE. A thorough understanding of the limitations of each of the incorporated codes is important for assessing the quality of the simulation, but the burden of managing the technical details is not forced on the user.

We have demonstrated modularity and interchangeability of the AMUSE community codes for solving the basic physical processes described above. The user can select from a variety of gravitational N-body solvers (symplectic N-body codes, direct N-body codes, and tree codes) by changing a single word in the script. The same holds for stellar evolution codes (parameterized and Henyey codes), hydrodynamics solvers (smooth particle hydrodynamics and adaptive mesh refinement codes), and radiative transfer codes (Monte Carlo photon transport codes, and tessellated photon flow solvers). The modularity, and how a complex cascade of numerical solvers can grow to an end product, is illustrated in Chapter 9, and in Part II. This cascade of jargon may sound like Jabberwocky (Carol, 1871) right now, but we hope that much of it will eventually make sense by the time you reach the end of this book.

1.3.4 Terminology

The basic structure of the AMUSE framework is illustrated in Figure 1.8. The figure is subdivided into three blocks. The top block is mostly user written, the middle block implements the interfaces onto the community codes, and the bottom block is the community modules and the actual community source code, which is connected to the rest of the framework via MPI or sockets.

The components can be described as follows: [**though it is a bit confusing to do so, maybe rewrite – Steven**]

- The Python *user script* (top layer in Figure 1.8), which addresses a specific physical problem or set of problems, in the form of system-provided or user-written Python commands that serve as the user interface onto the framework. This is the only part of a production script that needs to be written by the user. The coupling between community codes is implemented in this layer, with the help of support classes for converting units and data structures provided by AMUSE.

- The *community module*, comprising three key elements:

  – The *manager* (second layer in top block), which provides an object-oriented interface onto the communication layer via a suite of system-provided utility functions. This layer handles unit conversion, and also contains the state engine and the associated data repository, all of which are required to guarantee consistency of data across modules. This layer is generic—it is not specific to any particular problem or physical domain.

  – The *communication layer* (second block), which realizes the bidirectional communication between the manager and the community code layer. This is implemented via a proxy and an associated partner, which together provide the connection to each community code.

  – The *community code layer* (third block), which contains the actual community codes and implements control and data management operations on them. Each piece of code in this layer is domain-specific, although the code may be designed to be very general within its particular physical domain.

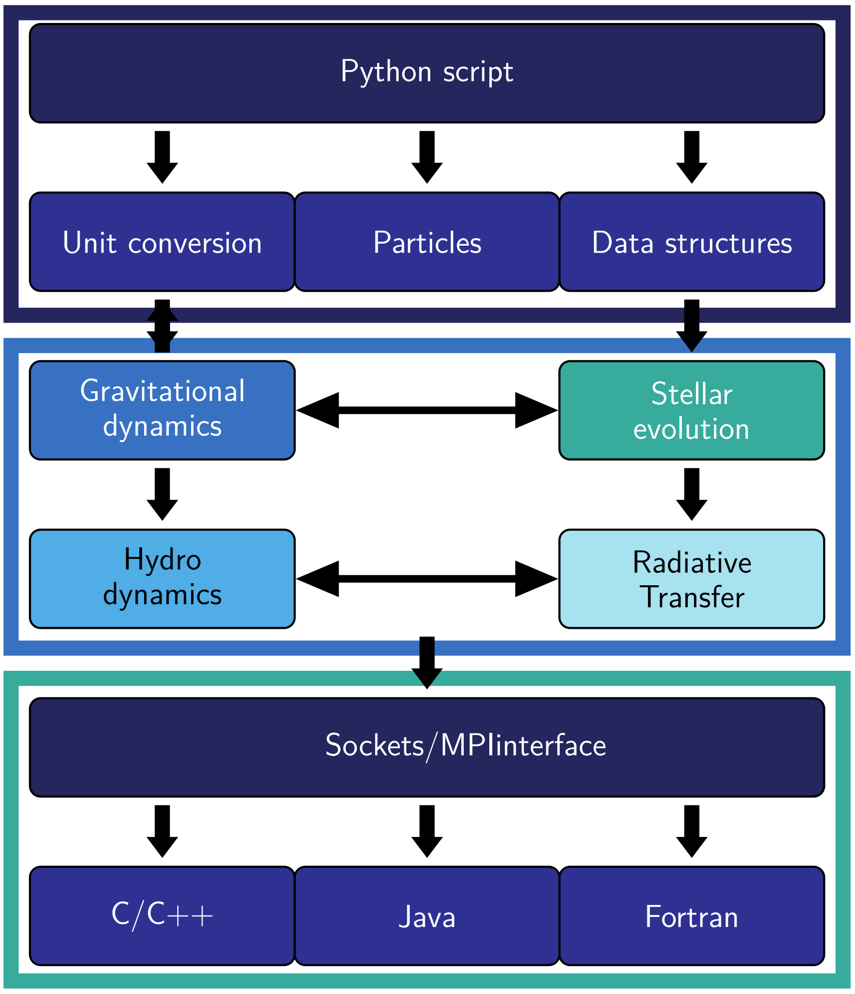

Figure 1.8: General structure of the AMUSE framework. The user writes the Python script (top layer). The lower layers in the first block are responsible for units and data structures. The framework in the second block represents the interfaces to the community modules (blue for gravitational dynamics, green for stellar evolution, purple for hydrodynamics and radiative transfer in light blue to conform with the color coding of Figure 1.6), which connect via sockets or MPI (third block) to the community codes at bottom. Information flows downward to the community modules, as indicated by the arrows, as well as upward, back to the Python script.

1.3.5 Installing AMUSE

Before you continue reading, we advise you to install the AMUSE package. Although, we made great progress on improving the installation, it still remains a bit of a hit-and-miss endeavor. The package has many dependencies, and some of them turn out to be rather fragile. We can do as much as explain how some installations work and why some don't, but there are too many compilers, additional packages, operating systems, computers and wide variety of hardware architectures to be exhaustive on the topic. In Appendix E.12, in particular Appendix A.1, we give a more thorough overview of the AMUSE installation, in case the following simple installation fails.

Here is a brief overview of the installation steps. We assume that you are working in a Linux or macOS environment. We don't recommend or support AMUSE under

Windows, except within a Linux environment such as Windows subsystem for Linux[3] or Multipass[4]. For the following we assume that you have a basic understanding of your operating system and its makefile structure, and that the basic packages are installed. If this is not the case, we refer to Appendix E.12, for further details.

We recommend also to install from the GitHub source. In that case, you always have the latest version.

We also recommend that you work from some sort of virtual environment, either conda[5] or Python3's `venv`[6]. Information on these packages is available online, but other virtual environments could work equally well.

1.3.5.1 Installing AMUSE in a python virtual environment

Here we will start from a Python venv on a Ubuntu linux laptop. This tutorial was written on a fresh Ubuntu 22.04 LTS (6.2.0-32-generic) installation on a Lenovo X1 laptop. Not a super-duper computer, but it should suffice for all example calculations in this book. If only, because all the example calculations in this book are generated with this same machine. We would like to emphasize here that the revenues we receive for writing this book is by far insufficient to purchase such a laptop.

Open a terminal and copy the lines below to create an environment for your project (note that the leading only indicates the terminal prompt, and it may be different on your computer. The instructions are written for `bash` or another Bourne shell.

```
%> mkdir {MY_AMUSE_PROJECT} && cd {MY_AMUSE_PROJECT}
%> python3 -m venv env
%> source env/bin/activate
```

The last line activates the virtual environment, it must be repeated every time you open a new terminal. It might be good to make an alias for this, such as

```
%> alias amuse='source {MY_AMUSE_PROJECT}/env/bin/activate'
```

Every time you stop using AMUSE, you can type

```
%> deactivate
```

to leave the virtual environment.

You may now want to start by making sure that your have the latest version of `pip` installed.

```
%> python3 -m pip install --upgrade pip
%> python3 -m pip --version
```

Now you have updated `pip` and the resulting output should indicate it latest version. Maybe you see something like

[3]https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux
[4]https://multipass.run
[5]https://docs.conda.io/projects/conda/en/latest/index.html
[6]https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/

```
pip 22.1.2 from /home/spz/.local/lib/python3.10/site-packages/pip (python 3.10)
```

Indicating that I will be using `pip` version 22.1.2 with python 3.10. You will probably find a later version for both packages.

Now initiate the virtual environment by

```
%> python3 -m venv amuse
```

and activate it

```
%> source env/bin/activate
```

You have created the amuse environment in your root directory, and you have it activated. Now install the `pip` requests

```
%> python3 -m pip install requests
```

In principle you are now done installing the virtual environment. Every time you stop using AMUSE you can type

```
%> deactivate
```

And you can restart by typing

```
%> source env/bin/activate
```

Now, before we install the prerequisites for amuse, you may want to install a few extra packages. If you are running on a supercomputer, you may want to use `modules` to install these. The most important modules include

```
%> module load CUDA
%> module load MPFR
```

Your system administrator will have set default choices for these, but of course you are free to select a specific version if necessary. The advantage of using modules instead of `pip` is that they will be tuned for you architecture.

AMUSE can be installed in two ways: via PyPI packages and using the developer installation. We recommend using the former method here, though we will briefly address the developer installation too.

After installing the prerequisite libraries, you can install the AMUSE framework:

```
%> pip install amuse-framework
```

This will install both the core of AMUSE and other required Python modules. You may also want to install these additional packages that will be useful later on, though they are not essential for the fundamental AMUSE installation:

```
%> pip install astropy jupyter pandas seaborn matplotlib
```

Now, you can install the individual AMUSE community codes:

```
%> pip install amuse-{COMMUNITY_CODE_NAME}
```

You don't need to install all of the codes, but for the examples in this book, we recommend the following base set:

```
%> pip install amuse-hermite amuse-bhtree amuse-sse amuse-seba amuse-fi \ldots
```

Also, you might want to install the examples from this book:

```
%> pip install amuse-examples
```

1.3.5.2 Installing AMUSE in a conda virtual environment

Here we will start from a conda virtual environment on a Ubuntu linux laptop. This tutorial was written on a fresh Ubuntu 22.04 LTS (6.2.0-37-generic) installation on a Lenovo p14s laptop. Again, not a super-duper computer, but it suffices for all example calculations in this book. WE recommend this way of installing AMUSE for developers, but it is also useful for any regular (or non-regular) user.

First you create a virtual environment using conda, and activate it.

```
%> conda create -n amuse_env
%> conda activate amuse_env
```

For convenience we install the AMUSE source and executables in a separate directory (in this case a directory called `amuse` in your home directory

```
mkdir ~/amuse
cd ~/amuse
```

In the nest step we download the package through git.

```
git clone https://github.com/amusecode/amuse.git ./
```

AMUSE requires a considerable number of helper packages, which are listed in the local `amuse` directory in the file `requirements.txt`. A convenient way to install them in your conda environment is through `pip`, which can be done as follows:

```
  %> pip install -r requirements.txt
```

Now we have to build AMUSE but before doing so we And allow pip to build it from source. The "-e" option here indicates that your install a project in the develop mode, allowing you to edit the source.

```
  %> pip install -e .
```

In the last step, we configure the make-files, and build the code

```
  %> ./configure
  %> make
```

This last step may take a while (about 20 minutes), even on a modern core-i7 processor with SSDs.

After this procedure your fundamental AMUSE framework has been build, and probably most of the packages and libraries. End the end of the installation the operating system prompts you with a line indicating what has been build. On my computer this looks as follows

```
   52 out of 62 codes built, 7 out of 7 libraries built
```

indicating that 52 out of 62 packages have been build, and all libraries. It means that not all codes were successfully build, but all the libraries.

The output will be a list of all the codes successfully build, and a list of packages that failed. It could look as follows:

```
Community codes not built (because of errors/ missing libraries):
================================================================================
 * bonsai2
 * distributed
 * etics
 * flash
 * higpus
 * mocassin
 * pikachu
 * sakura
 * sei
 * simplex
Optional builds skipped, need special libraries:
 * fi - periodic_gl, gl
 * hacs64 - gpu
 * huayno - cl
 * mi6 - gpu
 * phigrape - gl_gpu, gl, mpi, grape
================================================================================
```

Indicating the failed packages, and those that require to be downloaded separately (such as `FLASH`), require specific hardware (such as `distributed`) or dedicated compilers (such as `Fi`), or maybe some of these codes simply contain a bug introduced in some of our last edits or an update from one of the authors.

This is, in principle nothing to be worried about. Some libraries are written for very specific hardware, such as GPUs (and you may have an unsupported version) and some codes require special external libraries that you have not installed on your system. With your current installation (of at least some dozen codes are build) you can probably run most of the scripts in this book.

You can see what went wrong (or good) in the file `build.log`, which in my case

```
   %> less build.log
```

In this example case discussed here the radiative transfer code `mocassin` (version 2.02.70) was not compiled. It turns out, reading the file `build.log`, that in line 152 and in line 872 the call iteration_mod.f90. The latter, on line 872

```
    call mpi_allreduce(grid(iG)%lgBlack, lgBlackTemp, size, &
         & mpi_integer, mpi_sum, mpi_comm_world, ierr)
```

fails to accept the second argument which is a REAL rather than an INTEGER due to an earlier division of an INTEGER by a REAL, which in the new F90 compiler casts the argument to a REAL, whereas an older compiler would have cast it to an INTEGER. This is not a hard bug to solve, but it could be tricky to just cast the argument. It probably requires some interventions with the original authors to resolve this issue. By the time you read this, this bug should be resolved, but because maintaining community code packages in the latest compilers is probably not high on the `mocassin` developers (they have more important things to do, such as run the code on older compilers in order to generate research), this bug may actually still be in the source. Luckily, AMUSE contains other radiative transfer packages that did build properly, and we can still do our research, while waiting for `mocassin` to be repaired.

As a final note, out of convenience, one can build separate codes instead of building all codes simultaneously. This is achieved by the following command:

```
%> make {mycode}.code
```

Here one has to replace {mycode} by code desired. For example 'hermite'.

Once executed, the build system returns with the following

```
Community codes built
================================================================================
* hermite
================================================================================
1 out of 1 codes built, 0 out of 0 libraries built
(not all codes and libraries need to be built)
```

indicating that one code has been build (hermite), and no libraries (as there are none to be build).

1.3.5.3 testing your AMUSE installation

After a successful installation, we encourage you to run a standard set of tests in the directory `${AMUSE_DIR}/test`[**FIXME: use pytest directly – Steven**]. In the `test` directory you may want to go to one of the sub-directories, such as `pytest codes_tests`, `core_tests`, `ext_tests` or `ticket_tests`, and run

```
%> pytest *
```

This may result in a few warnings, mainly due to round-off, but hopefully no errors.

Run some of the examples from the book. Just say

```
%> python amuse_script.py
```

Here we assume that you have activated the virtual environment in which you installed AMUSE.

Your AMUSE installation comes with all of the example, data, and figure files used in this book. They can be found in the `amuse.examples` module. You may want to run your first AMUSE example script by typing on the command line

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

This should produce a figure that looks similar to Figure 2.4 below.

To maintain AMUSE, you may want to download the latest version once in a while and rebuild the package.

```
   %> cd ~/amuse
   %> git update
   %> make
```

1.3.6 Running AMUSE

By now, you should have built and tested AMUSE, as described in Appendix E.12 (in particular see Appendix A.1), and are probably wondering "What can AMUSE do for me?" This section will get you started. If you're new to Python you'll find the official Python documentation[7] a valuable resource. Most of the scripts in this book were rewritten with at least Python version 3.7 in mind, but much of the coding will remain up-to-date with the current version 3.12 and higher.

You can run AMUSE interactively from Python or iPython:

```
%> python
```

Once Python has started, you can load AMUSE by

```
from amuse.lab import *
```

This line is a lazy but very quick way to import most of the AMUSE package in a single call. We generally do not advise it because it pollutes the Python name space. Rather, we recommend that modules are imported from AMUSE individually, and in the examples that follow this is how we will present source code.

The code examples and snippets presented in the book are designed to be cut and pasted into your Python interpreter or your file editor. The complete source code for running any of the simulations and for producing all generated figures in this book is available in the `amuse-examples` package, and once installed can be imported in the following way (using the `ae` namespace as shorthand for `amuse.examples`):[8]

```
import amuse.examples as ae
```

The complete source code of an example (e.g. `gravity_minimal`) can be seen with the `ae.print_example` function:

```
ae.print_example("gravity_minimal")
```

If you are using AMUSE from a Jupyter notebook, you can copy the example directly to a new cell, ready for editing:

[7] https://docs.python.org/3

[8] Examples are also online at https://github.com/amusecode/amuse/tree/main/examples/textbook.

```
ae.example_to_cell("gravity_minimal")
```

To run an example with default settings, you can use the `run_example` function:

```
ae.run_example("gravity_minimal")
```

We hope that these examples and code snippets encourage you to experiment and initiate interesting research projects.

1.3.6.1 Running AMUSE without network access

Running AMUSE without a network connection is curiously not always supported by your favorite operating system. The underlying reason hides in the requirement for MPI to have a network connection. This can be solved by using the smart sockets connection channel, or by using a loopback network. The latter can be done with the following command line:

```
 %> mpirun -n 1 --mca btl_tcp_if_include lo python -m amuse.examples.
   gravity_minimal
```

It's a rather complex command, but fundamentally, it boils down to running the MPI command `mpirun` using a single core with the specific command for process binding for the tcp protocol on your local network. This –so called– `MCA` command (`-mca`) can be invoked in mpirun with the arguments `btl_tcp_if_include` and an indicator for the local network (`lo`). This is a whole list of jargon, which may be hard to parse if you barely installed the AMUSE package, but maybe it shows that some of the things we do are not always as easily supported by local operating systems as we wish. There are quite a few other complex message-passing and scheduling issues if you desire to run AMUSE on larger machines, but since the architectures of such machines change within the mean time scale of a PhD (and on the time scale of a new release of this book), we will not dwell further onto this topic. [**This needs rewriting, or maybe scrapping and pointing to wiki/documentation – Steven**]

1.4 Internal data handling and communication

1.4.1 Particles and Particle sets

Before continuing on to some examples, a short digression on Python classes may be in order. If you are already familiar with this material, you can skip this and move on to the next section. For more detail, please consult Appendix F.

In our experience, most of the syntactic and mathematical features of Python (conditionals, loops, arithmetic operations, etc.—even the rigid use of indentation to define program structure)—are broadly intelligible to most programmers, regardless of background. However, some more complex data structures may be unfamiliar to readers used to less structured programming languages, such as C or Fortran. Accordingly, we pause here to elaborate on this topic in AMUSE, using some important data structures as templates.

A Python class is a data structure together with a collection of associated functions, or methods, designed to manipulate the data. A particularly simple AMUSE class is a `Particle`, which contains only basic particle attributes, such as mass, position, velocity, etc., as well as a key that uniquely identifies the particle in the AMUSE system. It has essentially no user-useful methods to do anything with the data. We can set or access its contents directly, with no constraints on what we add:

```
from amuse.datamodel import Particle
from amuse.units import units
p = Particle()
p.mass = 1.0 | units.MSun
p.mass2 = 42 | units.kg
p.name = "Christine Chapel"
p.rank = "Nurse"
print(p)
```

The `particle` class is not very useful in and of itself, but it forms the basis for a vitally important class called a `particle set`, which is basically an array of particle objects, with conventions regarding content and a collection of methods providing information on and control over the data. An element of a `particle set` is expected to have certain attributes, such as `mass`, `position`, `velocity`, `radius`, and possibly others. The built-in methods assume that these data exist.

We can create a particle set of 10 particles as follows:

```
from amuse.datamodel import Particles
particles = Particles(10)
```

We can then set up attributes in various ways:

```
particles.mass = 1 | units.MSun
```

sets all masses, while

```
particles[3].mass = 42 | units.kg
```

sets just the mass for the fourth particle (Python starts counting at 0). We could also have set the masses (or positions, etc.) in the `Particles` creation call, with

```
particles = Particles(10, mass=1 | units.MSun)
```

Some more useful ways of setting particle set attributes are presented below.

In some cases, one would like to expand the attribute list of a particle with an array. This can happen when one would like to add information about a chemical network to a particle, or some other attributes that normally is associated with particle data. Adding abundances, for example, for Hydrogen, Helium, Lithium and Iron could be realized as follows

```
particles.add_vector_attribute("abundance", ("H", "He", "Li", "Fe"))
particles.abundance = [0.75, 0.24, 0.005, 0.005]
```

In the first line we add a vector attribute called “abundance” to the particle set, and specify the array of names for this attribute. In the second line we initialize the values for all the 10 particles with the same chemical composition.

Once the basic attributes are set, the methods associated with the `particle set` provide commonly used functionality. For example,

```
energy_kin = particles.kinetic_energy()
```

computes the total kinetic energy of the particle set, while

```
energy_pot = particles.potential_energy()
```

computes the total potential energy. Similarly,

```
com = particles.center_of_mass()
```

calculates the center of mass of the particle set, while

```
particles.scale_to_standard()
```

places the set in the center of mass frame and scales it to a standard configuration (see Section 3.1.2.1). Many AMUSE functions take a particle set as input and return analytical information on it.

You can ask Python to list the methods associated with a class by using the `dir` function:

```
dir(Particles)
```

will provide a list of all the methods available in the `Particles` class. (Note the Python convention that names beginning and ending with double underscores are intended for private use, even though you can see them and in principle use them yourself—not recommended!) More extended information can be obtained with

```
help(Particles)
```

which provides a detailed description of the methods in the class.

Even here, though, Python asserts its inherent flexibility. Unlike classes in more strongly typed languages, Python classes are customizable. You can add your own variables at will for your own purposes—just don't expect them to be properly managed or preserved by the built-in methods!

1.4.2 Grid-based data

Technically grids are conceptually quite different than particles, but in principle they can represent the same information, at least numerically. Grids are also used in gravity codes (such as in tree codes Barnes & Hut (1986) and particle-mesh solvers (Hockney & Eastwood, 1988)), and they are quite common for hydrodynamics solvers. In principle the same information can be conveyed with particles, or with grids, but for a wide variety of reasons grids often turn out more complex because they are not necessarily Cartesian.

Voronoi grids, for example, represent a tessellated space, whereas a KDTree-type grid or a refined grid tend to have different data structure. These differences in the fundamental data structure makes it a complex to unify the grid-based data structures. We therefore project grids on the already present particle data structure.

Examples of grid based codes include Athena and FLASH. Both use adaptive grids, although, this is not how we initialize them. These grids are easiest initialized using a Cartesian grid, after which the code internally re-adjusts the grid structure to reflect structure of the gas (often the density or temperature structure).

A simple grid for the hydro-code Athena can be initialized as follows

```
from amuse.units import generic_unit_system
from amuse.community.athena import Athena
instance = Athena()
instance.parameters.x_boundary_conditions = ("periodic", "periodic")
instance.parameters.mesh_length = (8, 1, 1) | generic_unit_system.length
instance.parameters.mesh_size = (8, 1, 1)
grid = instance.get_extended_grid()
instance.initialize_grid()
print(grid.rho)
```

A more generic way of making a grid is

```
def make_grid(
    number_of_grid_cells, length, constant_hydrogen_density,
    inner_radius, outer_radius
):
    grid = Grid.create(
    [number_of_grid_cells] * 3,
        length.as_vector_with_length(3)
    )
    grid.radius = grid.position.lengths()
    grid.hydrogen_density = constant_hydrogen_density
    grid.hydrogen_density[grid.radius <= inner_radius] = 0 | units.cm ** -3
    grid.hydrogen_density[grid.radius >= outer_radius] = 0 | units.cm ** -3
    return grid
```

We can plot the particles of a gaseous distribution simply using matplotlib, as demonstrated in Figure 1.9.

In Figure 1.10 we show three representations of a gaseous distribution (right), as a Voronoi tessellated structure (left) and as an adaptive Cartesian grid (middle).

We construct and plot the Voronoi tessellation from the particle gas `particle_data`, using the `scipy` package `Voronoi`.

```
from scipy.spatial import Voronoi, voronoi_plot_2d
voronoi_data = Voronoi(
    particle_data, furthest_site=False, qhull_options="Qbb Qc Qz Gc",
)
voronoi_plot_2d(voronoi_data, ax, show_vertices=False)
```

The refined grid is plotted conveniently using the YT package (see https://yt-project.org/doc/index.html).

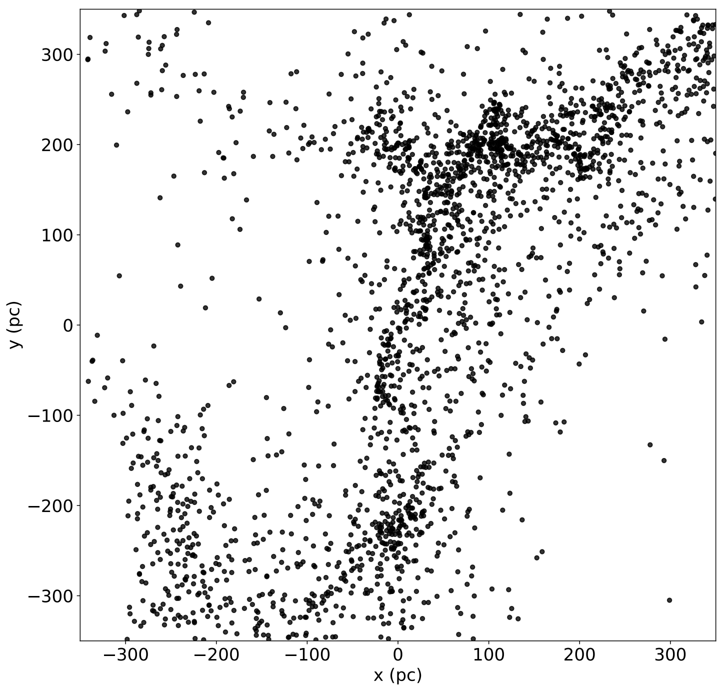

Figure 1.9: The calculation was performed using the Torch package in AMUSE. The projection is 700×700 pc. Tessellated, Adaptive grid and the gaseous distributions are presented in Figure 1.10. The script to generate this figure can be found in `ae.hydro.voramr_logic_progression_plots`.

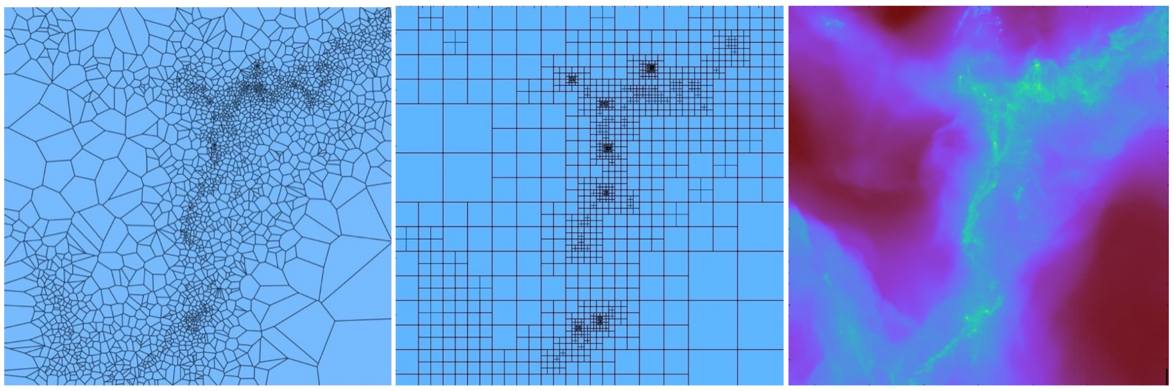

Figure 1.10: Voronoi tessellation (left), adaptive Cartesian grid (middle) and a gaseous representation (right) from a simulation performed using FLASH. The projection is 700×700 pc. in Figure 1.9 we show the particle gas distribution of the same filament. The script to generate this figure can be found in `ae.hydro.voramr_logic_progression_plots`.

```
AMR_data = yt.load("../data/lvl12-VorAMR_hdf5_chk_0001")
plt = yt.ProjectionPlot(AMR_data, "z", "bgpt")
plt.set_cmap(field=("flash", "bgpt"), cmap="arbre")
plt.annotate_grids(cmap="black_only")
```

Several codes adopted grids. Some of them solve for gravity, such as the various tree codes, and others solve for either hydrodynamics or radiative transfer. More on these codes in their respective chapters: Chapters 3, 5 and 6.

1.4.3 Data handling between code and script

Information is stored in `bodies` (mass, position, velocity, etc.), which are part of local memory and live in the user Python script. This data does not naturally live in the selected code, but the essential parts can be copied to the code. For a stellar evolution code this might be the mass and radius, and for an N-body code this could also include position and velocity. This local information is not automatically updated after the code has been run using `{code}.evolve_model()`. Here `{code}` may be any type of code for evolving stars or integrating equations of motion. To access the new data, we could copy it directly from the remote code by referencing `{code}.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 because it requires the construction of a channel. More importantly, it is less explicit. Note that some code may have various types of particles. This is not uncommon in particle-based hydrodynamics solvers, which may make a distinction between `dm_particles` for dark matter, and `gas_particles` for the gas.

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 for a gravity code is declared with

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

Here `gravity` is a pointer to the running code, here (considering the name) it is probably a gravity solver. To copy the particle data from the running 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.

1.4.4 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 1.4.3) before writing them to a file. A snapshot can be saved to a file named `filename` as follows:

```
write_set_to_file(bodies, filename)
```

Optionally, a "`format={format}`" argument indicates the format in which the snapshot if stored—by default "`amuse`", which uses `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 plain text ("`txt`").

Time is not an attribute of the particle set, but rather of the code, in this case the N-body integrator. As a consequence, the age of a particle is not stored by default. This generally saves data and the historical context stems from the idea that all stars are born at the same time. Some codes do not even allow for particles to have different ages, but in AMUSE we are not bound by these limitations. To save the time of a snapshot, we can use:

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

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

```
bodies = read_set_from_file(filename)
```

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

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

For storing the ages of individual particles, one can simply add a new attribute to the particle set:

```
bodies.age = 0 | units.Myr
```

or to an individual particle:

```
Sun.age = 4.5678 | units.Gyr
```

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

```
many_snapshots = read_set_from_file(filename)
for snapshot in many_snapshots.history:
    print(f"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,
    copy_history=False,
    close_file=False,
)
```

Alternatively, one could write compound particle sets to a single file and recover them thusly: [9]

```
gas, stars = read_set_from_file(
    filename, names = ["gas", "stars"],
    copy_history = False,
    close_file = False,
)
```

Yet another way, which is often used in planetary systems, is to make a distinction in the particle set by introducing an extra attribute:

```
stars.name = "star"
planets.name = "planet"
moons.name = "moon"
bodies = stars + planets + moons
write_set_to_file(filename, bodies)
```

After reading the file one can split the particle set:

```
bodies = read_set_from_file(filename)
stars = bodies[bodies.name == "star"]
```

and similarly for the other particle subsets.

Bibliography

Aarseth, Sverre J. 1999. From NBODY1 to NBODY6: The Growth of an Industry. *PASP* , **111**(765), 1333–1346.

Aarseth, Sverre J. 2011 (Feb.). *NBODY Codes: Numerical Simulations of Many-body (N-body) Gravitational Interactions.* Astrophysics Source Code Library, record ascl:1102.006.

Backus, John. 1978. *The History of Fortran I, II, and III.* New York, NY, USA: Association for Computing Machinery. Page 2574.

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

Beck, K., & Andres, C. 2004. *Extreme Programming Explained: Embrace Change (2Nd Edition).* Addison-Wesley Professional.

[9]**[I'm not sure this is dealing with the same problem – Steven]**

Belenkiy, Ari. 2010. An astronomical murder? *Astronomy & Geophysics*, **51**(2), 2.9–2.13.

Binney, James, & Tremaine, Scott. 1987. *Galactic dynamics.*

Calaprice, A. 2002. *Dear Professor Einstein: Albert Einstein's Letters to and from Children.* Prometheus.

Carol, L. 1871. *Through the Looking-Glass, and What Alice Found There.* Macmillan.

Dorman, .F. 2006. The career of Senenmut from Hatsehpsut. *Pages 107–109 of: From Queen to Pharaoh Ed. C. Roehrig.*

Dorman, P. F. 1991. *The Architecture and Decoration of Tombs 71 and 353.* Metropolitan Museum of Art Egyptian Expedition, New York.

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., & Tufo, H. 2000. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *ApJS* , **131**(1), 273–334.

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., & Tufo, H. 2010 (Oct.). *FLASH: Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes.* Astrophysics Source Code Library, record ascl:1010.082.

Gosling, J., Joy, B., & Steele, G.L. 1996. *The Java Language Specification.* 1st edn. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gosling, J., Joy, B., Steele, G. L., Bracha, G., & Buckley, A. 2014. *The Java Language Specification, Java SE 8 Edition.* 1st edn. Addison-Wesley Professional.

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

Holmberg, Erik. 1941. On the Clustering Tendencies among the Nebulae. II. a Study of Encounters Between Laboratory Models of Stellar Systems by a New Integration Procedure. *ApJ* , **94**(Nov.), 385.

Hunter, John D. 2007. Matplotlib: A 2D Graphics Environment. *Computing in Science and Engineering*, **9**(3), 90–95.

Hut, Piet, McMillan, Steve, Makino, Jun, & Portegies Zwart, Simon. 2010 (Oct.). *Starlab: A Software Environment for Collisional Stellar Dynamics.* Astrophysics Source Code Library, record ascl:1010.076.

L., Ewing, D., Nathan, & S., Anthony. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, **22**, 789–828.

Maassen, J., & Bal, H. E. 2007. Smartsockets: solving the connectivity problems in grid computing. *Pages 1–10 of: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing.* ACM. 1272368.

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

Ossendrijver, M. 2016. Ancient Babylonian astronomers calculated Jupiter's position from the area under a time-velocity graph. *Science*, **351**(Jan.), 482–484.

Pellò, Caterina. 2022. HYPATIA OF ALEXANDRIA FROM ICON TO HISTORY - (S.) Ronchey Hypatia. The True Story. *The Classical Review*, **72**(2), 701703.

Pelupessy, F. I., van Elteren, A., de Vries, N., McMillan, S. L. W., Drost, N., & Portegies Zwart, S. F. 2013. The Astrophysical Multipurpose Software Environment. *A&A* , **557**(Sept.), A84.

Portegies Zwart, S., McMillan, S. L. W., van Elteren, E., Pelupessy, I., & de Vries, N. 2013. Multi-physics simulations using a hierarchical interchangeable software interface. *Computer Physics Communications*, **183**(Mar.), 456–468.

Portegies Zwart, S. F., Makino, J., McMillan, S. L. W., & Hut, P. 1999. Star cluster ecology. III. Runaway collisions in young compact star clusters. *A&A* , **348**(Aug.), 117–126.

Portegies Zwart, Simon. 2011 (July). *AMUSE: Astrophysical Multipurpose Software Environment.* Astrophysics Source Code Library, record ascl:1107.007.

Portegies Zwart, Simon, McMillan, Steve, Harfst, Stefan, Groen, Derek, Fujii, Michiko, Nualláin, Breanndán Ó., Glebbeek, Evert, Heggie, Douglas, Lombardi, James, Hut, Piet, Angelou, Vangelis, Banerjee, Sambaran, Belkus, Houria, Fragos, Tassos, Fregeau, John, Gaburov, Evghenii, Izzard, Rob, Jurić, Mario, Justham, Stephen, Sottoriva, Andrea, Teuben, Peter, van Bever, Joris, Yaron, Ofer, & Zemp, Marcel. 2009. A multiphysics and multiscale software environment for modeling astrophysical systems. *New Astron.* , **14**(4), 369–378.

Ritchie, D. M. 1993a. The Development of the C Language. *SIGPLAN Not.*, **28**(3), 201–208.

Ritchie, D.M. 1993b. The Development of the C Language. *Pages 201–208 of: The Second ACM SIGPLAN Conference on History of Programming Languages.* HOPL-II. New York, NY, USA: ACM.

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

Springel, Volker, Yoshida, Naoki, & White, Simon D. M. 2001. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astron.* , **6**(2), 79–117.

Stroustrup, B. 2013. *The C++ Programming Language.* 4th edn. Addison-Wesley Professional.

The HDF Group. 2000-2010. *Hierarchical data format version 5.*

van Rossum, Guido. 1995a (Apr.). *Extending and embedding the Python interpreter.* Report CS-R9527. pub-CWI, pub-CWI:adr.

van Rossum, Guido. 1995b (Apr.). *Python library reference.* Report CS-R9524. pub-CWI, pub-CWI:adr.

van Rossum, Guido. 1995c (Apr.). *Python reference manual.* Report CS-R9525.

van Rossum, Guido. 1995d (Apr.). *Python tutorial.* Report CS-R9526.

Van Rossum, Guido, & Drake, Fred L. 2009. *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace.

Wilkinson, C.K. 1979. Egyptian Wall Paintings. *Pages 24–25 of: The Metropolitan Museum's Collection of Facsimiles.* The Metropolitan Museum of Art Bulletin, vol. 36.

Zuse, K. 1967. Rechnender Raum. *Elektronische Datenverarbeitung*, **8**, 336–344.

Zuse, K. 1993. *The Computer – My Life.*

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.