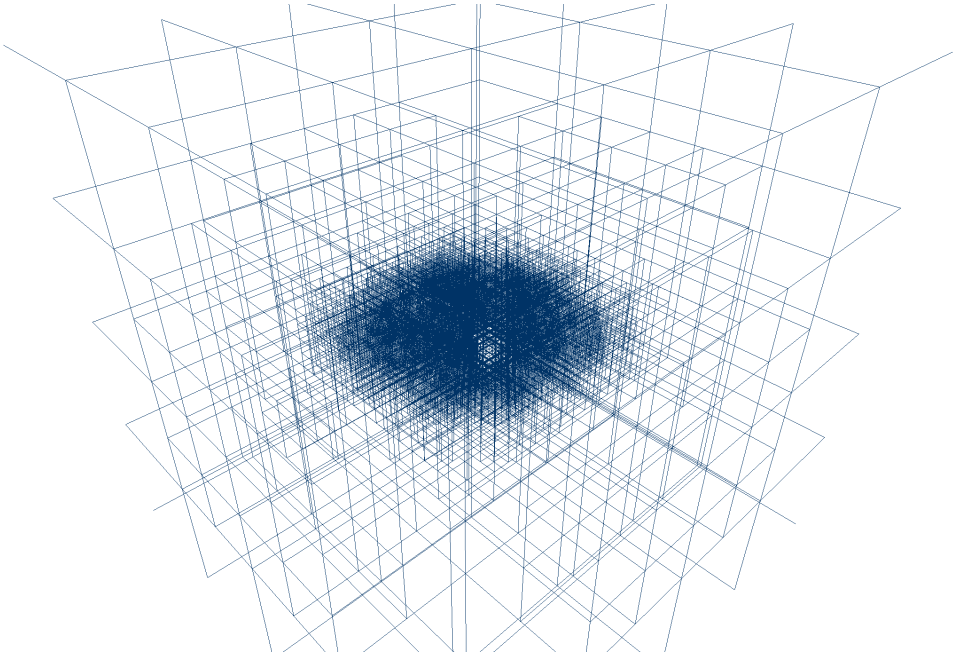
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Functional Methods to Set Up and Analyze Hydrodynamical Simulations of Star-Forming Molecular Clouds

Funktionale Methoden für Setup und Analyse hydrodynamischer Simulationen von sternbildenden Molekülwolken

Masterarbeit

Georg Michna

September 30, 2013

Supervisor:

Prof. Dr. Andreas Burkert

Contents

1 Introduction

This thesis examines a set of modern programming techniques, applying them to simulations of star-forming molecular clouds. It is composed of three main parts, each handling a distinct topic:

- Methods to efficiently set up, export, start, and analyze runs of a Smoothed Particle Hydrodynamics (SPH) simulation. Example implementations to use and test the Espresso simulator (introduced in the next section) are presented.

- Simulations of the radial density distribution of isothermal star-forming filaments using a custom one-dimensional grid simulator with cylindrical symmetry.

- Assessment of the F# programming language, Common Language Infrastructure (CLI) and the functional programming paradigm in general for use in physical applications.

The motive for a focus on programming practices stems from the increasing performance of simulators – and computational systems in general. When an expert group sets up and runs a simulation over months, it may not be significant how much time was spent on realizing particle distributions, managing simulations or transforming data. However, when a single student on a typical machine is able to run a simulation, such overhead may dominate the overall time spent.

The following analyses and experiments aim to manage and run simulations quickly, flexibly and with a lower error rate. The methodology is given deliberate emphasis compared to the algorithms and results, since its importance rises as simulations themselves become more efficient. I would like readers to see this shift as the overall theme that connects the only loosely related topics of this thesis.

1.1 SPH, GPGPU, and Espresso

Smoothed Particle Hydrodynamics, introduced by Gingold and Monaghan in 1977 [18], uses the positions and properties of particles to fully describe the state of a simulated fluid. Spatial grids, the central data structure to many other types of simulators, are not part of the representation; if they are used at all, they only serve as a performance optimization. This can necessitate additional steps to relate SPH states with analytical or abstract models; it can also complicate tasks such as visualization or the placement of particles. Chapter 2 handles this topic in detail.

The Espresso simulator is a not yet publicly released SPH simulator supporting general purpose graphics processing unit (GPGPU) execution that is currently under development by Martin Zintl [34]. GPUs are able to massively outperform CPU-based options, allowing simulations featuring millions of particles to run with reasonable speed on workstation computers. Much of the material presented in this thesis, especially in Chapter 2, is related to the testing and development of Espresso.

There is no official release of Espresso at the time of this writing. This thesis' code version control currently hosts Espresso; contact Martin Zintl [34] for information on a public release or me for access. Since the combined repository may contain unpublished material, permission to use code from this thesis by people outside the CAST group of the USM Munich requires Martin Zintl's consent. I can separately provide the code relevant to this thesis, which runs independently but is unable to run SPH simulations without Espresso.

1.2 Stability of isothermal filaments

The exact processes leading to star formation in molecular clouds are subject to ongoing debate. Observations of star-forming filaments in molecular clouds, conducted in a 2011 study by Hacar et al. [13], largely motivated Chapter 3. Studying

Figure 1.1: Shows a star-forming filamentary structure in the Taurus molecular cloud, as observed by the APEX telescope [2], in a visible-light image superimposed with submillimetre-wavelength observations (orange glow). Individual filaments of this cloud are the subject of studies by A. Hacar et al. [13] which in turn motivated simulations of such filaments' radial density profile detailed in Chapter 3. *Credit: ESO/APEX (MPIfR/ESO/OSO)/A. Hacar et al./Digitized Sky Survey 2. Acknowledgment: Davide De Martin.*

the L1517 dark cloud in Taurus, pictured in Figure 1.1, it has found filaments to be surprisingly quiescent, featuring coherent velocity distributions very different from their turbulent environment. This gives new insight into the star-forming processes in such molecular clouds, as dense cores that collapse into stars may be formed by fragmentation of such velocity-coherent filaments.

It also allows for more specific approaches in the theoretical modeling of filaments' structure, strengthening the view that isothermal cylinder models apply. Investigating the validity of this theory and applying it in numerical simulations of gravitationally supported filaments' radial profiles is the main topic of Chapter 3.

The model for comparison is an analytical equilibrium solution of the infinite, self-gravitating cylinder by J. Ostriker [24]. It has finite mass per length, as its density scales with r^{-4} for large r. Section 3.2 discusses its statements for the isothermal case and some of its implications for pressure-supported filaments. The results of these analytical considerations are then reproduced in dynamical simulations, under slightly dampened motion and sufficient run-time to relax into a stationary distribution.

1.3 Evolving views on programming

Recent years have seen a shift shift in *programming paradigms*, the thought models that gov-

ern programming and the design of its languages. Classical languages like FORTRAN have become more object-oriented, while object-oriented languages such as C# have become more functional. The evolution from C to the more object-oriented C++ to the introduction of some functional features in C++11 is an example of this development.

This thesis uses F# for most of its code and places an emphasis on functional programming style, which is briefly introduced in Section 4.2.

Hydrodynamics may be an especially interesting context for the application of F# and its emphasis on the functional programming paradigm. The book Expert F# 2.0 [7], a learning resource and reference used for this thesis, begins its introduction to the concept of imperative programming – the classical competitor to functional programming – by naming the simulation of fluid dynamics as a typical application. Other applications in theoretical astrophysics, such as the various n-body simulations ranging from asteroid orbits to globular clusters and cosmology, show a similar dominance. While classical, imperative programming in the style of FORTRAN and its descendants has certainly been an effective tool, astronomy – and, in my personal experience, physics in general – is so firmly in its hands that concepts from object-oriented and functional programming tend to be forgotten.

A special feature of F#, which is rarely seen in this quality even in other functional languages, is its support of *units of measure* in code. Implementations created for this thesis make heavy use of them. Section 4.5 provides some additional information about this.

1.4 Programming jargon

Due to the nature of this thesis, it includes programming jargon that may not be in common use amongst astrophysicists. I suggest that readers who are not at all familiar with the concept of *immutability* or *functional programming* read Section 4.2 early. It is only a rough introduction; trying to explain the entire field of functional programming within this thesis is not feasible. Still, it may help explain some of the atypical design decisions in this project.

Jargon can be especially confusing when it has a conventional meaning that differs from the intended meaning, such as the word "functional" when referring to the functional paradigm. Other terms to look out for are:

- *Method*: in object-oriented programming, a method is a function tied to an object instance. In the CLI, *static methods* still bear this name, even though they are tied only to a type, not an instance. F# supports and distinguishes between functions and methods, so the text uses either term depending on the implementation.

- *Record*: an F# record is a data structure aggregating named (and typed) values. Records are often immutable. F# provides useful syntax for creating a modified version of a record. Many data types in this thesis are implemented as records.

- *Module*: In F#, modules are a tool for structuring. They group code and, unlike other types, allow to expose functions (as opposed to methods).

- *CLI* and *CIL*: The swapped letters may seem like mistakes, but these are different – though related – concepts. The CLI is the *Common Language Infrastructure*; it forms the platform F# is built on. Section 4.3 introduces it. The CLI outputs a byte-code after its first compilation step that is called CIL, the *Common Intermediate Language*. Whether or not this pair of abbreviations is sensible, it is considered proper terminology and will be used throughout this thesis.

1.5 Weighing optimization against simplicity

When designing a project such as this one, reliability and simplicity sometimes conflict with performance. In most of these cases, I did *not* favor performance. For instance, all floating point calculations are done in double precision and there is no option to change this. Including such a "precision switch" would, in my opinion, cost users more time than the faster program gains them. The additional time-consuming steps users need to take due to increased complexity can be subtle and tricky to predict. Sticking with the example of a precision switch: much of the code would be affected and require additional documentation. Users would have to keep precision in mind as an error source. Type-safe implementation would become more complicated. The program run-times were just not long enough to justify such optimization. In a similar vein, random number generation uses an entropy

source of cryptographic quality by default. Extended versions of the library functions allow the user to change this, but in most cases, that should not be necessary. The cryptographic generator is reasonably fast, yet a single case of artifacts due to bad randomness can cost a lot of time. (See Section 2.2 for a description of the used random number generation.)

Generally, physicists' time is a rather expensive resource compared to CPU time. It should be included in speed considerations. Setup and analysis scripts usually take by far more time to write than to run. When facing the option of improving a program's speed at the cost of adding complexity, it is easy to underestimate the cost of the complexity. As Donald Knuth famously said, "premature optimization is the root of all evil." Even though this is a demonstration code without real users, it tries to balance between performance optimization and simplicity of the interface that users see – to respect the constraints a project with multiple active users would have. (And not be evil.)

2 Functional setup and analysis of SPH simulations

Modern programming methods have the potential to improve the efficiency of writing code in computational astrophysics. The SPH setup and analysis F# library that was written for this thesis, called SPHS library in the following, is intended as a concrete example of how this can be achieved. It provides tools and data structures that can be called from any F# program, or any CLI program when abandoning units of measure. It aims to allow efficient combination of these tools, both with each other and external programs.

One of the SPHS library's tasks is to allow the user to set new initial conditions as quickly as possible. The most intuitive way of defining initial conditions is often by analytical functions, such as density distributions, particle distributions, or velocity fields. Unlike grid-based simulators, SPH simulators cannot directly take such distributions as input. This leaves it to the user to create particles that approximate the desired distribution. To speed up the creation of initial conditions, the `SPHS.Make` module and a space partitioner are introduced. They provide functionality to easily specify grids, glasses, and arbitrary distribution functions. The outputs are 3D coordinate sequences which can easily and flexibly be transformed into particles.

2.1 Program structure and intended workflow

Many tools, such as the common plotting programs Gnuplot [11] and Octave [22], are written in one language and used in another. Their front-end languages are often oversimplified; they demote the user to a second-class citizen who does not have access to the full capabilities of the program's code. The user ends up writing analysis code in a different language, to then export the results into the tool's inferior language. The language is only used because it exclusively offers certain features, not because of its quality. The SPHS library attempts to avoid this. By using a well-designed general-purpose programming language as the front-end, it allows the user to introduce new features in the same way the library functions were written. Users can integrate external tools by writing wrapper functions, in the same way the library function `Simulation.RunLocal` can compile and run the Espresso simulator. Formally, there is no notable difference between using the library and expanding the library. Both simply add new definitions. Only the last step differs, in which a desired computation is fully defined and executed.

The library loads the following definitions in order:

- the EBT data format (See Section 2.5)
- CGS base units and astrophysical unit values (parsec, $M_\odot$, G, ...)
- geometry types for 3D vectors, cubes and cuboids
- a random number generator class (see Section 2.2)
- an SPH particle type and thread-safe ID generator
- tools and data types to run Espresso
- the `Make` module and partitioner (See Sections 2.3 and 2.4)
- 3D point cloud rendering and 2D function plotting (see Section 2.6)
- a 1D grid simulator featured in Chapter 3

Figure 2.1 shows an overview of the library's setup and analysis features. Even though it shows additional programs only as tools to process output, all

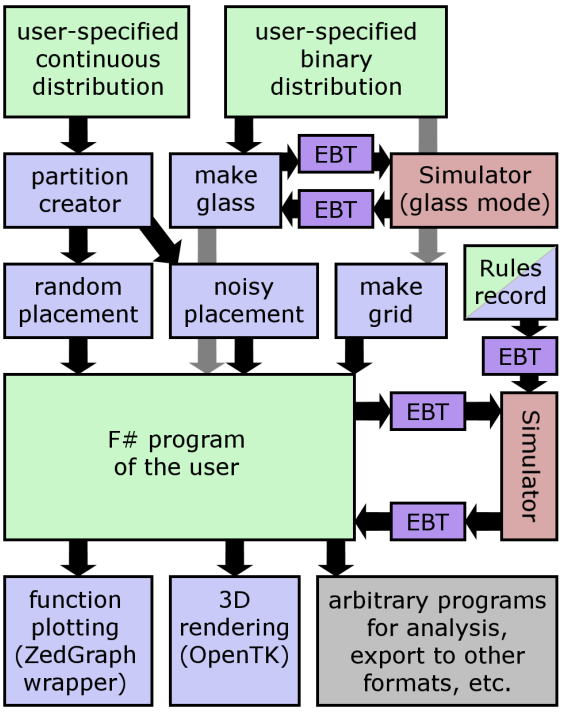


Figure 2.1: The data pipeline of the SPHS library in its intended standard use case. Arrows show information flow; grey arrows pass behind boxes. User-written parts are green, the SPHS library is blue with the EBT part violet, and the native/GPU Espresso simulator red. Usage of noisy placement by `Make.glass` is omitted as an irrelevant implementation detail. The `Rules` record has two colors as its structure is predefined, but its content set by the user.

user code can call external libraries if necessary – both CLI and native code (e.g. using F#'s Native Interop features).

The graphic shows how all functionality is arranged around the main program that is written by the user. Assumptions on this program have been avoided wherever possible; it could run multiple simulations or only use a subset of the library, using custom code in place of the remainder. Though its features rely on some common definitions such as the geometry types, the library consists of many parts that are independent in usage. Five such parts are visible upon closer inspection of Figure 2.1: continuous placement, homogeneous placement, simulation interaction, function plotting, and 3D rendering. In addition, the data types, random number features, and EBT file format could be used independently. (Though the other library features could not be used independently of these.)

Settings for Espresso are specified using the *Rules record*. (Appendix A.3 has a list of implemented rules.)

2.2 Random number generation

Many of the algorithms presented in the following rely on randomness. For convenient use of random numbers, a helper class is provided. It offers the creation of double-precision floating-point numbers uniformly distributed in $[0, 1]$ as well as three-dimensional vectors with unit-of-measure support, uniformly distributed in cubes or cuboids defined by using the library's vector, cube, or cuboid types.

The class is not thread-safe, but can be instantiated multiple times for parallel usage. This is done via one of three factory methods, which differ in their entropy sources:

- `Standard ()`: a cache-enhanced wrapper over `RNGCryptoServiceProvider`, a class of the CLI Runtime that creates very high quality random sequences.
- `Well512 seed`: a deterministic pseudo-random entropy source that returns the same sequence for the same value of `seed`, which is a 64-bit integer. *This is useful to create exactly repeatable processes*
- `Well512 ()`: similar to the seeded version, but uses the standard entropy source to obtain the seed.

Performance between the entropy sources is hard to judge as it varies, depending on compilation optimizations, CLI Runtime, operating system, and hardware.

Caching code that greatly improves performance and the `Well512` *pseudo-random generators have been supplied by Christian Winnerlein, whom I thank for his support. I also thank him for warning me of serious flaws in the .NET library's standard random number generator, which caused me to remove it from the selection.*

2.3 Setup of homogeneous shapes

Grids

The library's simplest shortcut function, `Make.grid`, creates any type of grid that can be expressed by a cubic primitive cell. The parameters it takes are a grid type, a bounding box, a binary distribution and the desired output density. The grid type is provided either by picking from three common grids – simple cubic, face-centered cubic and body-centered cubic – or via a list of occupied unit cell positions.

The algorithm scales the unit cell to have the desired particle density and repeats it periodically in all directions of space. It then fills the result into the given bounding box, yielding all positions where the binary distribution function returns *true*. This allows it to cut shapes from the grid. The coordinate system of the grid unit cell aligns with that of the output. `Make.grid` does not attempt to approximate the distribution function, calling it for every possibly valid grid position inside the bounding box. Therefore, if large voids exist in the bounding box and run-time should become a problem, performance can be increased by using smaller bounding boxes, calling `Make.grid` multiple times for separate objects if needed.

Glasses

It is commonly required to fill shapes with a *glass*, which is a fairly relaxed particle distribution that shows no distinct grid axes. Glass is useful since a simulation using it neither involves a sudden relaxation in the beginning nor shows anisotropic behavior due to the small-scale particle distribution.

Similar to `Make.grid` cutting from a grid, cutting from a glass distribution is possible with `Make.glass`. The function's signature is analogous to its grid-based counterpart, except that it – naturally – takes no grid type. Instead, it needs the simulator settings – provided via a Rules record – and a boolean determining whether to generate a new glass or use a cached one.

Despite being similar to the grid equivalent in usage, internally glass creation is a comparatively lengthy process. First a cube is randomly filled with particles, then the result is exported and shifted around by Espresso in *glass creation mode*, so that the glass corresponds to the physical settings of the simulation. The result is then imported back to be used as an infinite periodic glass. From this glass, the desired shape is cut. A typical next step would be to generate particles from these glass positions and export them again to finally start the physical simulation.

To initially fill the cube, the "noisy" semi-random placement algorithm for arbitrary distributions (described later in this chapter) is called with a homogeneous distribution. This creates a fixed amount of positions, currently set to 10,000, at each of which a generic SPH particle is made. To manipulate these particles in Espresso, simulator rules are prepared by copying the rules given as a parameter, with the following rule changes: the simulator output is reduced to positions, boundaries are set to the cube in which the particles were placed, glass steps become 1,000 and physical steps 0, periodic boundary conditions are set on all dimensions, and gravity is disabled. *Note that this implementation may require to be edited if further simulator features are introduced. Any member of the rules record will be copied into the settings of the glass creation run, so the implementation should disable all available settings that can interfere with glass creation.*

The new rules are exported, together with the generic particles, into an EBT formatted file (see Section 2.5), which is passed to Espresso. After the simulator finishes compiling and running – Espresso may need to recompile itself depending on the settings – its output positions are re-imported. The simulator output is also kept on disk as a cache. If a cache file is present and the glass generation function is called with the parameter for enforced glass generation set to *false*, all steps up to the import from file are omitted and the cached glass is used.

Now that we have a periodic cube of glass, the next steps are identical to `Make.grid`: the glass is infinitely repeated and positions accepted if they match the distribution. *In fact,* `Make.glass` *should call* `Make.grid` *at this point. It does not merely for historical reasons. This restructuring step was omitted due to time constraints concerning testing. The current implementation is equivalent though; the change would only remove a repetition of code.*

Make.glass is intended for 3D use but does have minimal handling of 2D simulations. It does correctly recognize 2D rules and disables z-repetitions accordingly. However, due to the unit-of-measure system being fixed at compile time, density units and their interpretation remain three-dimensional. To comfortably use this feature in 2D, the function could be split into a 2D and 3D variant with different unit-of-measure signatures.

2.4 Placement of arbitrary distributions

The general form of a distribution is a density function over space. Most such distributions cannot be reasonably approximated by homogeneous shapes as done in the previous section, so a more general method is required.

A simple but rather slow random placement algorithm works as follows: it generates pairs of random positions and numbers between 0 and the maximum of the density distribution. For each such pair, it looks at the density distribution at the random position and compares it to the random number. If the random number is below the distribution, this position is accepted into the output, otherwise it is discarded. This process is repeated until enough positions have been found. This is feasible for roughly homogeneous distributions, but becomes very inefficient if empty space or high peaks in the distribution are involved. Since the algorithm must account for the possibility of hitting a peak in the distribution, the probability of placing a particle after randomly selecting a location elsewhere becomes very low. This can make this method of placement unacceptably slow. Another inconvenience is that the maximum of the distribution is required from the start to run the algorithm.

The SPHS library provides quick random placement following an almost arbitrary density distribution. It is a two-step process: first, the distribution function is examined to create an object roughly describable as a partition of space, organized for particle placement. In the second step, this partition object is used by a placement algorithm, of which two are introduced, to get the final positions. For a given use case, it is fairly easy to wrap the process into one function that takes a *physical* density and outputs *particles*. The general implementation presented here is agnostic to its physical use case to improve flexibility; it could in principle be used for other distributions than particle density.

This is actually the third random placement algorithm implemented for this thesis. The first was a single-step system that had similar abilities but was replaced due to its excessive parameter complexity, while the second was a different formulation of the version presented in the following. Neither is of special interest as their entire functionality would be redundant with the library functions introduced in this chapter.

As an example, let us define a disk with exponentially declining density in both its radius and distance from the disk plane. This is roughly what one might find in a spiral galaxy's disk. Our goal is to obtain a particle distribution that approximates it, as shown in Figure 2.2. A density distribution can be expressed as an F# function that maps a 3D position to a dimensionless positive value:

```
let disk (p : Vector3<cm>) =
    let r = (p.WithY 0.0<_>).Length()
    let h = abs p.Y
    exp (-h / hScale - r / rScale)
```

This function takes a 3D position vector and maps it to the resulting density. In this case, we decide to use CGS units for the disk, hence positions are denoted in centimeters. The disk lies in the xz-plane, with `hScale` and `rScale` defining the scale heights in cylindrical axial and radial direction respectively. Note that this function makes no statement about the absolute density, but only about its relative distribution in space, so multiplying the last expression with a constant factor would not yield a different distribution.

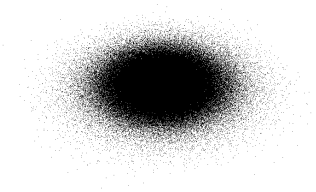


Figure 2.2: Output positions of the partitioner using a distribution function exponential in xz-radius and y-direction. See Figure 2.3 for the space partitioning used. See Figure 2.7 for a radial quality check of the distribution.

Partition

To raise the efficiency of particle placement, it is useful to distinguish interesting volumes of space, such as an irregular surface, from volumes of space with a simple distribution, such as empty areas. This is the purpose of the partitioner. It splits space into cubes similar to refinement in an octree, examining each cube to determine whether it

is *interesting*, meaning difficult to use for random placement. It refines such cubes into smaller cubes until either they are no longer interesting or their density integral is estimated to be small compared to the total distribution's integral.

Cubes are examined by sampling the distribution at random positions. The algorithm tracks the amount of samples, the maximum value found, and the aggregate of all found densities. The average density divided by the maximum density is the efficiency for random placement: it is equal to the probability that a generated position is discarded. In overview, a refinement step:

- obtains an estimate of the global density integral from the previous step
- distributes unfinished cubes to worker threads for parallel handling
- splits unfinished cubes into 8 smaller cubes
- samples the new cubes
- discards cubes that are empty for all samples
- regards cubes as finished if their expected efficiency is above the target value
- regards cubes as finished if they hold little content compared to the global distribution
- queues remaining cubes as unfinished work for the next step

This means that the partition is not an actual octree, but only the last layer of it with empty boxes removed.

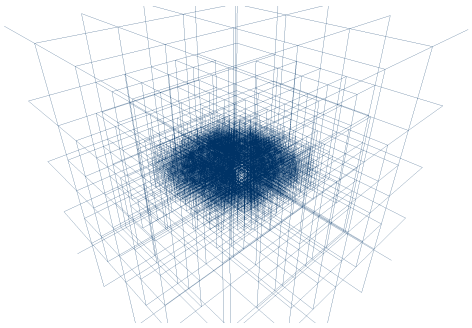


Figure 2.3: Partitioner output of the example disk. The outer cubes do not get deleted as they are not perfectly empty; a particle may appear in them with very low probability.

Figure 2.3 shows the cubes created by the partitioner by visualizing six edges per cube. Refinement parameters are set to: an efficiency target of 40%, a density integral limit of 10^{-5} of the total estimate (this defines "holding little content" in the previous list) and 8M minimum samples. These settings work for many distributions; they are used when calling the shorthand version of `GeneratorPartition.Create`.

Binary probability tree

The partition's final data structure should be suitable to select boxes randomly in accordance with the density distribution. To achieve this, each cube is assigned a unique interval within $[0, 1]$ and placed in a binary tree sorted by these intervals. Any value $\in [0, 1]$ maps to exactly one cube and the size of each cube's interval scales linearly with the probability of random position ending up in it if placed according to the given distribution. The binary search tree is constructed on top of the arbitrarily ordered cubes. Nodes contain two sub-trees and the boundary value separating the intervals of each. This allows navigating to the cube corresponding to a value in $\mathcal{O}(\log n)$, where n is the number of cubes.

Leaves of the binary tree hold two pieces of information: the cube's location and the maximum sampled density. Other statistics are discarded. The partitioner outputs only the binary tree, which is not ordered with respect to space.

Random placement

Given the binary tree introduced above, using it for random placement is simple. First, the algorithm chooses a random number in $[0, 1]$ and navigates to the corresponding box. Then it tries to randomly place a particle until it succeeds: it chooses a random location in the cube and a random number between 0 and the distribution maximum of the cube. If the number lies below the density at the tested position, the position was found.

The implementation has some additional tweaks. It is trivially parallel for multiple particles: a thread that is generating positions – using its own random number generator – does not interfere with other threads that use the same partition. Therefore, random placement runs multithreaded by default. The program also limits placement attempts after a cube has been selected, eventually choosing a new cube if it does not succeed. This way the program terminates in reasonable time if the partitioner, by chance, hits a very small peak in an

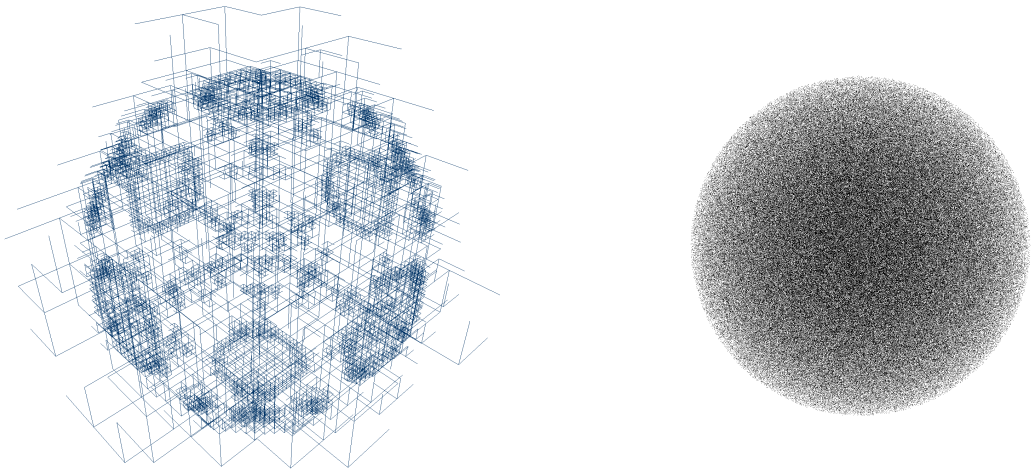


Figure 2.4: Partition and randomly filled positions of a hard ball distribution. Higher refinement is only necessary for volumes containing the surface: cubes further outside are empty and get deleted, those further inside are efficient for placement anyway.

otherwise empty box and the placement algorithm fails to find it.

Using random placement on the disk defined at the beginning of this section, which is done by using `Make.random` on the partition of Figure 2.3, creates a distribution such as Figure 2.2. The complete process to obtain randomly placed particles for SPH would be:

1. Define the distribution function
2. Construct a `GeneratorPartition` using one of its `Create` methods
3. Call `Make.random` (or `Make.random_ex` for additional options) with the partition and desired position count. (An alternative algorithm will be introduced later in this section)
4. Map the positions to particles using a custom mapper function that determines particle properties.

Quality test and benchmark

To test the algorithm under extreme conditions and compare implementations, a hard-to-select benchmark distribution was created. It is a combination of two shapes: the outer shape is a fractal that cuts lengths into thirds in each step and dimension, setting the density to zero in some of the 27 resulting cubes. The inner shape consists of sharp-edged fragments of a thin spherical shell. Since steps of the partitioning algorithm cut lengths into halves, forming eight cubes each time similar to an octree, the partition does not align favorably with the fractal – and not at all with the shell fragments.

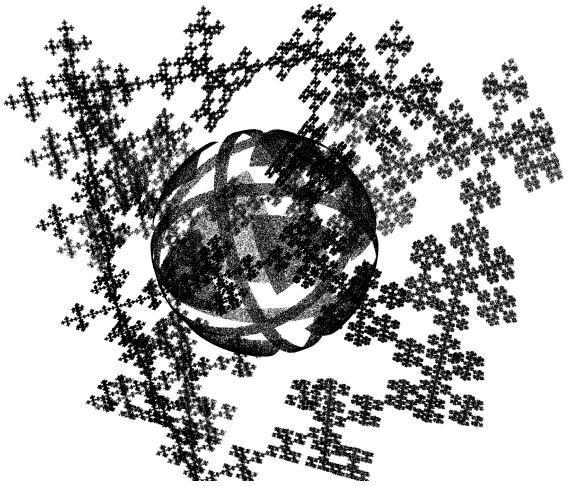


Figure 2.6: Benchmark output distribution

Figure 2.5 shows a partition based on the combined distribution. Thin edges and tips are hard to select with this algorithm, so it is reasonable to assume that most physical applications will show better results than this benchmark. The symmetry and clear geometry of the shape makes it easy to

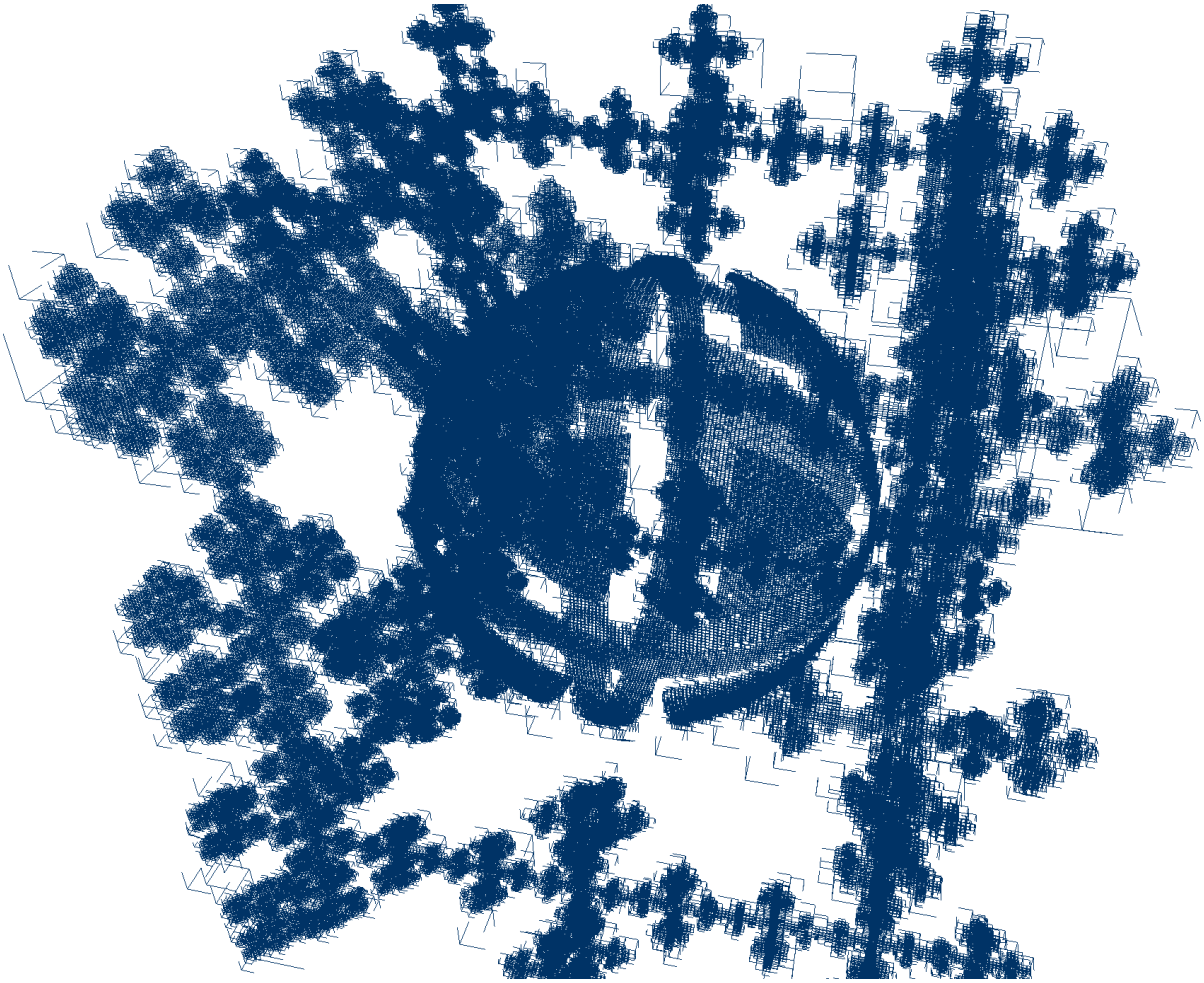


Figure 2.5: Partition of an extreme shape used to test and benchmark methods for particle placement. The fractal has 5 iterations of which the first has the shape inverted. The shell pieces have a width of 0.18% of the box's edge. Partitioner samples per refinement layer have been raised to 40M (5 times the current default) to reliably reproduce this structure.

see flaws in particle placement when displaying a rotating image of it. This was useful in algorithm development.

The algorithm produced the output shown in Figures 2.5 and 2.6 in 45 seconds in cryptographic random number quality using the Microsoft .NET 4 runtime on an AMD FX-8150 CPU. Performance was limited by two factors: First, due to a compiler bug in Visual Studio 12 that appeared late in the writing of this thesis, enabling tail call optimization slowed down a frequently run part of the algorithm instead of accelerating it as it should. Therefore, the optimization was disabled. Tail calls are a major feature of functional compilers; their unavailability may impact performance. This problem is likely to disappear in future compiler versions. Second, an optimization shortcut – skipping sampling if refinement becomes very likely – was omitted in the implementation. Fast placement of overly complex distributions was not a priority for this work, because after the optimization introduced by the partitioner, the execution time of the presented algorithm was no longer the bottleneck in a typical simulation.

Reproduction quality of the partition in terms of the position distribution appeared flawless in tests. Figure 2.7 shows its precision for the radial direction of our example disk.

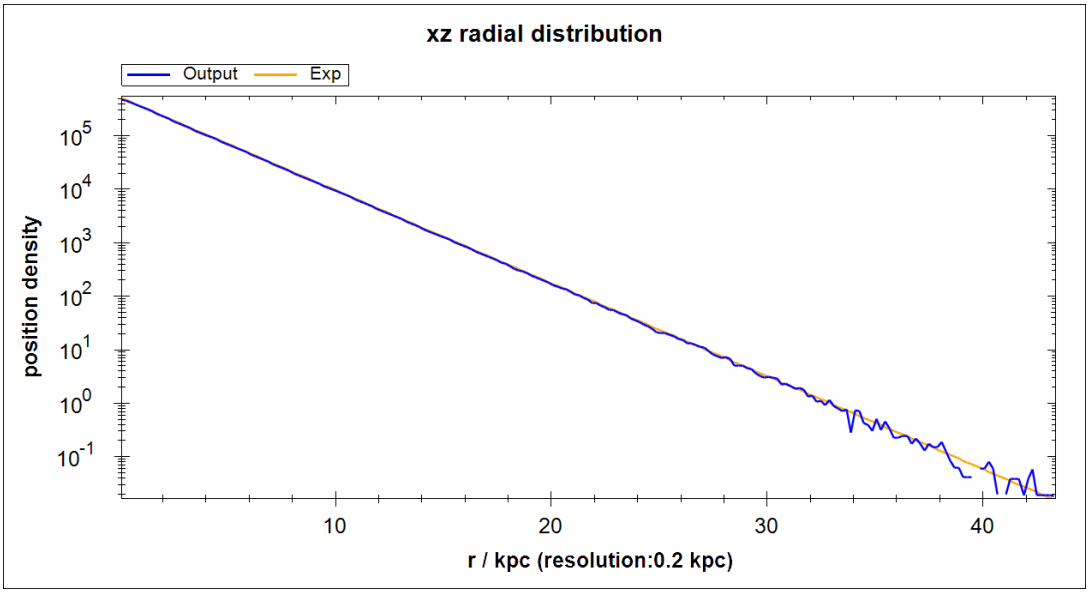


Figure 2.7: Quality check of the disk partition in Figure 2.3, measuring the radial density distribution, spanning about seven orders of magnitude. The orange line is the function $Ce^{-r/\texttt{rScale}}$, where $\texttt{rScale} = 2.5$kpc is from the input disk definition and C scales it with particle count. The partitioner was set to its standard precision of 8M samples (and polled for 20M positions for the plot). Random noise at low particle density is expected in random placement, so this result can be considered a perfect reproduction of the input radial density function.

"Noisy" placement

Randomness is not always desirable on all scales. The random placement algorithm may leave boxes empty even though it estimated from earlier sampling that they would typically yield multiple positions for particle placement. This is the correct behavior for true random placement, where randomness is equally present on all scales. For cases in which this is unwanted, the *noisy placement algorithm* presents an alternative. It is similar to random placement in usage, but reduces the randomness between cubes of the partition.

Let N be the number of particles to be placed. We want to give each cube at least its fair share of particles:

$$n_{\min}(\text{cube}) = \left\lfloor N \frac{\int_{\text{cube}} \rho(\vec{x})\mathrm{d}V}{\int_{\text{global}} \rho(\vec{x})\mathrm{d}V} \right\rfloor \approx \left\lfloor \frac{N \cdot \texttt{boxdint}}{\texttt{gdint}} \right\rfloor$$

Here, `gdint` is again the partitioner's global density integral and `boxdint` the corresponding estimate for only the relevant cube. Given sufficient samples, the last expression approximates the actual integrals, except for extreme distributions which are not relevant here. For each cube, the noisy placement algorithm approximates $n_{\min}$ in this way and generates the according amount of positions. Then it places the remaining positions randomly.

This implementation is somewhat minimalistic; it does not remove small-scale noise. This becomes more sensible when put into context: Espresso allows specifying a number of glass steps to be executed before simulating. These glass steps move particles a small distance into a more relaxed state without affecting their velocities. As random placement sometimes positions particles very close to each other, small-scale relaxation from this unphysical initial positioning occurs when the simulation starts. This problem can be significantly reduced by only a few steps of glass-forming movement. Such local, low-grade glass formation is useful in conjunction with the noisy placement algorithm, as it smoothes out any small-scale noise. Combining the two keeps the distortions to the distribution caused by glass steps small and retains randomness. *Note though that the current partitioner considers*

boxes with high density "not interesting" if they are efficient and ceases to refine them. This may stop noise suppression on medium scales of simple distributions. Should this library be used in the future and this feature become important, edit the `interesting` (`PGenBox<'u> -> bool`) *function of the partitioner to consider boxes holding a large fraction of* `gdint` *(the global density integral estimate) interesting. The threshold fraction could become another parameter or be influenced by the sample count.*

There are experiments to allow the passing of samples of the density function to Espresso. This data could be used during glass steps to further reduce the distortions they cause in continuous distributions. This has not been pursued further for this thesis, as the increased precision was not necessary.

Possible extensions

In principle, many further placement algorithms using the partitioner are conceivable. If the previously mentioned modification to the partitioner is applied, so that the total density integral per cube can be capped, the resulting partition could be used to create especially homogeneous distributions instead of random ones, for example by simply sampling down to a number of cubes on the order of the number of particles. *Though this particular method would not offer very good performance.*

An internal setup program of Espresso influenced by the placement algorithms of this thesis uses a Peano space-filling curve and makes moves a distance along it according to the density found at the previous sampling point. This creates smooth distributions but has a bias to "jump" too deep into sudden density spikes. Using the partitioner and then applying this algorithm to individual cubes, configured to match the locally desired particle count, may allow to automatically fill arbitrary distributions both precisely and without small-scale tension that needs to be smoothed out.

2.5 EBT data format

To allow the export and import of simulator states between the user's program and the simulator, as well as storage of unprocessed results, the library uses a custom binary file format named Easy Block Tree (EBT). The decision to use a new format is a trade-off: an existing, extensive format such as HDF5 [14] could perform the same task while providing tools and compatibility, but would also introduce much additional complexity and provide features of little importance to the given use case, such as the ability to efficiently alter files already on disk. In contrast, text-based formats such as CSV [5] waste performance and disk space if uncompressed. Fixed blocks of binary data (e.g. the GADGET2 format [10]) are not easily extensible, i.e. may break compatibility on minor changes. During development done for this thesis, the implementations of the setup scripts, analysis tools and simulator were often changed independently and in parallel, so a certain level of compatibility after changes was useful.

The main features desired for the format are:

- *Simple:* a small program should be able to parse files to find and extract a piece of data. Code complexity should be minimal.
- *Compact:* large data blocks should be efficiently readable and induce minimal overhead
- *Extensible:* it should be possible to include additional types of data without breaking compatibility with existing parsers
- *Transparent:* the format must allow any data to be stored without modification
- *Indexing:* means for quick navigation of large files should be provided

Structure

The idea of EBT is to only structure data and otherwise be as simple as possible. High-level meaning of data depends on context and need not be understood by programs working outside this context.

Concretely this means that EBT is recursive: data is stored in a sequence of blocks, which can again be a sequence of blocks, and so forth. This effectively creates a tree-structure, hence the name *block tree*. Each block is marked with a *block type ID* which signifies the meaning of the data within. This is where context dependence comes into play: if a block contains an EBT sequence, the meaning of IDs inside it depends on the container block's ID and consequently on any blocks further up the hierarchy. Further specification details, such as whether block order matters or whether IDs are unique, are deliberately undefined in the general case to allow exploiting context.

The blocks come in two sizes, one with a 10 byte header and one with a 3 byte header but a low limit of content length. Figure 2.8 shows their exact composition. See appendix A.1 for the specification.

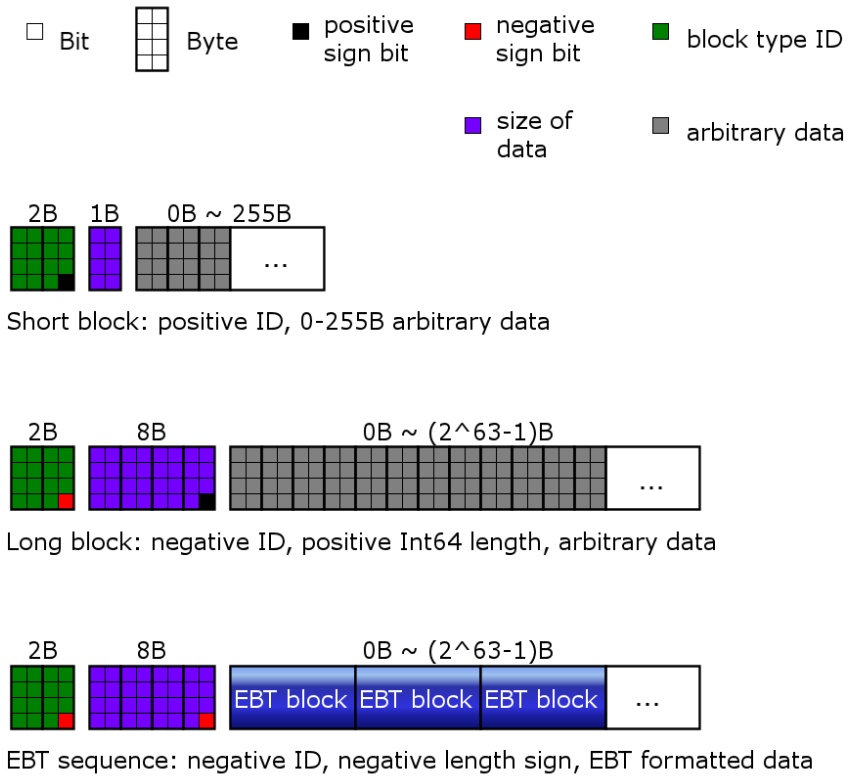


Figure 2.8: Composition of EBT blocks. Each block has an ID and indicates the length of its contents. The format comprises three types of blocks: a small one limited to 255B length of data, one that allows for large amounts of data, and one that contains a sequence of smaller EBT blocks. The flags to distinguish between them are encoded in the sign bit of the ID and long size respectively. The smallest possible blocks are 3 Bytes in length, end in 9 zero-bits and hold no information except for their ID and existence.

Index

Fast seeking is provided as an optional extension. It introduces a complete or partial index of EBT blocks in a file. A minimal summary of the index design targets would be: *quick, easy to read, flexible to write*. More specifically, it should:

- accelerate navigation to a known address in an EBT file, where the address is a sequence of block type IDs and integers that specify the amount of block occurrences to skip. *Such an address would read similar to "The third planet in the fifth system in the first galaxy in the sixth cluster of the file". Here, planet, system, etc. would be denoted by block type IDs.*
- allow the index to be written such that it induces minimal hard disk overhead (can be read in one chunk).
- allow flexible writing of the index so that it need not be completely rewritten on changes and is not fixed to a position in the file, which can be inconvenient for certain writing algorithms.
- allow partial indexes; indexing of all blocks is not required when the index is used.
- not complicate the code that reads the index unnecessarily.

The indexing feature defines three types of blocks: an index address block, an index container block, and individual index blocks. Index blocks can list block sequences in order, holding each child block's ID, position in the file, and the address of its index

block if present. A top-level container block is introduced that can be used to hold the entire index. An index container can be written at any position in the file and pointed to from the first block in the file.

Enabling a parser to use the index is fairly simple. It must check whether the first block indicates an index. If so, it navigates the index instead of the actual blocks until it either finds what it is looking for or hits a block that is not indexed. In the latter case, it enters the lowest block it found by index and then parses the child blocks normally.

The design idea behind the index is to allow accelerating the most time-consuming seeking operations without unnecessary interference with the rest of the format. Allowing to store the entire index at one location in the file works towards this goal in two ways: first, it improves performance, as the entire index can be quickly read from disk in one chunk. Second, if all sorts of blocks held indexes, IDs would have to be reserved in all blocks for that purpose, which is not the case with a central index. This should typically mean that the index is written at the end of a file.

The index definition is not confined to this type of usage though. Since index blocks reference each other by file address, the sub-index need not be in the same container. This would make it possible to pack lots of indexed EBT files into one file and write a "super index" that jumps to the individual indexes after the first level, removing the need to reorganize the individual contained files' indexes or to write a second index. All addresses of the individual indexes would have to be shifted by the original file's position in the new container though.

This detail may still be suboptimal, but it is not relevant for our use case. I consider continuing the development of the EBT index specification, for a wider range of uses, after the completion of this thesis. Generic EBT blocks themselves are already agnostic to their position in a file, memory, or an arbitrary stream. It may be useful to modify the index format to fulfill this condition as well, such that lower-level index blocks remain valid, independent of changes in higher-level block positions.

The technical definitions of the index are omitted here, as it is merely a performance optimization and longer than the main format definition. Please refer to appendix A.2 for the specification.

SPH-specific format definitions

Using the abstract format defined above requires concrete block meanings. A simulation file for Espresso can be built using the following top-level blocks:

- a *rules block* with simulator settings and global properties.
- *snapshot blocks* that hold particle data at a given time within the simulation. Initial conditions of particles are also exported this way.
- the previously explained *index container block* and the *index address block* to point to the index.
- internal blocks used for Espresso development, such as a benchmark and a debug block.

Each of these are again EBT sequences; the index capability was used for navigation. This structure is used for both input and output of simulations, with input including the rules block and one snapshot block that represents the initial conditions.

Figure 2.9 shows an example structure of an EBT file containing simulator output. The graphic shows many different types of information stored, but depending on the context repetitions of the same block type can be allowed. As indicated by the *#1* for the snapshot, more snapshots can follow – at any later position in the file. Importantly, the EBT format fulfills its purpose of extensibility here. A hitherto unknown block type could be inserted anywhere and, unless the local block definition explicitly forbids it, it would cause no problems in using the remaining file normally. To my knowledge, no block currently has restrictions in this respect; the SPHS library and Espresso should skip over any additional information they do not understand without it interfering with their functionality.

A detailed listing of block contents is omitted here, as it would be excessive and quickly outdated. *While this thesis was written, Espresso was still under development and its format changed frequently. Its implementation of EBT – and thus also the in- and output of the setup library – did not yet match the final draft of the EBT format as seen in Figure 2.8 exactly: blocks with EBT content output by Espresso were not yet flagged as EBT formatted and such flags may cause errors. Also, Espresso was experimenting with a shortened version of the index and a variation of the format introducing 2 bit type information. Anyone interested in using the Espresso simulator is recommended to contact Martin Zintl* [34] *for up-to-date information.*

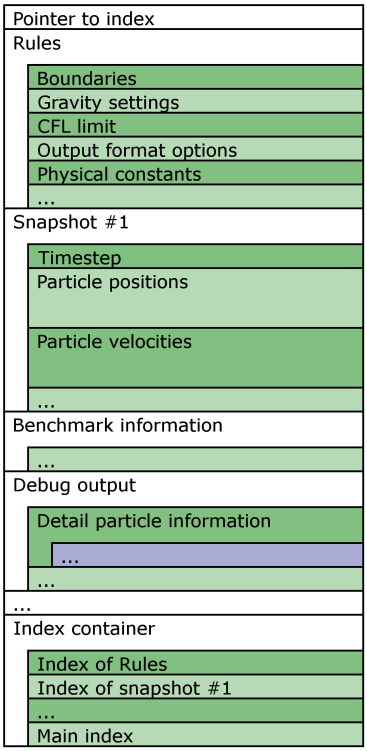


Figure 2.9: Shows the rough layout of an EBT-formatted output for Espresso. Much of the content is optional. Espresso can be set to not output unnecessary information. When creating input, a file with just the *rules* and *snapshot* top-level blocks is valid, as is skipping most rules to use defaults.

2.6 Integration of plotting and rendering

A common method to plot data resulting from the simulation is to output data points to a file in a primitive format, such as a sequence of coordinates in a binary block or in comma-separated text representation. Then, a separate plotting program (e.g. Octave [22], Gnuplot [11], IDL [16]) is executed, manually or by a separate scripting language. Such an approach is viable in our use case, but suboptimal, especially when coming from a high-level context: the extra conversion step complicates interaction between the main program and the plotting routines.

For this thesis, a simple wrapper over the CLI library ZedGraph [33] to create 2D-plots was written. In addition, a program to interactively preview simulation output as a 3D pixel cloud was implemented via OpenTK [23] (a thin wrapper over the OpenGL cross-platform graphics API).

2.7 Usage

Combining the modules of the SPHS library is achieved by writing a program that uses them in order. Remembering the slightly convoluted Figure 2.1 might make this appear more complicated than it is. In usage, most interactions are implementation details that happen in the background and need not be explicitly mentioned in the calling code. A basic simulation can be set up in the following steps:

- Write basic definitions to be used throughout the program, such as scaling factors and analytically known distribution functions.
- Create simulator rules, usually by referring to a common set of rules and applying changes where necessary.
- Obtain the particle positions by using one of the placement algorithms.
- Map the positions to particles with the desired properties.
- Execute the simulation with the finished rules and particles. The program waits for it to complete.
- Import results and run analysis code on them, optionally rendering the distribution or creating plots.

Figure 2.10 shows a concrete usage example of the library. It shows an F# function called `demo` that defines and executes a simulation. This function could be declared like this in any F# program, provided that it has loaded the main SPHS features and its CGS-based standard units, via `open SPHS` and `open SPHS.Units` respectively. *Of course, the library itself has to be present and referenced so the compiler can find it.*

The example simulation is very similar to the spherical collapse simulations in the upcoming sections 2.8 and 2.9. A glass with a particle density of 0.1 per cubic parsec is used to fill a sphere. Each of the particles has a mass of $10^3 M_\odot$, resulting in a physical density of $100 M_\odot/\mathrm{pc}^3$. When defining the rules, the program copies `Rules.DefaultSPH`, a set of rules defined in the library that contains typical rules for SPH. So any rules that are not specified are

```
01: let demo () =
02:     let name = "example"
03:     let box = Cuboid.CenteredCube (100. * parsec)
04:
05:     let rules =
06:         {
07:             Rules.DefaultSPH with
08:                 bounds = box
09:                 dt = 1e4 * year
10:                 snapshots = 201L
11:                 enableGravity = true
12:         }
13:
14:     let ball (pos : Vector3<cm>) = pos.Length() < 40.0 * parsec
15:
16:     Make.glass(true, rules, box, ball, (0.1/pc3))
17:     |> Seq.map (fun pos ->
18:         {Particle.Zero with Position = pos; m = 1e3 * MSun; e = 1e-4 * erg/g})
19:     |> Simulation.runLocal rules name
20:
21:     Plot.Interactive box name 0 200 0.0 1.0
```

Figure 2.10: A program that uses the SPHS library to simulate a homogeneous gas ball. Lines 7–11 define how the simulator settings called `rules` differ from the default, where `dt` is the time between output snapshots. Line 14 defines a boolean distribution: a ball at the center with 40 parsec radius. Line 16 uses the glass generator to obtain individual positions, which lines 17–18 map to a sequence of particles with fixed properties. *Note that the pipeline operator* `|>` *is used to pass the previous result to the next expression.* Line 19 exports the particles and rules and runs the simulation, waiting until it finishes and stores the results on disk (under "example"). Line 21 imports the results and launches an interactive program to examine snapshots 0–200. *The last two parameters influence color in rendering.*

filled with default settings. For example, the equation of state in the simulation is adiabatic. For the simulations in Section 2.8, the lines `isIsotherm = true` and `constEnergy = Some(e)` were added to the rules, where `e` is the constant specific energy for the simulation.

To readers not used to functional programming, the most alien part of the example might be lines 16–17. (These were originally written as one line that was split due to width restrictions in print, but either way is valid.) They take the sequence of positions from the glass generator call above and *map* the positions to particles, which are needed for the simulation in the line below. *Map* here refers to the higher-order function, which is a standard tool in functional programming. It applies another function – which it is given as an argument – to each element of a sequence, returning the sequence of results. `Seq.map` is the most general (least restrictively typed) version of *map* in F#.

The particle sequence is then passed to `Simulation.runLocal`, along with the rules and a name for output files. This function conducts the interaction with Espresso. Internally, it first re-writes selected source files of Espresso and exports rules and particles into an EBT-formatted file, then runs a batch file that causes Espresso to check whether its compilation state matches the desired one – if not, Espresso is recompiled – and runs the simulation. The function blocks its thread until Espresso terminates to allow waiting for the results.

The last line imports the results and launches an interactive colored pixel cloud renderer that can be used to watch the simulation progress. This interactive renderer is not given much attention in this thesis because Martin Zintl [34] is working on a

high-performance volumetric visualization program for Windows that can read EBT simulation output. It utilizes DirectX 11 for GPU acceleration and will provide interactive visualization for upcoming versions of Espresso.

Line 21 in Figure 2.10 can be seen as a placeholder for any kind of analysis code. Usually, a program to calculate properties of the results would be in its place, such as a comparison between results and analytical expectations. This is where, in the creation of this thesis, the plotting features from Section 2.6 were typically used.

2.8 Test: isothermal spherical collapse

When the setup library was first used in conjunction with Espresso, the latter's gravity implementation had not been active during recent code changes. It thus needed testing to ensure that the changes would not impede correct behavior.

A simple first test of gravity is to simulate the collapse of a homogeneous sphere with no or negligible pressure support. This case is especially suited because the acceleration of particles toward the sphere's center increases linearly with the particles' distance to the center, keeping the cloud's density profile homogeneous during the entire collapse. This is quite straightforward to show, assuming Newton's formulation of gravity and non-expanding Euclidean space – which is reasonable in simulation of this type and scale. First, we note that the gravitational potential outside a spherically symmetric body is equal to the potential of a point of the same mass at the body's center (see Gauß' gravity or Newton's Shell Theorem). Then, we calculate the acceleration and note that it is linear in radius as long as the density distribution remains homogeneous:

$$a(r) = -\frac{GM}{r^2} = -\frac{4Gr^3\pi\rho}{3r^2} = -\frac{4}{3}G\rho r$$
$$\implies a(r) \sim -r$$

Here, ρ is the gas density in the sphere and thus $M = (4/3)r^3\pi\rho$; G is the gravitational constant. At the beginning of the simulation, the sphere is at rest; it is neither contracting nor expanding. This means velocity is a linear function of radius, albeit with a slope of zero. From then on, at any point in time until the sphere has collapsed, velocity remains a linear function of radius: applying acceleration to the velocity means that we add two functions which are linear in the same parameter, which is r in this case. Such an operation always yields a function again linear in this parameter. The velocity remains linear in r, so the sphere remains homogeneous, which in turn means that the acceleration remains linear in r as well.

This implies that a very simple one-dimensional solver can already predict its behavior in collapse, providing expected results for comparison. Knowing that the sphere remains homogeneous, tracking only its outer shell is sufficient to calculate its properties throughout the collapse. An iterative solver can do this by updating only the radius and radial velocity of the outermost end. The gravitational acceleration at the outer shell of the sphere is just:

$$a_{\text{outer}} = \ddot{r} = -\frac{GM}{r^2}$$

where M is the sphere's total mass. This allows to integrate approximately:

- Update $\dot{r}$ using Δt and r
- Update r to $r + \dot{r}\Delta t$

Given sufficiently small time-steps, this provides the expected radii and densities of the sphere during the collapse.

There is an analytical solution as well [25]. *The analytical expression for the free-fall time* $t_{ff} = \sqrt{3\pi/(32G\rho)}$ *is used in the following to calculate the expected final collapse time.*

The corresponding simulation in Espresso is a sphere of 268,000 particles at negligible temperature, arranged in an FCC grid by the grid creator introduced in Section 2.3. Figure 2.11 shows a comparison of the density over time between the iterative solver and the Espresso simulation, spanning over 92% of the analytical time of collapse, showing nearly identical results.

As a consistency check for the multiple unit conversions between the SPHS library and Espresso, the collapse program calculated densities with different methods. One counted particles in a shell from 10 to 12 parsec from the center and multiplied the count with the constant particle mass of the initial setup, then divided by the sampled volume; the other set Espresso to export the physical densities used for SPH in its EBT output, then filtered particles from the same area the previous algorithm used. Both methods agreed very well with each other (data not shown as it looks near-identical).

What either of these two method did not do was to check the spatial homogeneity of the density. The sampling in a shell to plot density over

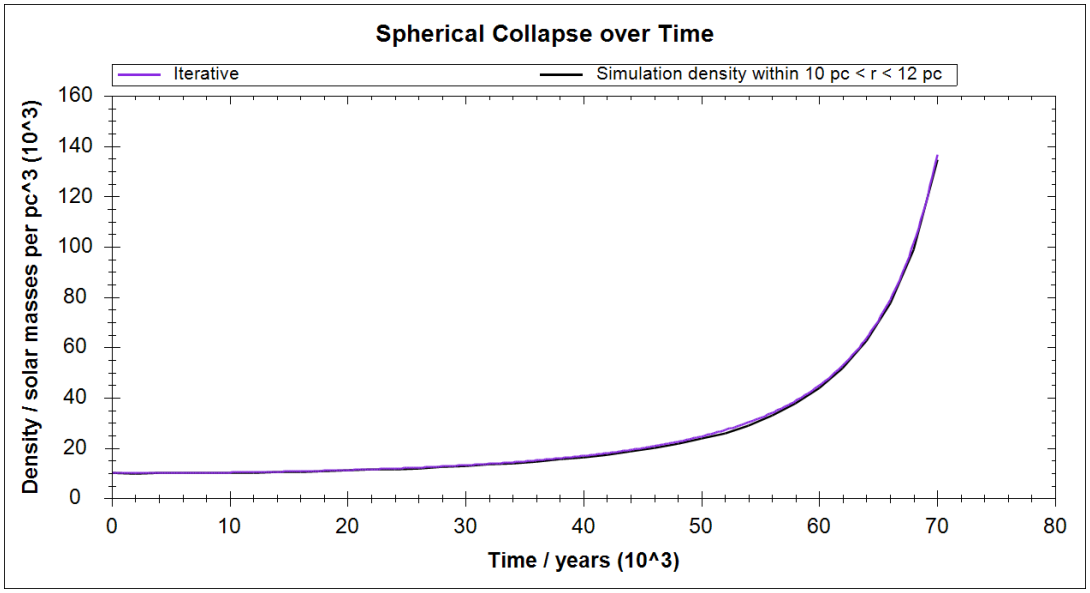


Figure 2.11: Collapse of an isothermal sphere at very low temperature in Espresso and the iterative solver, showing very good agreement. The sphere is far enough above the Jeans mass at its temperature to collapse freely.

time (for Figure 2.11) would have shown most major inhomogeneities as oscillations around the expected density over time. But it could miss fluctuations forming inside the shell and, though unlikely, in theory a strange inhomogeneity could offset another error and yield the correct result for the wrong reasons. To verify homogeneity, Figure 2.12 shows the radial density distributions at different times of the collapse, including the density-over-radius function produced by the iterative solver. With the exception of the outer edge, which is difficult to sample due to the particle-based simulation, the distribution looks very homogeneous and develops as expected.

The simulation was repeated multiple times with different parameters. The plot shown in Figure 2.12 comes from a simulation with a particle count of 2.68M, tenfold that of the original run. The difference in smoothness to the lower-resolution plot is minor (data not shown); 3D renderings of the distribution suggest that even this minimal difference is only caused by the sampling noise for the plot; the algorithm for its creation simply counted particles without weighting them.

Runs with deactivated hydrodynamics produce a similar result as those with a very small temperature at the beginning, but allow the simulation to continue into the final moments of the collapse. Analysis using 3D-visualization over time showed that the collapse time matches expectations to within 1% or better; it was exact within the precision of snapshot sampling used in that simulation.

Overall, the simulations reproduced the isothermal spherical collapse very well. Deviations from the expected values were small, yet still dominated by resolution and sampling.

2.9 Test: adiabatic energy conservation

The more or less purely gravitational collapses of the previous section are very specific tests of the implementation of gravity; with the exception of the late phases of some of the simulations, they ignore hydrodynamics and thereby the essence of an SPH simulation. For a wider test, another collapse setup was created: the heating and re-expansion of a contracting sphere with an adiabatic equation of state.

The purpose of the simulation is again to test Espresso, this time using gravity and hydrodynamics in conjunction and running analysis code that focuses on the conservation of energy.

In an adiabatic process, by definition no heat is exchanged between the considered system and its environment. Thus, in any simulation declared as

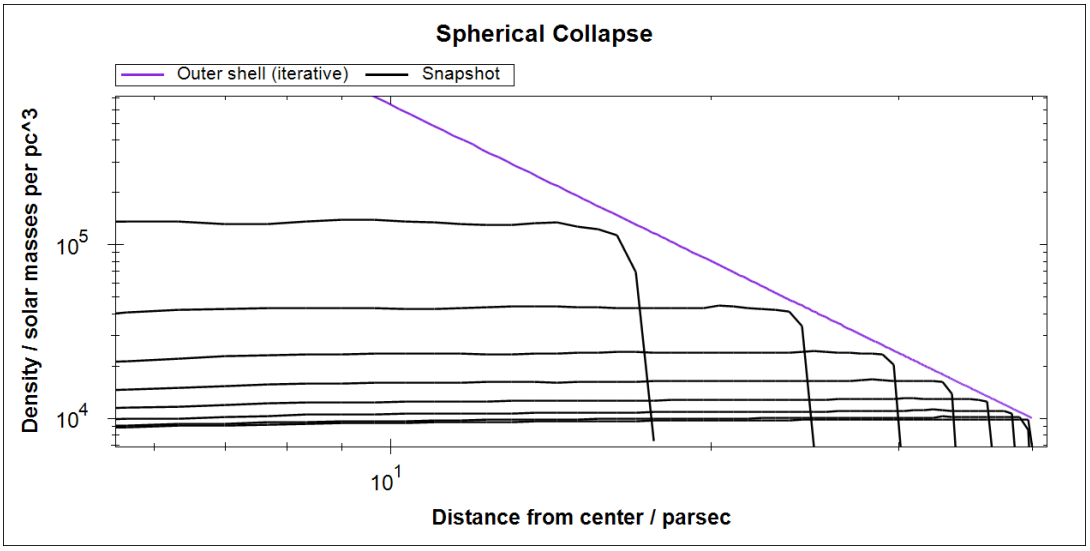


Figure 2.12: Radial density profile plots of different times in an isothermal collapse. For slightly better sampling, it was created from the 2.68M particles run. Of the 81 snapshots during the free-fall time, those dividable by 10 up to 70 are shown, each of which shows good homogeneity.

adiabatic, the total energy of an isolated gas cloud should remain constant over time.

For the purposes of the simulation, there are three variable energy components of the system:

- the thermal energy of the gas
- the kinetic energy of the particles
- the gravitational potential energy of the particle distribution

Since we are conducting SPH, the first two are somewhat intertwined compared to the physical reality. SPH particles substitute for a large quantity of actual gas particles, so the distinction between thermal and kinetic energy is an arbitrary choice defined by the size of simulated particles. In other words, physical processes that are no longer resolved become temperature. This is a general property of the concept of temperature and not an issue for the test at hand, as we only sum over both quantities, while the SPH implementation must ensure that the particles' properties correctly mimic the microscopic systems they represent. Though its meaning may be a little muddy, the total kinetic energy of the SPH particles, as measured from their mass and velocities, gives a good impression of the large-scale gas movement, so it is included separately in plots.

Obtaining the first two quantities – thermal and kinetic energy – from the simulator is very simple. Espresso is set to output the specific energies and velocities of all particles, then the analysis program converts these to energies and sums over all particles. The gravitational potential energy is not so simple. It is encoded in the positions of all particles, or, more precisely, all pairs of particles:

$$E_{\text{pot}} = \frac{1}{2} \sum_{i \neq j} -\frac{G m_i m_j}{|\vec{x}_i - \vec{x}_j|}$$

Calculating this sum (or half of it, by skipping the duplicate pairs that only differ in swapped indices) is called the *direct summation algorithm for gravity*. Such an algorithm is evidently $\mathcal{O}(n^2)$ and therefore not suitable for the execution of simulations with large particle numbers. Espresso therefore uses an octree to approximate the gravitational forces on particles.

For the purposes of measuring energy within a single snapshot, direct summation can still be used: since a sum over all particles must yield the correct result, sampling a subset of particles returns an estimate of the exact result of direct summation. Given a sufficiently large, randomly selected sample, this can be used to approximate the grav-

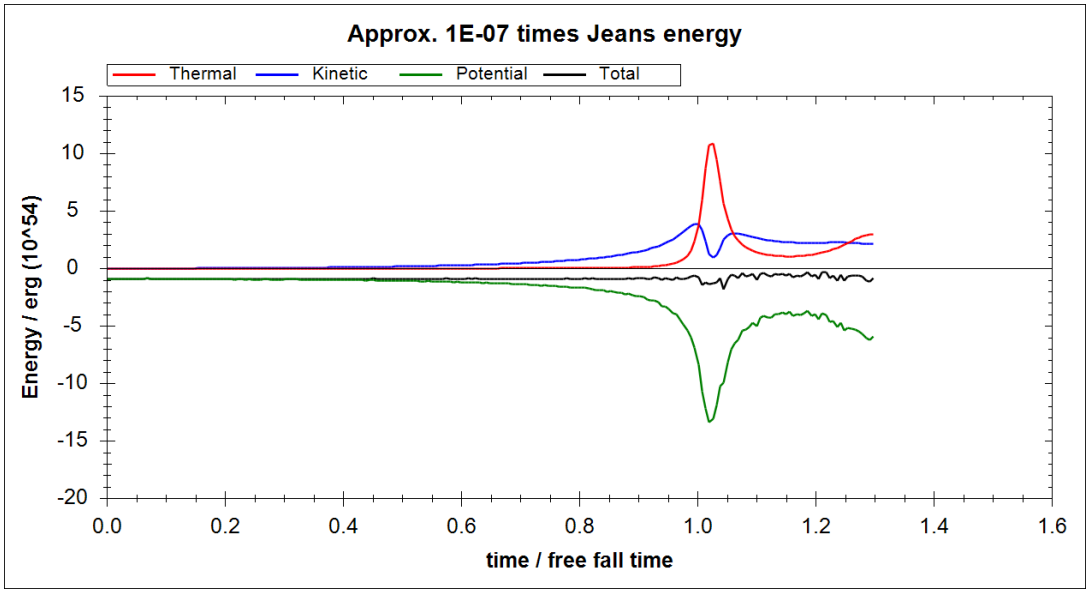


Figure 2.13: Energy measurements over time of an adiabatic spherical collapse. The simulated object was a 40 parsec gas sphere in vacuum surroundings at a density of 1 particle per cubic parsec weighing $1000 M_\odot$ each, set up using the glass-cut generator described in Section 2.3 and a very small initial temperature of $10^{-7} \cdot E_{\text{Jeans}}$. The noise in the gravitational and total energy is dominated by insufficient sampling of distances to determine gravitational energy for the plot; precision is also dependent on the CFL criterion parameter passed to Espresso. Both sources of error can be reduced further at the expense of performance.

itational potential energy.

$$E_{\text{pot}} \approx \frac{1}{2}\left(\frac{n}{n_{\text{sample}}}\right)^2 \sum_{i \neq j} -\frac{G m_i m_j}{|\vec{x}_i - \vec{x}_j|},$$

where $i, j \in$ sample. This is rather imprecise for small samples and slow otherwise, but has the advantage of not using the same representation as Espresso, which would somewhat defeat the purpose of testing its validity.

Figure 2.13 shows the energy measurements over time in the simulation. Due to the adiabatic equation of state, the collapsing gas heats up and stops the contraction. The simulation setup is again a 40 parsec radius sphere, though set up using the glass-cut placement algorithm. (See the figure description for more simulation details.) The glass generator was preferred over a grid because the distinct grid axes can cause large-scale anisotropic behavior. Glass is agnostic to direction and therefore considered to better represent problems such as this one.

The outcome of this simulation was not strongly dependent on particle count (multiple similar plots to Figure 2.13 not shown), but sensitive to time-resolution, which is determined by the Courant number: the maximum factor between time-steps and the Courant-Friedrichs-Lewy (CFL) [3] stability limit. With increases in the average time-steps for the hydrodynamical simulation, energy conservation was increasingly violated. Espresso's implementation and default settings were undergoing changes at the time of this writing, including algorithmic changes that may affect the influence of time resolution on energy conservation; please refer to Martin Zintl [34] for a current specification of simulator settings.

As a simple additional test, the same sphere was set up again with a homogeneous temperature that places it exactly at its Jeans mass. Figure 2.14 shows the outcome: the cloud only slightly rearranged itself to form a stable density profile, but otherwise showed very little change in the energy distribution. As in the previous run, time is scaled to the free-fall time that would be expected in the absence of pressure support, as expected for this setup.

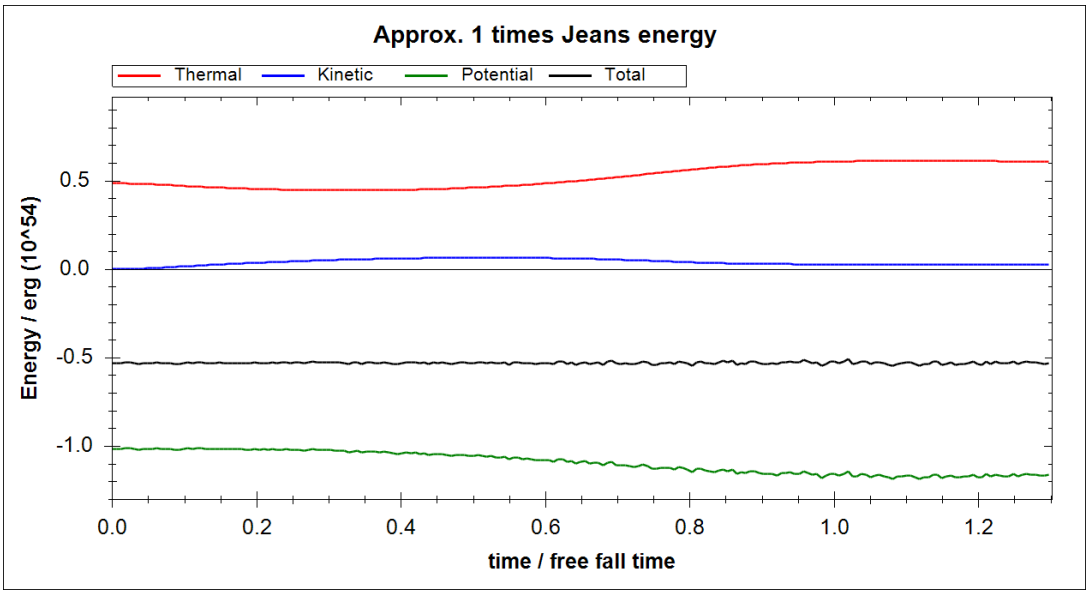


Figure 2.14: Energy measurements over time of an adiabatic simulation set up similar to the one presented in Figure 2.13, but with the initial gas temperature calculated such that the structure is exactly at Jeans mass. The run shows only very little change in the distribution of energy between the potential, kinetic and thermal fraction.

3 Radial simulation of isothermal self-gravitating filaments

Chapter 2 dealt with code employed around SPH simulations – but not the simulation itself. This chapter will attempt to make up for this, though the code used for the most part is not an SPH code, but a grid code, which is much better suited to the problem at hand. To examine the radial profile of isothermal filaments and their development over time, a one-dimensional simulation in a cylindrical space topology is implemented.

Analytically known stationary density profiles of infinitely extended cylinders are introduced and extended to pressure-supported and finite-radius cases. Simulations are set up to attempt their reproduction and study their dynamics and stability.

Star-forming filaments are elongated structures that to some extent resemble cylinders. Before we get to the simulation details, let us begin with some observational context and theoretical background on these.

3.1 Star-forming filaments in observations

Star-forming molecular clouds feature prominent filamentary structures quite different from the diffuse host cloud surrounding them. Dense cores observed in their vicinity are an important source of low-mass stars. Understanding the processes that connect these two structures – and their environment – is integral to the understanding of star-formation.

Hacar and Tafalla 2011 [13] studied filaments of the L1517 molecular cloud in detail and their work provides the main motivation for the content of this chapter. The filaments consist of approximately isothermal molecular gas at temperatures of about 10K. Multiple filaments can form long chains, such as seen in Figure 1.1 of the introduction; individual filaments are thread-like shapes typically 0.5 parsec in length. In contradiction to theories that proposed turbulence as a dominant factor in the stabilization and behavior of filaments, these observations found them to be very quiescent objects of sub-sonic inner motion and a strongly coherent velocity profile.

This new image of filaments makes them very distinct from the molecular cloud's components they are embedded in, which is turbulent and dominated by supersonic motion (see again [13] for many details and sources). The observed filaments appear more similar to the dense cores that form individual stars, which are supported by internal thermal pressure [12], than to their environment. The surrounding gas can then act as pressure support for the filament.

An isothermal cylinder model can be applied to approximate the filaments' behavior. Theoretical analysis of radial stability in infinite isothermal cylinders yields profiles known as Ostriker distributions (Discussed in Section 3.2 in detail). Of four density profiles of filaments studied by Hacar and Tafalla [13], three displayed a remarkable resemblance to the corresponding Ostriker distribution function. The mass per length found for the filaments was also reasonably close to a critical value relevant in the same model.

These findings may imply a model on star formation in molecular clouds:

1. Filament formation occurs. There are multiple suggested mechanisms for this, for example perturbations caused by uniform magnetic fields, as proposed by Nagai et al. 1998 [21].

2. The individual filaments relax into a quiescent state maintained by self-gravitation and pressure support from the environment, while supported against collapse by their own isothermal gas pressure. Radial density profiles at this time may resemble the cylinders of Ostriker's model.

3. The filaments fragment into dense cores that surpass their Jeans mass and collapse, until they become opaque and leave the isothermal regime, to ultimately form stars. See e.g. Schmalzl et al. 2010 [29].

Simulations of filaments face multiple problems. For one, including a turbulence-dominated environment is not easy. Another issue is the filament's immediate instability when simulated in isolation. From perturbation analysis, fragmentation along the axis with a wavelength on the order of four effective radii is expected. However, a naïve implementation to simulate a typical filament will not live to see this. The structure's gravitational pull quickly collapses the outer ends, pulling them into the center and forming more of a blob than a thread as seen in reality. Multiple points may contribute to this discrepancy; magnetic fields and the nature of the pressure support may play a role and, importantly, the presence of other dense filaments in a chain as seen in L1517 provides gravitational pull on the filament's axis.

We focus on the radial profile and formulate the following questions to the proposed theory:

- How does the Ostriker model, which is infinitely extended in both the radial and axial direction, relate to the finite physical filaments observed in the ISM?
- The Ostriker model defines a specific mass per length for which it is valid. What is its meaning in a real-world context that does not match an exact mass value?
- Can we numerically support a model that answers these questions and show its stability, in agreement with observations?

To answer these questions, we first discuss the analytical implications of Ostriker profiles. We then proceed to create a simulation that mimics the radial dynamics of isothermal, cylindrically symmetric distributions – first attempting to use a restricted version of Espresso, then implementing a custom grid code.

3.2 The Ostriker density profile

J. Ostriker 1964 [24] derived the analytically stable – actually metastable – solution to the radial density profile of infinitely extended self-gravitating cylinders. His proofs yield a family of stable distribution functions for the isothermal case:

$$\rho(r) = \rho_0 \frac{1}{\left(1 + \frac{1}{8}\xi^2\right)^2}, \quad \text{where} \quad \xi = \sqrt{\frac{4\pi G \rho_0}{c_s^2}} r$$

An important property of this result is that ρ_0 is a free parameter. Theoretically, a solution exists for any central density. This is also why this infinite-radius case is not stable in the sense of self-stabilization; it could shift between energetically equivalent solutions.

This density distribution can be reformulated a bit more conveniently by defining $H = r\sqrt{8}/\xi$:

$$\rho(r) = \frac{\rho_0}{\left(1 + (r/H)^2\right)^2}; \qquad H = \sqrt{\frac{2c_s^2}{\pi G \rho_0}}$$

This formulation has the advantage that H is easy to relate to as a characteristic scale height: the mass per length of a homogeneous cylinder with density ρ_0 and radius H equals that of its Ostriker equivalent. Now, we integrate the distribution over radius:

$$\begin{aligned}\int_0^R \int_0^{2\pi} \rho(r) \mathrm{d}\varphi \mathrm{d}r = \int_0^R 2\pi r \rho(r) \mathrm{d}r = \\ = 2\pi\rho_0 \int_0^R \frac{r}{\left(1 + (r/H)^2\right)^2} \mathrm{d}r = \pi\rho_0 \frac{H^2 R^2}{H^2 + R^2}\end{aligned}$$

This integration for finite radii will be useful in a bit, but for now, let us use it to determine the total mass per length in the infinitely extended case:

$$\begin{aligned}\left(\frac{M}{L}\right)_{\text{crit}} &= \int_{r=0}^{\infty} \int_{\varphi=0}^{2\pi} \rho(r) r \quad \mathrm{d}\varphi \mathrm{d}r \\ &= \rho_0 H^2 \pi = \frac{2c_s^2}{G}\end{aligned}$$

This shows that there exists an exactly defined mass per length for all infinite Ostriker distributions, the *critical* mass per length.

How to apply the Ostriker solution to physical filaments is not immediately clear. It is unlikely that an actual gas cloud has the correct mass per length; even if it were to match exactly, the free parameter ρ_0 would remain unaccounted for, as well as the pressurized environment of the molecular cloud that contains the filament.

Yet, these matters together can be used to fully define the Ostriker distribution for a pressure-supported case. Note first that ρ_0 is the *only* free

parameter of the general Ostriker distribution. H depends on it in an unambiguous manner and all remaining parameters have been assumed to be constants.

Therefore, for finite-radius filaments, we have four parameters which we want to constrain with respect to each other:

- The central density ρ_0.
- The outer radius after relaxation R.
- The outside pressure support P_{outer}.
- The mass per length $(M/L) < (M/L)_{\text{crit}}$.

(We expect that any setup at or above critical (M/L) that begins contracting contracts indefinitely, restricting ourselves to lower masses.)

For the purposes of obtaining quiescent filaments, we consider an external influx of mass not relevant, as we are interested in a stable solution. Thus we assume that mass is conserved. This makes (M/L) a fixed, constant property of the studied system. *Mass influx was not examined for this thesis. Should this become of interest, the conducted simulations could easily be modified to include it.*

Now, we have three remaining parameters to relate: ρ_0, R and $\rho(R)$. But we have already done this. The integration of the Ostriker distribution family to a finite radius must yield our fixed (M/L).

$$\frac{M}{L} = \pi\rho_0 \frac{H^2 R^2}{H^2 + R^2}$$

As we noted, H and ρ_0 are essentially the same parameter. So we have successfully related R with ρ_0 – and if either is given, the Ostriker distribution function $\rho(r)$ is fully determined. Since the filaments are isothermal and pressure should not jump at the outer edge for a stationary solution, the pressure support is then simply:

$$P_{\text{outer}} = \rho(R) c_s^2$$

An interesting interpretation of this is that we are *capping* infinite Ostriker filaments at finite radii. Gravitationally, there is no inward flow of information. This will be explained for the gravity details of Section 3.4. Hydrodynamical forces are local, so when in pressure equilibrium, the inner part of the filament *is not influenced* by its outer part; the forces acting inside the filament are identical between the infinite and pressure-supported cases.

With this, we have derived a well-defined stable solution for every $(M/L) < (M/L)_{\text{crit}}$ at a fixed filament radius. In addition, we have shown that given (M/L), specifying a distribution's radius is equivalent to providing the central density or the external pressure.

3.3 Espresso runs

A first series of simulations of isothermal filaments was conducted by running the Espresso simulator using tools from Chapter 2. An isothermal, cylindrical filament was set up in "noisy" random placement mode with a cylindrical distribution of particles of identical mass following an Ostriker profile. To check its properties without immediately collapsing it along its long axis, Espresso was modified to restrict movement along this axis. All velocities along it were reduced to almost zero, to still allow some relaxation on the scales of particle distances, while preventing any significant movement during the simulated time.

Analyzing processes in the radial direction of filaments this way proved difficult and ultimately led to the custom simulator that is the main topic of this chapter. The main issues were:

- A lack of a suitable 2D gravity setting. In a real two-dimensional simulation, the gravitational force of a circularly symmetric object scales with r^{-1}, not r^{-2} as for a spherically symmetric object in three dimensions. When the simulations were conducted, Espresso did not have operational support for this.
- In a similar vein, the simulator does not allow 1D simulations, nor would it allow applying a non-Euclidean space topology. Espresso is not intended for this type of problem, so a full 3D simulation had to be performed, costing performance.
- The placement algorithms were not intended for a three-dimensional simulation with restricted movement along one axis; they may create layers with more particles than others, creating spurious tension in the gas that may need further analysis to not impair results. Allowing minimal movement along the cylinder's axis removed these concerns, but in turn limited total simulation time as this caused an unphysical type of collapse at the outer ends when enough time passed for particles to move significantly.

- Periodic boundary conditions were not supported in Espresso's gravity implementation. In combination with the lack of 2D gravity this necessitates simulation of a long filament to correctly represent the situation, rendering results further to the ends of the filament inapplicable to the simple radial problem.
- The simulation boundaries were fixed to cuboids and streaming boundary conditions only supported for a constant flux. Thus, the simulation used hard boundary conditions of a non-cylindrical shape, which interacted either with the outer end of the filament or outer hot particles placed to simulate external pressure. Any waves from initial relaxation reflected on these boundaries, causing anisotropic waves.
- SPH surface tension created disturbances when particles intended to provide outside pressure interacted with filament particles. Setups partially mixing them initially resulted in each type locked between the others; hot particles did not escape from the cold filament as one might expect. *Espresso was undergoing changes to this at the time of these simulations, the situation regarding this may have changed now, depending on the settings Espresso runs with.*
- Gravitational softening became a major influence on results (this will be detailed next) and Espresso did not support adaptive softening at the time of this writing (which would change softening with respect to the local conditions, see e.g. Iannuzzi and Dolag 2011 [15]).

There were additional minor problems, such as random placement or minimal random relative motion between the locked "layers" displacing the filament's center, often differently for different sections, complicating automated analysis of the results.

Stability from softening

For the calculation of the gravitational force, simulated particles in Espresso are not represented as a point mass. They use a potential that is flattened out at the center, which is represented by splines in the default settings. This technique, which is known as gravitational softening, is common in SPH and used mainly for two reasons:

First, while approximating the gravitational potential of spherically symmetric bodies – like SPH particles – by point masses does yield the correct result at higher distances, it obviously does not when sampled inside that body.

Second, even more importantly, the implementation of movement and acceleration in Espresso, like in many particle-based simulators, relies on the assumption that time-steps are small with respect to any acceleration of their particles. Changes in velocity, such as the effect of the gravitational force, should not be major within a single time-step.

In typical implementations, gravitational softening reduces the overall effect of gravity. The effect of this reduction is not large in typical simulations, as gravity is equally important on all scales, but only the smallest experience a significant reduction and, as mentioned earlier, this may even be physically sensible in the case of overlapping particles.

Yet the effects of softening are not intuitive and can be outright surprising. One might expect that the scale at which the potential is flattened – the *softening parameter* – sets the scale for distortions to the physical reality and that in cases of high mass concentrations effects from softening get outweighed anywhere outside this scale. A 1996 paper by Sommer-Larsen et al. [30] shows that this is *not at all* the case. To quote one of their findings: *"We demonstrate the perhaps somewhat surprising result that even in the complete absence of rotational support it is possible, for any finite choice of softening length ϵ and temperature T, to deposit an arbitrarily large mass of gas in pressure equilibrium and with a non-singular density distribution inside of r_0 for any $r_0 > 0$."*

Simulations in Espresso aimed at reproducing the radial density profile of isothermal filaments and their relaxation suffered from the same type of problem. Ostriker filaments at rest were set up, only to relax into unexpected configurations. Figure 3.1 shows three variants of a simulation at 160% of the critical mass per length, differing only in their softening parameters. They yield distributions that are much larger than the largest softening parameter used.

A similar set of simulations experimented with varying initial velocities – and yielded differing final results! This should not have a physical explanation to my knowledge, as Ostriker filaments' radial stability should only *depend on their mass per length.*

Analysis of the effect and fine-tuning of the softening, or implementation of adaptive softening, may solve this issue. However, the combination of problems shown in this section made it preferable to put this type of analysis on hold and instead

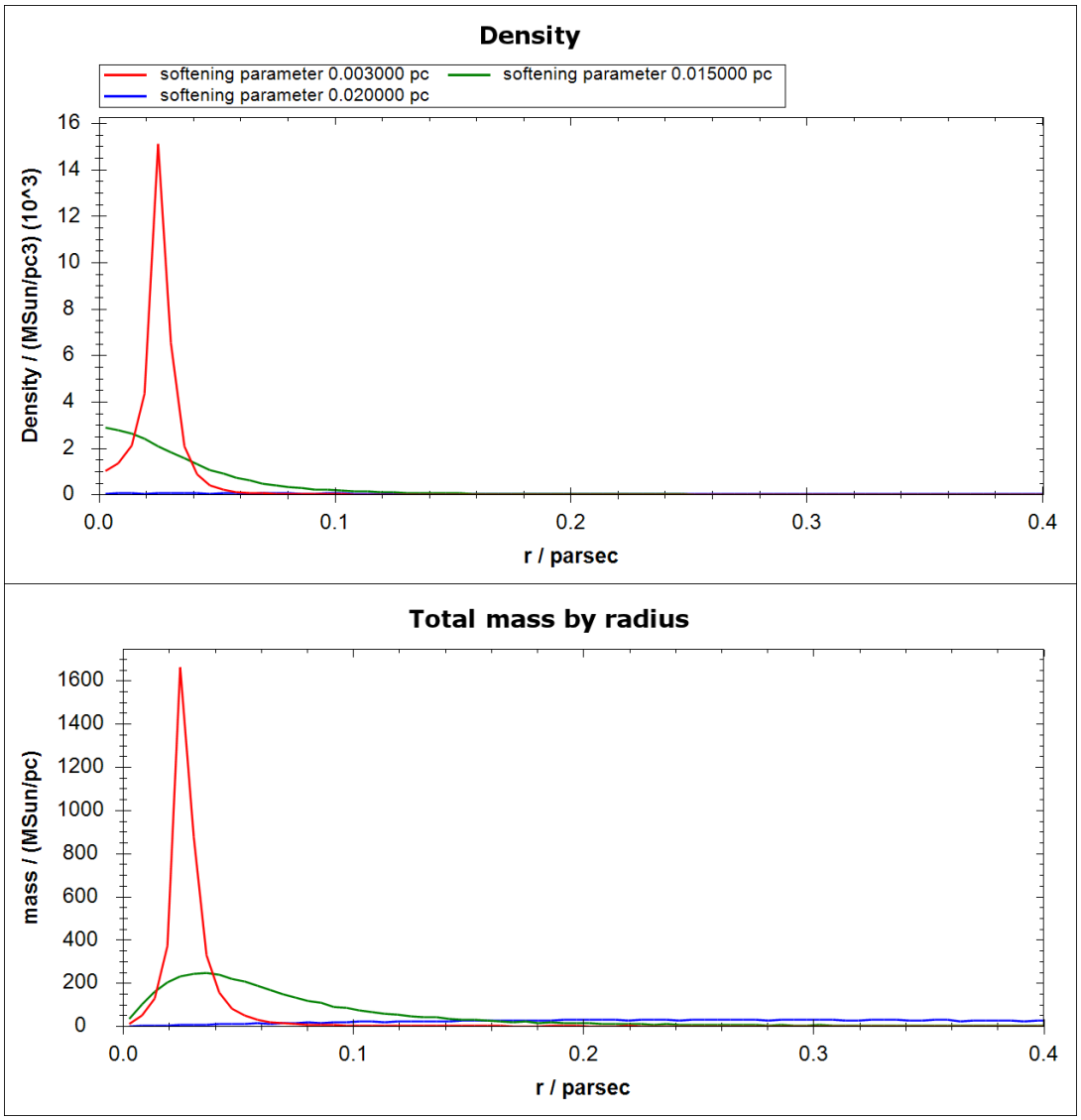


Figure 3.1: Radial profiles of theoretically unstable cylinders – above critical M/L – after relaxation in Espresso, patched to restrict movement along the cylinder axis, with varying softening parameter. The upper plot shows the radial density distribution as measured from the center of the initial setup. The lower plot shows $\partial m/\partial r$ from the same measurement. The gravitational softening plays a larger role than one would expect, creating shapes much larger than the softening scale. Impulse due to random noise allowed the most collapsed cylinder to drift from the center. The mass distribution of the high-softening run (blue line) is so expanded that its mass distribution plot is capped at the outer end, while it shows no notable peak in its density distribution.

concentrate on a kind of simulation specific to the case of interest.

3.4 Simulator implementation

With the problems associated with the use of three-dimensional SPH simulations for the radial analysis and the model not yet numerically reproduced, a specialized solution to the specific problem of relaxation in radial density profiles becomes an option. A one-dimensional approach to the problem has the advantage that it is inexpensive in terms of computations and allows to achieve much higher resolution. A CPU-based solution written directly in F#, without extensive optimization, suffices to create a simulation of good speed and resolution.

The algorithm used is a custom grid-code for isothermal hydrodynamics, with second-order accuracy in space and slope limitation for the density. It can be switched between a simple "pipe" topology and the radial direction of an infinite, perfectly symmetric cylinder. It could in principle be switched to other 1D topologies described by a function yielding cell interface widths along the simulated axis. The time-step used is constant in space but adaptive in time.

Grid

The simulation's data structure is designed for simple grid advection. Space is divided into cells of equal length along the simulated direction. Each cell holds the density and velocity of the contained gas; it is represented as a structure that holds two double-precision floating point numbers as its only data fields:

```
[<Struct>]
type Cell =
    val rho : float<g/cm^3>
    val u : float<cm/s>

    (...)
```

We fix CGS units as the numerical representation, which is unproblematic despite the somewhat large numbers this produces: double-precision floating-point values retain their full fraction's precision in an interval a little larger than $[10^{-307}, 10^{308}]$. Even a *cubic kiloparsec*, 1 kpc^3, only equals about $3 \cdot 10^{64}$cm^3.

(The `Struct` attribute is specific to the CIL type system; it indicates a *value type* that allows the type to be used without allocating a separate, reference-managed object. See F#/CLI references [7] [20] for details.)

The topology of the 1D-space to simulate is given by a record holding, among other fields, a *surface function* that maps positions along the simulated axis (e.g. radii) to cell's cross-sections, which are also the surfaces between the cells. This allows cells to vary in size.

In terms of the unit-of-measure system, a cell surface is a length and the content of a cell is a mass per length. In other words, the simulator always assumes that it is simulating a two-dimensional cut that is infinitely repeated along the third dimension. This is useful for the cylindrical topology and sufficiently flexible to test and debug in a simple "pipe" topology. The latter can be achieved by providing a constant surface function. For the cylindrical simulation, the surface function is a circle: $S/L = 2r\pi$, where S is a surface between cells, L is the length in the (not simulated) z direction, and r is the position along the simulated axis.

The ability to switch topology could in principle be used for other simulations, such as a pipe of varying width or a pressure wave from one point on a planet to the opposite side. It seems unlikely that this feature will be used, but there was little reason to restrict it, so it remains available.

Piecewise linear interpolation

The simulator only stores average densities of cells, yet does use a piecewise linear density function in calculations. It was created with the help of a lecture script about grid codes by Springel and Dullemond [31] which also introduces linear interpolation schemes and slope limiters. This subsection, as well as the following one on application of hydrodynamics, was written with repeated use of the lecture as a reference.

Inside a single cell, both its width – the amount of space per length – and density are a linear function of position. Thus, the topology gets reduced to a piecewise linear function too. (This makes no difference for cylinders.) The algorithm only samples cell size at their centers and borders. If one were to input a non-linear topology, it would only be approximated by the cells.

Interpolating the size of space is straightforward. The slope of the density distribution inside a cell, however, is not fully determined by the simulation's state. As previously mentioned, cells only store their gas velocity and mean density. Any information on the sub-cell distribution is lost between

time-steps. Therefore, the slope must be determined anew before each advection step.

For each cell, the algorithm estimates the slope toward its adjacent cells in both directions. It assumes that each cell has its mean density at the center. *Note that due to the varying cell shapes, this is only an approximation.* Next, it chooses the slope such that it interpolates between these central points:

$$\begin{aligned} \mathrm{s}(i, \mathrm{right}) &= \frac{\rho_{i+1} - \rho_i}{\Delta r} \\ \mathrm{s}(i, \mathrm{left}) &= \frac{\rho_i - \rho_{i-1}}{\Delta r} \end{aligned}$$

Here i is a cell index, s a slope on one side when doing simple linear interpolation, ρ the density and Δr the distance between two cell centers. Now, in a typical case, something like the average of these two slopes might be a sensible choice. The trouble is that they can be vastly different. What would seem like the proper interpolation on one side then would create an overshoot – or undershoot – of the interpolation at the cell interface on the other side.

The choice of a slope-limiting scheme determines how steep slopes can become in questionable cases. For this one-dimensional simulator, spatial precision was already high without interpolation, so avoiding numerical artifacts was the priority. Hence a conservative option is used: picking the less steep slope if their signs match and zero – no interpolation at all – otherwise. (Which happens at peaks and bottoms of the distribution.) This method is known as the minmod slope limiter, named after the minmod function it uses:

$$\mathrm{minmod}(a, b) = \begin{cases} 0 & \text{if } ab \leq 0 \\ a & \text{if } ab > 0 \wedge |a| \leq |b| \\ b & \text{if } ab > 0 \wedge |a| \geq |b| \end{cases}$$

This formulation emphasizes symmetry but may look strange. With $ab > 0$ and $|a| = |b|$ both the second and third cases are true. This is consistent; it implies that $a = b$. Therefore, both the second and third cases are indeed true.

Using both slope estimates and the minmod function, a cell's final density slope is:

$$\begin{aligned} \mathrm{slope}(i) &= \mathrm{minmod}\left(\mathrm{s}(i, \mathrm{left}), \mathrm{s}(i, \mathrm{right})\right) \\ &= \mathrm{minmod}\left(\frac{\rho_i - \rho_{i-1}}{\Delta r}, \frac{\rho_{i+1} - \rho_i}{\Delta r}\right). \end{aligned}$$

Hydrodynamics

The algorithm for fluid dynamics follows the methods detailed in Chapter 5 of the Springel-Dullemond lecture notes [31] and adapts them to the case at hand. The main modification is the inclusion of the interpolation scheme for cell size and density introduced in the previous subsection.

With simulations limited to the isothermal case, there is no need for an energy equation. We must adhere to Euler's continuity and momentum equations though:

$$\begin{aligned} \partial_t \rho + \nabla \cdot (\rho \vec{u}) &= 0 \\ \partial_t (\rho \vec{u}) + \nabla \cdot (\vec{u} \otimes \rho \vec{u}) + \nabla P + \underbrace{\rho \nabla \Phi}_{\text{gravity}} &= \vec{0} \end{aligned}$$

Here, $\vec{u}$ indicates gas velocity and P its pressure. We ignore the gravitational term; a Gauß solver detailed in a later part of this section will handle it. Since we only use one dimension of movement, these equations become:

$$\begin{aligned} \partial_t \rho + \partial_r (\rho u) &= 0 \\ \partial_t (\rho u) + \partial_r (\rho u^2) + \partial_r P &= 0 \end{aligned}$$

Since the actual simulations are in a cylinder's radial axis, this axis is denoted with r. With the exception of the pressure term in the momentum equation, these equations can be applied by using an *advection algorithm*: a program that lets mass and momentum flow between cells, in accordance with the local gas velocity. The pressure term can then be added by approximating the pressure gradient for each cell and applying the corresponding acceleration. We begin by dividing the algorithm into two steps.

$$\begin{pmatrix} \rho_{\mathrm{new}} \\ (\rho u)_{\mathrm{new}} \end{pmatrix} = \mathrm{pressurize}\left(\mathrm{advect}\begin{pmatrix} \rho \\ \rho u \end{pmatrix}\right)$$

The advection equations follow from the previous 1D Euler equations when the pressure term is neglected. For a time-step of Δt, we obtain new values for ρ and ρu:

$$\begin{aligned} \rho_{\mathrm{new}} &= \rho - \Delta t \partial_r (u\rho) \\ (\rho u)_{\mathrm{mid}} &= \rho u - \Delta t \partial_r (u \rho u) \end{aligned}$$

The subscript "mid" signifies an intermediate result, "new" the resulting values after the time-step was applied. The final density of one time-step follows directly from advection, while the resulting

impulse – and thus velocity – is still missing the pressure term.

The two terms of the form $\Delta t \partial_r (u \cdot \text{quantity})$ are nothing more than the integrated fluxes of the quantities to adjacent cells. The algorithm acquires these by calculating what fraction of one cell's mass moves into the other in Δt at the local velocity. It subtracts the resulting amount of mass and impulse from the cell with outgoing movement and adds it to the other. The velocity at the boundary is estimated by averaging over the gas velocities of both cells. These calculations heed the linear interpolation schemes for cell size and density.

Note that this is an *upwind scheme* that limits the flow of gas strictly to the direction of the current velocity at the cell boundary. No information flows opposite to the velocity's direction in the advection step.

Time-steps throughout the simulation were limited such that in any of them, advection does not move gas further than 0.39 cell-lengths. Such a factor connecting cell size to the gas movement during one step of time-integration is known as a Courant number, named after the Courant-Friedichs-Lewy (CFL) condition introduced in the thee authors' 1928 paper [3], which states the somewhat sensible rule that algorithms with a one-cell interaction range should not execute movements in a single step that exceed one cell's size.

We are now left with only with the pressure term, which is added in a second step:

$$(\rho u)_{\text{new}} = (\rho u)_{\text{mid}} - \Delta t \partial_r P$$

In the isothermal case, pressure is related to the speed of sound by $P = c_s^2 \rho$. This allows us to approximate $\partial_r P$:

$$\begin{aligned} \partial_r P &= \partial_r c_s^2 \rho \\ &= c_s^2 \frac{\rho_{\text{right}} - \rho_{\text{left}}}{2 \cdot \Delta r_{\text{cell}}} \end{aligned}$$

With this, we can calculate the acceleration. We have now obtained both ρ_{new} and $(\rho u)_{\text{new}}$. Dividing the two yields the new velocity and concludes the time-step.

Gravity

The simulator has a gravity feature, which can be enabled with a boolean called `EnableGravity` in the topology record. It is implemented using the Gaussian formulation of gravity:

$$\nabla \vec{g} = -4\pi G \rho$$

This means that density is the source term for the gravitational field. The law can also be written in integral form:

$$\oiint \vec{g} \mathrm{d}\vec{A} = -4\pi G M$$

where M is the total enclosed mass. This means that the total flux through any closed surface scales linearly with the mass contained within the surface.

This formulation makes it evident that the total gravitational flux through a closed surface only depends on the mass inside the surface, not on its distribution, nor on any mass outside of it. This is exploited by the algorithm. It is assumed that the first simulated cell is the innermost area and that gravitational field lines can only "escape" through each cell's outer surface. The algorithm iterates outward from that innermost cell, applying the gravitational acceleration on each cell. A rough description of the function applied to each cell would be:

- Take the total mass of all cells further inside as a parameter.
- Calculate the size of a closed surface at the center of the cell, e.g. a ring around the center for the cylinder (this calls the space topology function).
- Calculate the total mass up to the center of the cell (the mass in the inner half of the cell plus the mass of all cells further inside).
- Use the obtained surface and contained mass to calculate the gravitational flux at the cell's center. Apply the corresponding acceleration to the cell.
- Recursively call this function on the next cell, adding this cell's mass to the running total.

Note that this method is $\mathcal{O}(n)$ in the number of cells. *However, as in pure hydrodynamics, time-steps effectively scale with* $(1/n)$*, limiting the computational order of simulating a fixed time to* $\mathcal{O}(n^2)$.

Pressurized Surroundings

Star-forming filaments are usually embedded in a molecular cloud, a comparably hotter, more turbulent environment that provides pressure support. To allow simulation of such situations, a constant external pressure can be provided as an optional value of the topology record:

```
/// External pressure at the higher end
/// of the simulation
ExtPressure : float<g/(cm * s^2)> option
```

If provided, this changes the behavior of the simulator. The amount of simulated cells becomes variable, allowing cells at the outer end of the simulation to be disabled. If the external pressure suffices to push the contents of the outermost simulated cell into the second-outermost, the cells are collapsed and the then-empty outermost cell gets disabled. Conversely, if the outermost active cell is filled at a density sufficient to surmount the external pressure and is currently moving outward, normal advection to the next cell in the outward direction is enabled, adding the cell to the active simulation.

The current implementation does not re-allocate cells, so an absolute maximum size of simulation space must be provided, above which the number of active cells cannot grow. This could be trivially changed if ever needed – it might not be useful though, since growth of the simulation beyond expected sizes could indicate a problem and indefinite expansion of simulated space might cause extreme slowdown of the simulator, delaying termination of a faulty run.

3.5 Artificial dissipation

The simulator implementation presented so far only dissipates waves as a numerical artifact. It does not conserve energy – by definition, as it uses an isothermal equation of state. Still, this formulation is not inherently dissipative.

For a very simple example that disregards numerical dissipation, take a system of only two adjacent cells at different densities, contained in reflective boundary conditions to the outside, initially at rest. The difference in pressure will cause gas to start flowing from the more densely filled cell to the other. But when they close in on equal density, the gas velocity has been significantly accelerated. To slow down the flow again, the inverse process of the acceleration has to occur, leaving the initially more thinly filled cell with the excess gas – and the cycle repeats. This would be a typical example of a harmonic oscillator. If left to run for a very long time, it may dissipate due to time-step discretization errors, but the time this takes would depend strongly on numerical simulator settings, which is not desired to play a major role in a simulation's outcome.

The analog effect for multiple cells enables them to reflect density waves off the simulation boundaries. The limited resolution of the cells smears out such waves as they propagate, but left to its own devices, the simulator would tend to show numerically generated oscillations long before relaxing into a steady state.

We are primarily interested in the relaxed distribution, so a way to dissipate movement from the system is needed. There is no requirement for a physically exact relaxation process, so it is viable to simply pick a time-scale and modify the algorithm accordingly, such that it removes impulse from the system. This can be expressed by a parameter Ω indicating velocity dissipation over time:

$$v(t + \Delta t) = e^{-\Omega \Delta t}\, v(t) \approx (1 - \Omega \Delta t)\, v(t)$$

Wherein the approximation is simply the first-order term of the Taylor series. Applying the last expression reduces the overall velocities – provided that Ω is chosen such that $\Omega \Delta t < 1$, which it should be. If not, the algorithm falls back to setting all velocities to zero.

Oscillation suppression

The methods described to this point are not entirely sufficient to ascertain that simulations reliably relax into a steady state. We have not proven our simulation stable in combination with both the asymmetric, varying cell topology and the impulse dissipation that was just introduced. By using algorithms based on physical laws, we have provided reasonable interactions on short lengths and timescales, but remain vulnerable to systematic errors of the representation. A simple example of such an effect is *odd-even decoupling*.

A cell's change in impulse due to pressure is calculated using values from exactly the two neighboring cells. This has the silly consequence that in the absence of other forces, an equilibrium distribution can have completely different density profiles for odd and even cells! Because of the diffusive nature of the grid on a moving fluid, this is usually not a problem. But in general, it may produce artifacts.

Using only the previously introduced features can create such resolution-dependent artifacts. They are most problematic at the center of simulated space, where the geometry of cells and the difference to their neighbors is most prominent. For the cylindrical simulation, this can cause wave-generation at the innermost end of the grid. A multi-cell variant of odd-even decoupling creates a resolution-dependent preferred frequency of waves,

which can interact with the velocity dissipation algorithm in such a way that it drives the waves continuously. It may sound strange that an exponential dampening of velocity does not stop this process, but the interaction between a simple pressure approximation, advection between differently-sized cells (note that the second-innermost cell is three times the size of its inner neighbor), and odd-even decoupling no longer reflects physical nature reasonably. Indeed, the process was not self-strengthening until a steep density slope close to the center triggered it.

The cause of such unphysical behavior lies in the algorithm's inability to correct density oscillations close to its resolution limit without larger-scale movement. Therefore, a correction to the dissipation algorithm was included; it specifies a slow diffusion velocity, typically of 1% of the sound speed, at which material diffuses away from local peaks. The scale of its effect is small compared to the overall movement in the simulation, so it does not reduce precision significantly.

3.6 Fixed-border Ostriker simulations

Let us begin by setting up the first run with reflective boundary conditions at the outer end of the simulation, so that we effectively simulate a hard-walled cylinder. This allows to set a fixed radius for the distribution and observe its relaxation, final shape, and stability.

As we know from Section 3.2, obtaining Ostriker solutions for comparison requires an extra step when dealing with "capped" distributions that do not extend infinitely as in the original theory. We equate the finite integral over the general Ostriker distribution with the fixed (M/L) in the simulation setup:

$$\left(\frac{M}{L}\right)_{\text{setup}} = \pi\rho_0 \frac{H^2R^2}{H^2+R^2} \overset{\dagger}{=} \left(\frac{M}{L}\right)_{\text{crit}} \frac{R^2}{\frac{2c_s^2}{\pi G\rho_0}+R^2}$$

At †, we used:

$$H^2 = \frac{2c_s^2}{\pi G\rho_0}; \qquad \left(\frac{M}{L}\right)_{\text{crit}} = \frac{2c_s^2}{G}$$

We can rearrange to obtain:

$$\rho_0 = \frac{(M/L)_{\text{crit}}}{R^2\pi\left(\frac{(M/L)_{\text{crit}}}{(M/L)_{\text{setup}}}-1\right)}$$

This expression for ρ_0 is used in the Ostriker profile function to obtain the expected distributions, represented by orange lines in upcoming plots.

The first simulation's space features $2,000$ cells; they are limited by a reflective wall at 0.6 parsec radius. 25% of the critical mass per unit length is chosen to fill the volume, distributed homogeneously. To this end, we divide $0.25(M/L)_{\text{crit}}$ by the cylinder cross section $r^2\pi$ and initialize all cells with the resulting density. The initial velocity is set to zero and the constant speed of sound to $0.2\frac{\text{km}}{\text{s}}$.

With this, we run the simulation. Figure 3.2 shows how it progresses: it begins with the homogeneous distribution contracting to the center. Then, it reaches a central density at which isothermal pressure provides support against further collapse. Compressed by the remaining inward velocity, The gas is accelerated outward again. The process repeats, but with reduced strength due to the simulator's dissipation. In this manner, the distribution oscillates around the stable Ostriker profile and ultimately settles down to exactly the predicted curve.

The detail simulations of this section are available as videos. Feel free to contact me about them.

The central density over time in this simulation can be seen in Figure 3.3. It features a period of approximately $5 \cdot 10^6$ years. This value should be dominated by the physical dynamics of the system, though the artificial dissipation is expected to have some impact on it, as it systematically slows down all movement. How representative the value is for physical filaments is difficult to say, as this is still a very theoretical setup. Though we are running on physical units, the initial conditions were chosen to demonstrate the method's validity rather than match any real filament. Also, the hard outer limits, serving for consistency testing with respect to Ostriker filaments, create unnaturally strong resistance to filament expansion. Section 3.7 will feature a simulation that removes these.

One may now ask whether fixed-radius simulations of filaments reliably relax to their respective Ostriker solutions for other radii. Since we have the entire simulation and analysis process available from our program, it is easy to automate the execution of multiple simulations. We pick the interval $[0.5\text{pc}, 1.5\text{pc}]$ for the radii we are interested in and run the simulation 130 times.

To assess whether a simulation returned the expected result, The central density ρ_0 after relaxation is a simple indicator, for which the analytical formula for comparison was provided in the previous simulation. The remaining settings are chosen

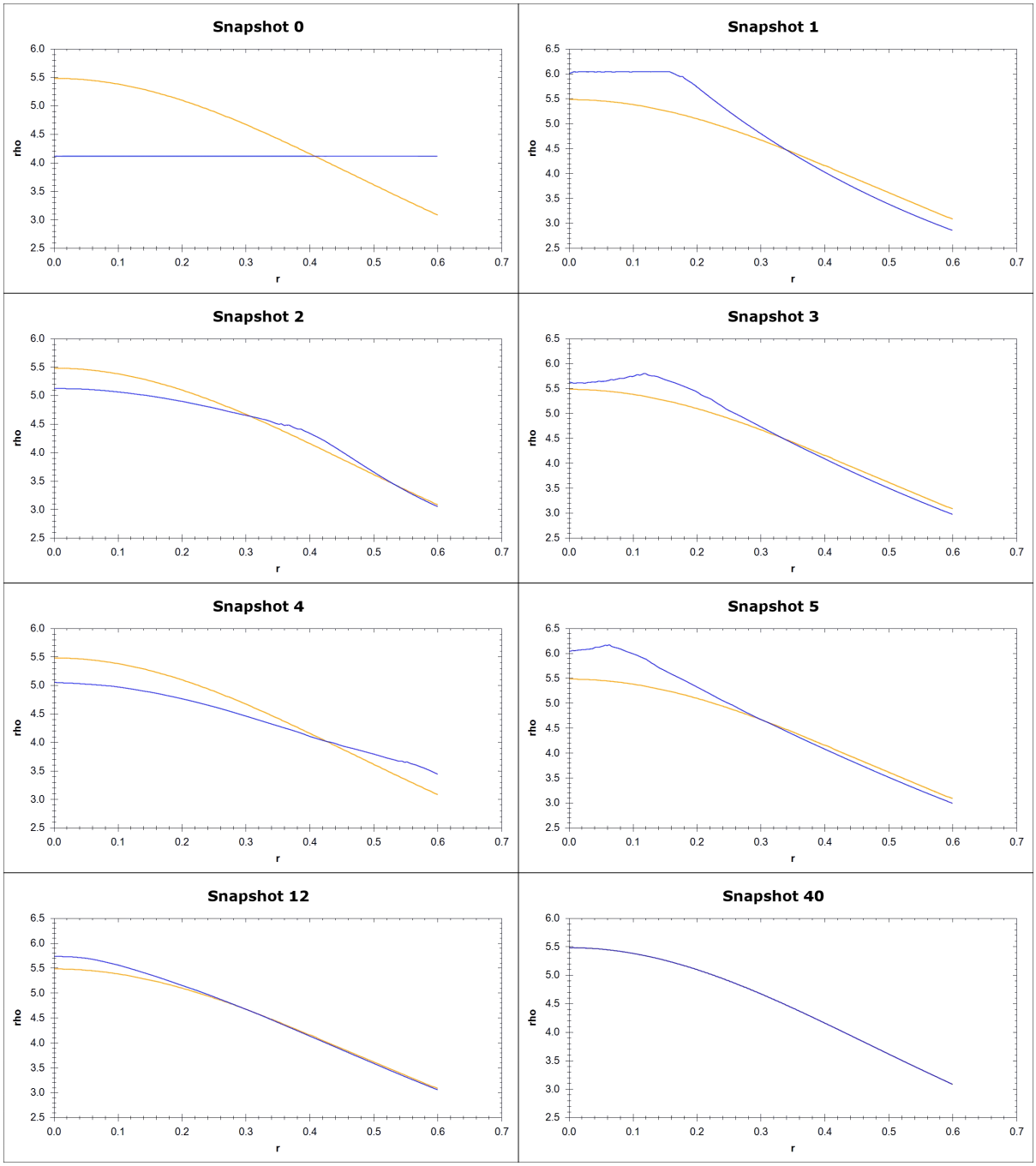


Figure 3.2: Shows the density distribution at different times during a simulation with hard boundaries. The blue line shows the simulator cells' densities, while the orange line shows the analytically stable solution for the given parameters as derived by Ostriker. The last two snapshots are taken at significantly later times to illustrate the final and perfect relaxation to equilibrium. The simulation was conducted with physical units, i.e. the radii of the images are in parsec and the densities in $M_\odot/\mathrm{pc}^3$. Note that the scale changes slightly between some snapshots.

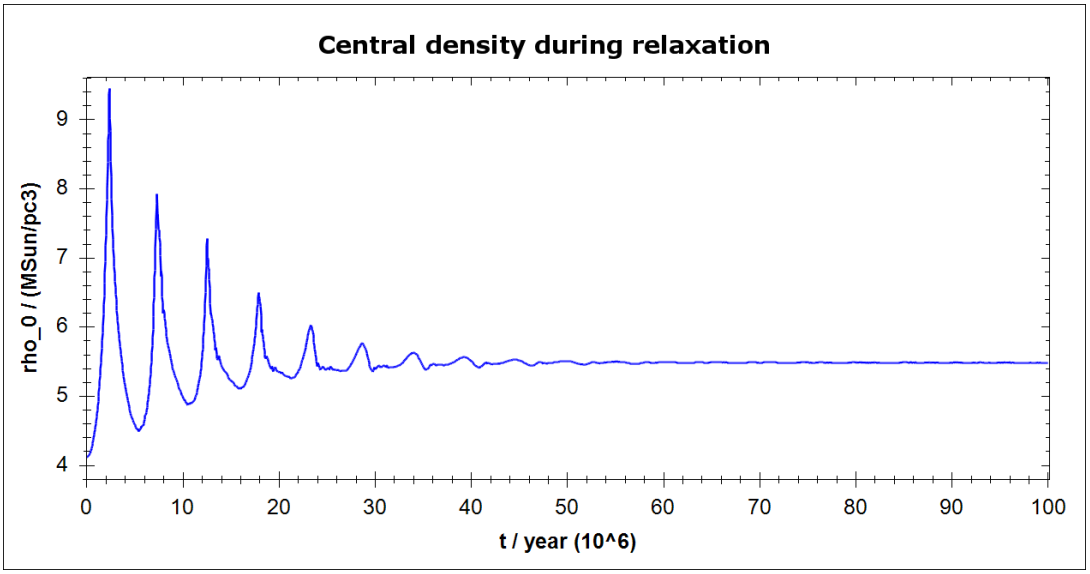


Figure 3.3: Central density ρ_0 over time of an initially homogeneous distribution weighing 25% of the critical mass per length, confined to a hard-walled cylinder of 0.6 parsec radius. The simulation was run on 2,000 cells at an isothermal sound speed of $0.2\frac{\text{km}}{\text{s}}$. The dampening time-scale is caused by arbitrary algorithmic dissipation (see Section 3.5), but this should not falsify the measured period of approximately $5 \cdot 10^6$ years strongly.

similar to the previous simulation, including the mass per unit length of $(M/L) = \frac{1}{4}(M/L)_{\text{crit}}$, so the returned value at an outer radius of 0.6 parsec represents the simulation we have seen in detail.

In Figure 3.4, the central densities of the simulation series is plotted over the outer radius of simulated space. Evidently, the relaxed result is stable and consistent with the analytical prediction over multiple runs and different radii.

Setting up simulations exactly at expected Ostriker profiles kept them stationary in very good approximation. Changing the densities of a valid solution by only $\pm 0.1\%$ caused visible contraction or expansion when compared to the initial distribution, indicating that the simulation is very precise and sensitive to minor changes.

Simulations close to critical mass per length

Multiple additional simulations were conducted using different sub-critical values for M/L; their data is not shown because they provided no qualitatively new results. Simulations with a low amount of mass featured shallow density profiles, while those using high amounts of mass revealed a larger portion of the infinitely extended Ostriker distributions, as theory predicts.

What did cause different behavior was approaching the critical mass per length, especially for simulations beginning with a homogeneous profile or other distributions involving high outer densities. The initial contraction causes a spike in central density, which was already prominent in Figure 3.3. This phase becomes increasingly difficult for the simulator to handle as the contained mass increases. Simulations conducted significantly above $(M/L)_{\text{crit}}$ consistently collapse and must be stopped due to excessive densities and accelerations. Choosing exactly $(M/L)_{\text{crit}}$ and involving any initial contraction (or relevant outer limit that causes contraction) also quickly results in an indefinite collapse that terminates the simulation. An example of this process be seen in Figure 3.5.

This result is consistent with expectations: an Ostriker filament at critical mass requires an infinitely extended distribution to remain stable. Even provided this, or a sufficiently large simulated space to approximate it, the system lacks a mechanism to stop collapses once they have begun.

Possibly, the variable simulation limit introduced in Section 3.4 and used in the upcoming Section 3.7

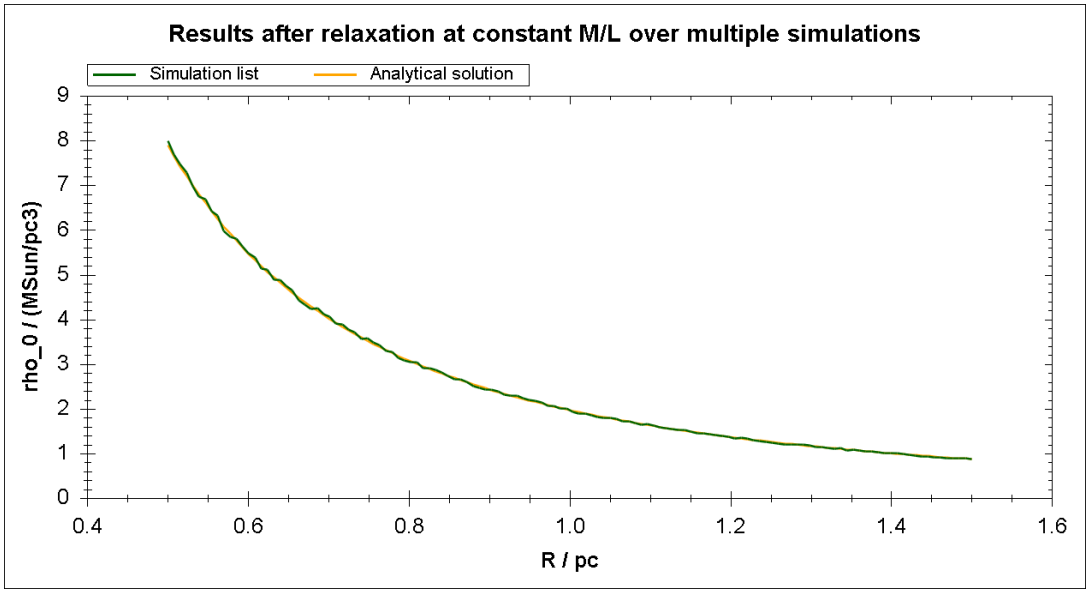


Figure 3.4: Density at the inner end of the distribution after relaxation over 130 simulations with varying outer boundary and mass per length set to 25% of the critical value.

could be utilized with a very low external pressure value to produce a gravitationally dominated full collapse to a singular distribution, but this would be meaningless, as the assumptions made for the simulation – such as the non-opacity of gas that justifies isothermal calculations – would no longer hold.

Setups slightly *below* the critical mass per length behaved similar at first glance. This turns out to be a numerical rather than physical property of the simulation. Simply retaining earlier settings while increasing M/L to over 99% of the critical value results in the termination of the simulation, very similar to what we saw in Figure 3.5. However, closer inspection reveals that the extreme setup does not cause an indefinite collapse the way the critical setup does. It only blurs the distinction between runaway processes that should be stopped and extreme conditions that are time-consuming to handle, causing the simulator to terminate prematurely as it hits limits such as a minimal time-step. As the restrictions that terminate simulations are loosened, it is possible to approach the critical value more closely, yielding increasingly extreme amplitudes of the oscillations in central density.

3.7 Variable-border Ostriker simulations

Real filaments are not contained in hard walls, which is why the option for pressurized surroundings of Section 3.4 was introduced. Figure 3.6 shows a relaxation process where, instead of setting radius directly, the constant outside pressure was configured to match the stable profile's pressure at a radius of 0.5 parsec. The initial mass distribution is again homogeneous; it is spread out over 0.7 parsec radius, but quickly contracts after the start of the simulation. The maximum simulation space spans one parsec and 1500 cells, but at no time in the simulation are all cells in use.

The simulation evolves similarly to its fixed-radius predecessors.

Approaching $(M/L)_{\text{crit}}$ in this type of setup would likely not give physically sensible results, as the external pressure accelerates the initial collapse until the distribution is very dense. The density peak after this initial contraction would create extreme conditions. This would clearly leave the valid domain of the algorithm, as a very strongly compressed filament may accrete additional gas and become critical, or become opaque and non-isothermal.

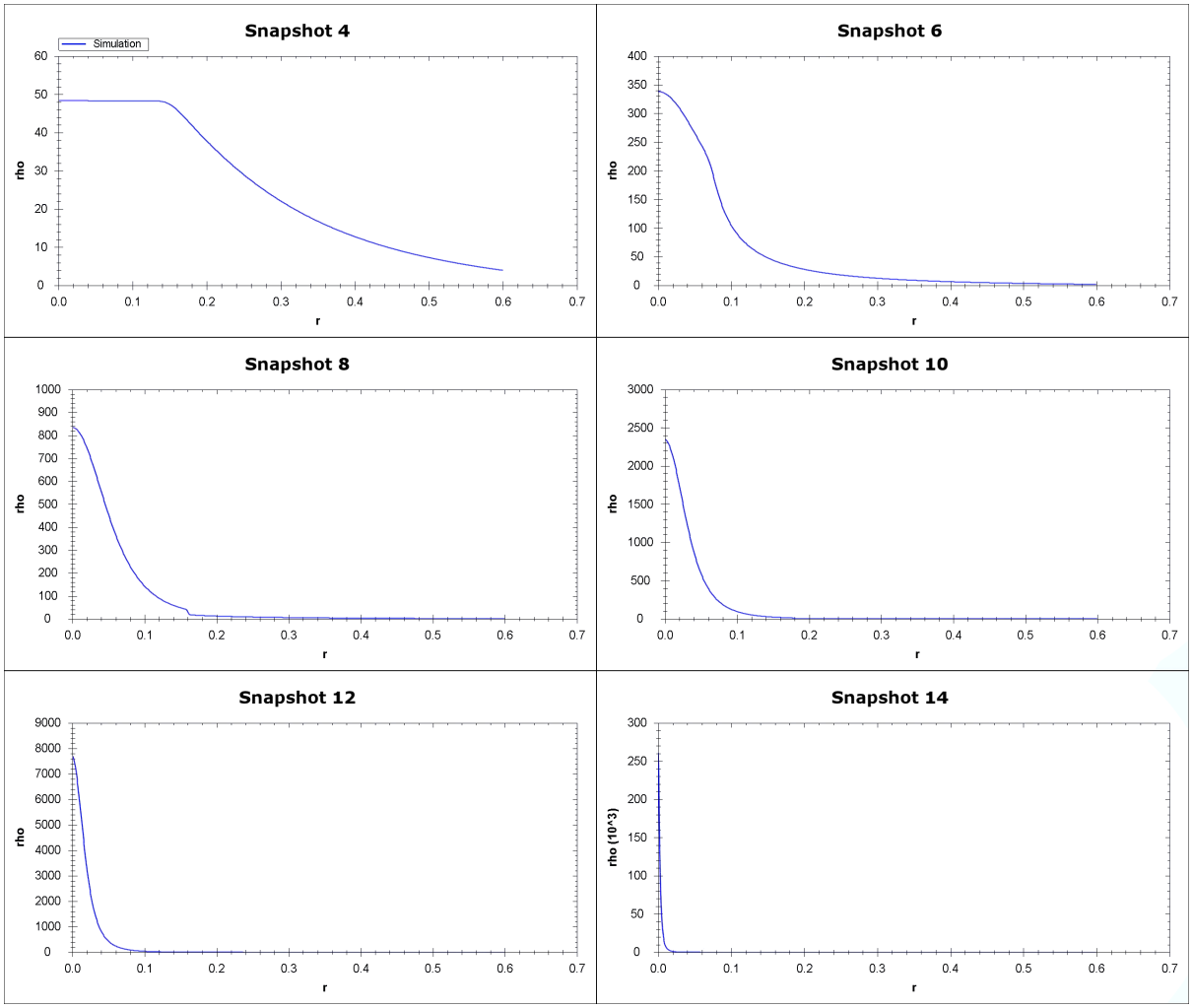


Figure 3.5: Simulation at critical (M/L) starting with a homogeneous initial distribution. To approximately mimic a stable Ostriker solution at this cylinder size – it is not possible to perfectly reach it with a finite radius – mass would have to be much more concentrated at the center. Thus, the distribution quickly begins to collapse. Given the full critical mass per length to maintain an arbitrarily high central density, the pressure support is insufficient to halt the distribution's collapse. Central density continues to rise until the simulator halts (last snapshot) as it is unable to handle the excessive outer velocities in fixed-radius simulation mode. Note the density scale change to $10^3 M_\odot/\mathrm{pc}^3$ in the last snapshot.

3.8 Conclusions

The pressure-supported variable-border simulation concludes the simulation series to reproduce stable Ostriker filaments of finite size. We have shown how the infinite Ostriker model can be utilized to relate finite physical filaments' radii, pressure support, and radial stability.

Results align with the theoretical solution that, under the assumption of an isotropic cylinder without movement along its axis, complete radial collapse is possible if and only if a critical mass per length of $(M/L)_{\mathrm{crit}} \geq 2c_s^2/G$ is present. Unlike infinitely extended isothermal cylinders, the finite variant can be stable below the critical mass per length. In such pressure-supported filaments, significant oscillations can occur without the filament becoming unstable.

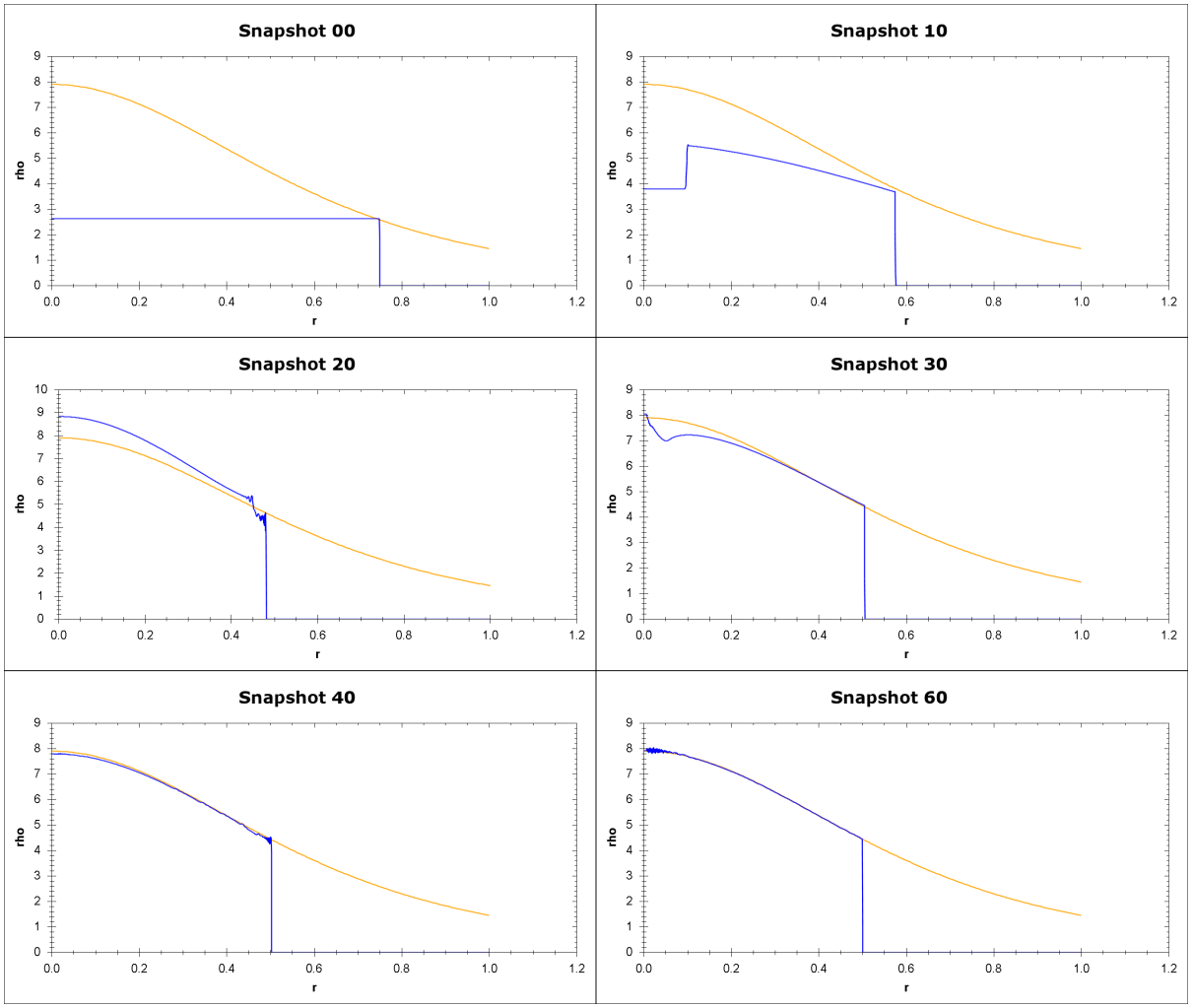


Figure 3.6: Plots of the density distribution at different time-steps of a radial simulation with external pressure, which was calculated to match the pressure of the corresponding Ostriker filament at 0.5 parsec. The blue line shows the simulated density distribution, the orange line is the analytically calculated Ostriker distribution for the given mass and external pressure. The maximal simulation space ranges up to 1.0 parsec and allocated 1,500 cells, of which any outside the distributions outer cut-off are disabled. Density is denoted in $M_{\odot}/\mathrm{pc}^3$. Note that snapshot 50 was skipped to make room for the almost fully relaxed distribution at Snapshot 60.

The results' insensitivity to the exact type of simulation and parameters used make it plausible that the model of stable Ostriker density profiles applies to filaments in pressurized environments, as suggested by Hacar et al. 2011 [13].

It may be of interest to conduct simulations of this type using real-world settings and observational data. Also, studying the lengthwise stability of filaments and their interaction with the environment, such as a possible influx of gas, might allow further interpretation of the stability limits of filament contraction, which ultimately define the border to star formation.

4 Assessment of F# in computational physics

This chapter can be seen as a distinct addition to the numerical and physical topics discussed in the previous two chapters. It focuses entirely on the programming concepts used in this thesis and the reasoning behind them.

As this project was largely independent from existing software, its programming language could be chosen freely. A selection of relevant choices can be seen in Figure 4.1, which lists some of their properties. The suitability of functional style and type safety, which will be discussed in the upcoming sections, made the "sharp" (#) languages of the listing preferred choices.

The project implementation began in C#, the CLI's most prominent language, which is similar to F# but has less functional emphasis. After evaluating some of the points described later in this chapter, new parts of the code were written in F# instead, linking to the existing C# code easily thanks to good interoperability between the two.

F# showed multiple advantages and ultimately, almost all the code was moved into the project's F# part. After further evaluation, multiple parts were re-implemented again in a more functional style. This chapter attempts to give an impression on what motivated this change.

4.1 Introducing F#

F# (pronounced "F Sharp") was designed and introduced at Microsoft Research, by a team led by Don Syme [8]. It follows in the footsteps of functional languages such as OCaml, to which it is similar enough to allow a "verbose syntax" mode that can compile a subset of OCaml programs as F# programs. Its main influence outside the functional realm is C#, the dominant language of the Common Language Infrastructure, with which it shares most of its type system.

Despite its name, in which the *F* reminds of the *functional* programming paradigm, F# is not a pure functional language. It is, on the contrary, a multi-paradigm language that combines object-oriented and imperative programming with functional programming. In addition, it introduces various language features that provide improved compatibility with other programming languages.

This can make it appear as a heavyweight language, a property that is generally not seen as desirable. Yet, when used correctly, only a subset of F# is relevant for any specific case. For example, multiple features are designed to improve language compatibility; they are *only* intended for interfacing with another language. For its internal operations, a program properly formulated in F# need not rely on such language features, even though some of them are seen as essential or useful in other languages. The fact that an F# programmer can get by without them is due to the way the language combines features from different programming paradigms.

Many features, especially amongst those of non-functional origin, are widely used in languages that influenced the design of F# but are discouraged or even unsupported in F#. This is usually the case when they are redundant to another feature of different origin which the language's designers deemed preferable.

This way that F# cherry-picks or neglects features is not always intuitive, especially when one is not used to the paradigms the chosen feature originates from. For example, in C#, the concept of *nullable types*, where *null* is a *non-value*, are a prominent feature that enables programmers to declare variables that might either have their designated type or not exist at all. F# allows nullable types only for compatibility. They are replaced by the almost identical *options*, which can explicitly be *some* object of a predefined type or *none*. The reason behind this seemingly absurd change is that options are not a core language feature, but implemented in just three lines of code using *discriminated unions*, which are a general solution to problems involving heterogeneous data. (See F# learning resources [20] [7] for details.)

Language	Preferred style	Typing	OO	Functional	Interactive	Execution	Speed
FORTRAN	imperative	static	†	no	no*	assembly	fast
C	imperative	static‡	no	no	no*	assembly	fast
C++	imperative, OO	static‡	yes	no	no*	assembly	fast
C++11	imperative, OO	static‡	yes	partially	no*	assembly	fast
Java	OO, imperative	static	yes	no	no*	JIT (JRE)	med.
C#	OO, imperative	static	yes	partially	in Mono	JIT (CIL)	med.
F#	functional, OO	static◇	yes	yes	yes	JIT (CIL)	med.
Python	imperative, OO	dynamic	yes	partially	yes	interpreted**	slow

† Depends on version. FORTRAN 2003 introduced multiple object-oriented features.

‡ My personal experience raises concerns about type safety in C and its direct descendants.

◇ F# uses inferred typing but applies very strict rules to type casts and introduces units of measure. It uses structural comparison in some of its constructs, but the cases are not considered dangerous to type-safety.

***** External tools provide interactivity of varying quality.

****** Python uses byte code that can be partially JIT-compiled. IronPython JIT-compiles to CIL.

Figure 4.1: A comparison of languages considered for this thesis, listing for each its preferred style(s) (the paradigms endorsed by structure and syntax), its preferred/primary type system, whether it can formulate object-oriented programs, whether it can formulate functional programs, whether it has a native interactive mode, its method of compilation or interpretation for execution, and a rough indication of the typical performance of programs in execution. For the latter, languages are divided into three classes: languages that are used in the fastest CPU-run programs, languages that compromise between features and performance, and the interpreted language Python that allows "duck typing" at the expense of type safety and performance. *The procedural imperative paradigm is omitted in the listing as all languages support it.*

4.2 The functional paradigm

Functional programming is a declarative style of programming derived from lambda calculus. Unlike the common imperative style, which performs computations via a program state and routines to mutate this state, it approaches problems through variable binding and the evaluation of functions. Where an imperative programmer uses a `for` or `while` loop, a functional programmer might use a recursive definition; where an imperative programmer changes a position variable, a functional programmer defines the shifted position with respect to the original one. Altering the original position is not a possible (or meaningful) operation in pure functional style, similar to mathematics, where changing a variable's meaning is not a sensible operation. (However, redefining a symbol is meaningful in mathematics and valid in functional programming.)

The *function as a first-class value* is a core principle of the functional programming paradigm. Defining a function within another and returning it, or passing it to yet another function, is a common sight. *Higher-order functions* that take other functions as input parameters are frequently used in high-level control code; this thesis has already seen their usage multiple times, for example in the partitioner of Section 2.4 (that takes a distribution function as input) or in the usage of `Seq.map` in Figure 2.10 on page 20.

Such operations do not require convoluted special syntax, additional explicitly created objects, or other such detours, as necessary in many imperative languages. Functions can be declared *anonymously* – without naming them with an identifier – directly where they are used, just like integers can be written as numbers directly where they are used.

An impression of a functional programmer's view

on the paradigm is given in the foreword of Expert F# 2.0 [7], a book on the language authored by its designers:

"Functional programming today is a close-kept secret amongst researchers, hackers, and elite programmers at banks and financial institutions, chip designers, graphic artists, and architects. As the grandchildren of Lisp, functional programming languages allow developers to write concise programs that are extremely close to the mathematical models they develop to understand the universe (...) However, to the uninformed developer, functional programming seems a cruel and unnatural act, effete mumbo jumbo."

Imperative and functional programs can be formulated to be equivalent and, with some drawbacks, be used in combination.

Immutability

Mutable types, as opposed to immutable types, are those data structures that can partially or completely change their data. In principle, this concept can be used outside of functional style and programs can mix both kinds of types as required. F# supports both mutable and immutable data types, but is designed such that immutable types are as short and comfortable in usage as possible, while mutable types and mutation operations use special syntax.

At first glance, it may seem absurd that functional languages deliberately prefer, or even enforce, an *inability* to change.

Removing mutability can be preferable because it reduces spurious interactions. This step is a logical continuation of the shift away from *global state machines*, which rely on a program-wide mutable state that is read and written to from many different routines. These can be hellishly error-prone, as programmers who write the individual routines make assumptions on the state that do not hold. A sufficiently complex program involving state has the same fundamental problem. It can suffer from all the typical issues of a global state machine if formulated unfavorably.

When a variable's value changes, its *meaning* changes. After such a change, running any operation that relies on the former meaning will fail. This creates code that must adhere to strict limits on its order of execution. Any change to this order may impede the program's validity, often under nontrivial conditions. Reformulating an algorithm to use immutable types removes this entire class of problems:

- The reordering of code based on immutable types is unproblematic. Undefined inputs will cause a compiler error, notifying the programmer about the inconsistency, while valid code remains valid independent of its new position. Reordering code based on mutable data can change its meaning in an arbitrary fashion, without any warning to the programmer.

- Immutable types are trivial to use as shared objects in parallel programs. Once constructed, they can be safely accessed by any number of threads without locks, atomics, or analysis of cache coherence. Mutable types, however, are extremely dangerous when used in parallel, possibly creating an exponentially large state space on a single error in synchronization.

- Analyzing the meaning of code based on immutable types does not require context beyond the declared meaning of the values it uses. For mutable types, analyzing the possible program states can become necessary, which can be a very complex and unpredictable problem.

Figure 2.10 on page 20 in Chapter 2 shows immutability in action. No value defined in the program ever changes; the code is merely a sequence of definitions that build on top of one another.

4.3 The Common Language Infrastructure

F# is part of a large effort to create a hardware-independent programming platform called the Common Language Infrastructure (CLI). It was originally developed by Microsoft and is currently standardized via ISO/IEC and ECMA [4]. Among its purposes are allowing the re-use of compiler features between languages as well as improving the interoperability between them. Recently, the CLI has gained additional interest from cross-platform development, since it allows for the re-use of executables on different operating systems and processor architectures.

When compiling a CLI program from source code, a language-specific compiler is called as usual. However, it is typically not compiled into hardware instructions, but instead into the Common Intermediate Language (confusingly abbreviating to CIL). The CLI normally compiles the final program from there. This final compilation step happens

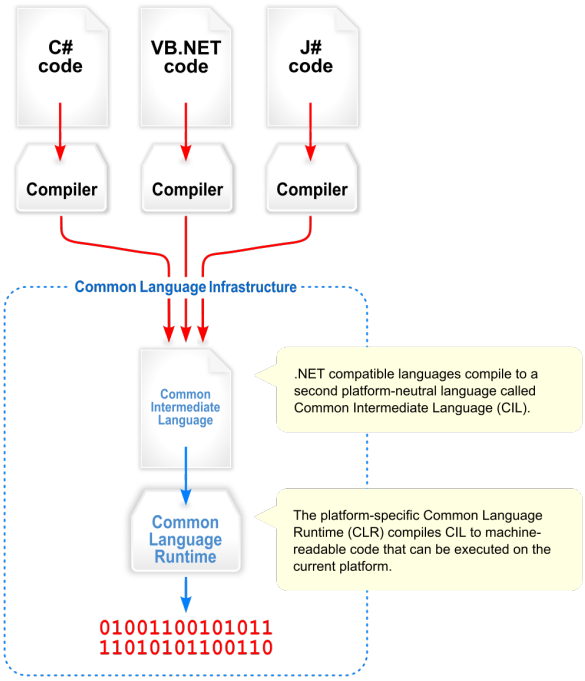


Figure 4.2: Languages using the CLI create similar and compatible compiler output that is also independent of the runtime and platform later used for execution. *Public domain image by Jarkko Piiroinen* [26]

only shortly before usage, a practice known as Just-In-Time compilation, allowing the usage of identical programs on different runtimes and platforms.

The CLI defines a large set of standard conventions for its languages. A common type system specifies not only basic types, but, in conjunction with a common language specification, enables compatibility of custom types between programs, even for those of different languages. This is as abstract and incredible as it sounds: complex, high-level data structures defined in one programming language can be used in another, directly, including metadata such as documentation snippets for development, without most of the quirks one might expect when doing such a thing. To use another program, it is simply loaded as a library file; any compatible features become available to the host program.

This is a huge step forward in my opinion. This thesis shows how plotting and rendering were included into programs. One might think that was tedious and error-prone. Actually, including both libraries and getting them to run was done in a matter of minutes. The difficulty was the lack of ready-made high-level functions for the renderer. Almost all of the programming time for visualization was spent interfacing the high-level code with the outdated OpenGL API of OpenTK – in part because of its reliance on the global state machine paradigm, which complicated integration.

CLI runtimes

There exist multiple implementations of the CLI, but the vast majority of current Runtimes is based on one of two projects:

- The Microsoft .NET Framework, which is the first implementation of the CLI, and its variants. See the MSDN [20] for information on the main version and the Compact Framework.
- The cross-platform Mono Project [19], of which the main code-base is licensed under the free software GPL license and maintained by Xamarin [32].

Frameworks of the .NET family offer good performance, but target only platforms of interest for Microsoft. Mono-based solutions have a very wide variety of platforms, from its main, free Runtime in Linux, Windows, and OSX, over Xamarin's proprietary Android version and cross-compiler to Apple iOS, to the Quantalea GPU compiler [27] privately offered to financial clients.

4.4 Performance

The speed F# programs execute with is currently notably dependent on the type of problem, language features used in the program and the Runtime it is executed on. F# is a statically typed language that is suited for compiler optimization, but the usage of functional and high-level language constructs tends to impede optimization by current compilers.

Generally, CLI compilers still yield assemblies of lower performance than that of the heavily optimized and specialized solutions like the Intel C compiler or Visual C++ compiler.

An anonymous employee of Trapeze Software has published a benchmark series in 2011 [28] that compares the Visual C++ compiler with the .NET and Mono CLI Runtimes for different settings and problems. In summary, it shows that the .NET Runtime is slightly slower than C++ VC9 and VC10 in comparable cases and about a factor 2 slower in cases that are problematic to optimize in C#,

which was the language the CLI programs for comparison used. Using Mono or x86-64 compilation in .NET results in a further performance penalty.

In the programming for this thesis, programs run on the Mono Runtime in SuSE Linux tended to run slower than on .NET in Windows, more so than the difference in the CPUs, RAM between the used machines should justify. The most significant case was in a C# implementation of 3D Fast Fourier Transformation (as-of-yet unused library feature, see Section A.4), where the slowdown depended on multi-threading settings and may have been dominated by an inefficiency in the dispatching of worker threads for parallel tasks in Mono.

The performance of F# on Mono when used for n-body simulations, one of the relevant cases for astrophysics, has been analyzed and compared to a multitude of languages and implementations in a series of benchmarks by Brent Fulgham [9]. The comparison is frequently updated; at the time of this writing, F# on Mono executes this benchmark at 2.8 times the time of the fastest implementation.

From a high-performance computing standpoint, these are not very good results. However, this must be put into the perspective of intended use cases. The most prominent project similar to the programs drafted for this thesis is the Astrophysical Multipurpose Software Environment (AMUSE) [1], which uses Python as its front-end. In the above-mentioned Fulgham n-body benchmark, Python 3 had an execution time about 36 times longer than F# had even on Mono, 100 times slower than the winner. An exact comparison would depend on the Python implementation and problem, but given its design as a dynamically typed scripting language, it seems unlikely that Python can compete with F# on either CLI Runtime in performance. (A CLI implementation of Python called IronPython exists, but does not change this situation.)

Outlook

F# and the CLI are not in principle constrained in terms of performance. Rather, their compilers and tools are much newer than those of C or C++ and thus have undergone less optimization. The performance of Mono especially may improve. Mono is a younger re-implementation of the .NET Runtime, which included many of its features only in recent years, and subject of ongoing development. Mono is transitioning to compilation via LLVM [17], which is designed to provide reusable optimization algorithms for compilers. In LLVM-enabled Mono, code is compiled in three steps: from its original language first to CIL, then to LLVM Intermediate Form, and finally to assembly for execution. As LLVM is used in many other compilers, there is a high level of interest in its further optimization, from which the Mono compiler would benefit.

Projects exist to compile F# for GPU, most notably a commercial framework by Quantalea [27] called Alea.cuBase that seamlessly integrates GPU programming with code executed on the normal CLI Runtime. It is also utilizing LLVM, but in this case to interface with NVidia CUDA [6], the compiler that also handles Espresso's performance-intensive SPH routines. Due to the concepts of *computation expressions* and *code quotations* in F# (see the MSDN [20] or Expert F# [7] for details), the language itself requires no modification to work with code blocks that are intended for special use instead of normal compilation. Alea.cuBase is a commercial product intended for financial analysis, with a restrictive license and case-by-case pricing, therefore it was not evaluated for this thesis. *As far as I could tell, other projects did not yet seem viable for deployment.*

4.5 Units of measure

The F# language allows values to be denoted in physical units of measure, such as the units defined in the SI and CGS systems. Computations involving values tagged this way will be checked for inconsistencies either when the program is compiled or even live by the IDE. The CIL byte-code output by the compiler holds information regarding units of measure only as metadata, so that the performance of the running program is not affected. Other programming languages can call library functions written in F# without a need to heed any special rules. Programs in F#, however, will fail to compile if units are specified incorrectly. This includes unit errors in calls to library functions such as the ones introduced in Chapter 2.

This feature deserves special mention as it proved tremendously useful in two ways. First, working with the wide array of units used in astrophysics, such as denoting lengths in cm, m, AU, light-years, and parsec, is neither an issue nor does it draw much attention away from the physical problem at hand. And second, even more helpful, is the system's ability to highlight inconsistencies due to incompatible units, caused by mistakes such as a missing factor in a calculation.

In the programming for this thesis, the IDE and

unit of measure system caught the majority of numerical mistakes within seconds of the faulty code being written.

4.6 Typing and type safety

Language support for units of measure can be seen as one specific tool concerning the much more general topic of *type safety*. In modern programming, expressions that implicitly convert between ways to interpret a value are considered *not type-safe*. For example, in C++, the following statement is valid:

```
if (myInt - 15) Start();
```

This line of code executes `Start()` if `myInt` $\neq$ 15, because in C++, the number zero is also the boolean value *true*.

Such expressions can shorten code, but from an abstract standpoint, the line is a nonsensical statement. Proponents of type safety argue that features of this kind are dangerous because they can cause compilers to output an absurd interpretation of a program where they should output an error message.

Most typing can be categorized using the following categories:

- *Strong* versus *weak* and *duck* typing: in strong typing, objects are either explicitly compatible or not. Weak typing, or its extreme case of "duck" typing, "make ends meet" if they can, even if this involves conversions.
- *Static* versus *dynamic* typing: dynamic types can change at run-time, static types cannot.
- *Explicit* versus *implicit, inferred* typing: in explicit typing, the programmer must state types explicitly. In implicit typing, only the minimal necessary information need be provided for code to be valid.

It is important to distinguish between *inferred static* typing and *dynamic* typing, which can appear very similar at first sight. Inferred typing, sometimes called implicit typing, does not alter the compiled program or any other behavior of the compiler, it only allows omitting annotations where they are obvious from context.

F# picks *strong static inferred* typing and is very consequent about type safety. Wherever possible, F# predicts the types of user-defined values, so programs only require minimal type annotations. But even something as simple as the lossless conversion from a 32-bit integer to a 64-bit integer must be explicitly declared, or else the program will fail to compile. *There is the exception of structural comparison, but discussing this topic is omitted here as it would go beyond the scope of this thesis.*

I believe that this choice of typing discipline is the correct choice for the majority of applications in physics – and also in other disciplines that involve high complexity and a low tolerance for errors. Dynamic typing is slow, cannot be reliably analyzed by IDEs, and is inherently unsafe, creating unnecessary room for errors for little benefit. Weak typing removes safety exactly where it is needed most: good code tends to execute numerical conversions at few, well-defined locations in the program. This means that weak typing saves its user a few local annotations that probably then become comments instead. The cost of this is the uncertainty that any part of the program may accidentally cause conversions, inducing a wide range of possible errors, with little indication as to where this might happen.

The case for implicit typing is, in a way, the opposite of weak safety in conversions. The benefit of being allowed to omit redundant type annotations is big, as these can significantly inflate code, especially when using descriptive type names. The programmer is free to write redundant annotations if it improves clarity. Should an invalid definition occur, the programmer can insert additional annotations and the development environment (or interactive compiler) can highlight the contradiction with increasing precision. There is virtually no downside to this approach, on the contrary: *automatic generalization* allows code to be used in its minimally constrained meaning, enabling it to remain type-safe while being valid for multiple different types at once, without additional programming effort.

4.7 IDE options

The integrated development environment (IDE) is a major component of modern programming. It assists the user in writing, debugging, analyzing, editing, visualizing, organizing, compiling, and testing programs, among many other possible features. The preferences in IDE choice vary wildly between use cases and users, with some using simple text editors, sometimes without even the highlighting of syntax in color. Others use specialized editors for graphical applications, as offered for mobile phones by Xamarin [32].

Figure 4.3 shows an experiment that combines a plotting window with a graphical user interface and

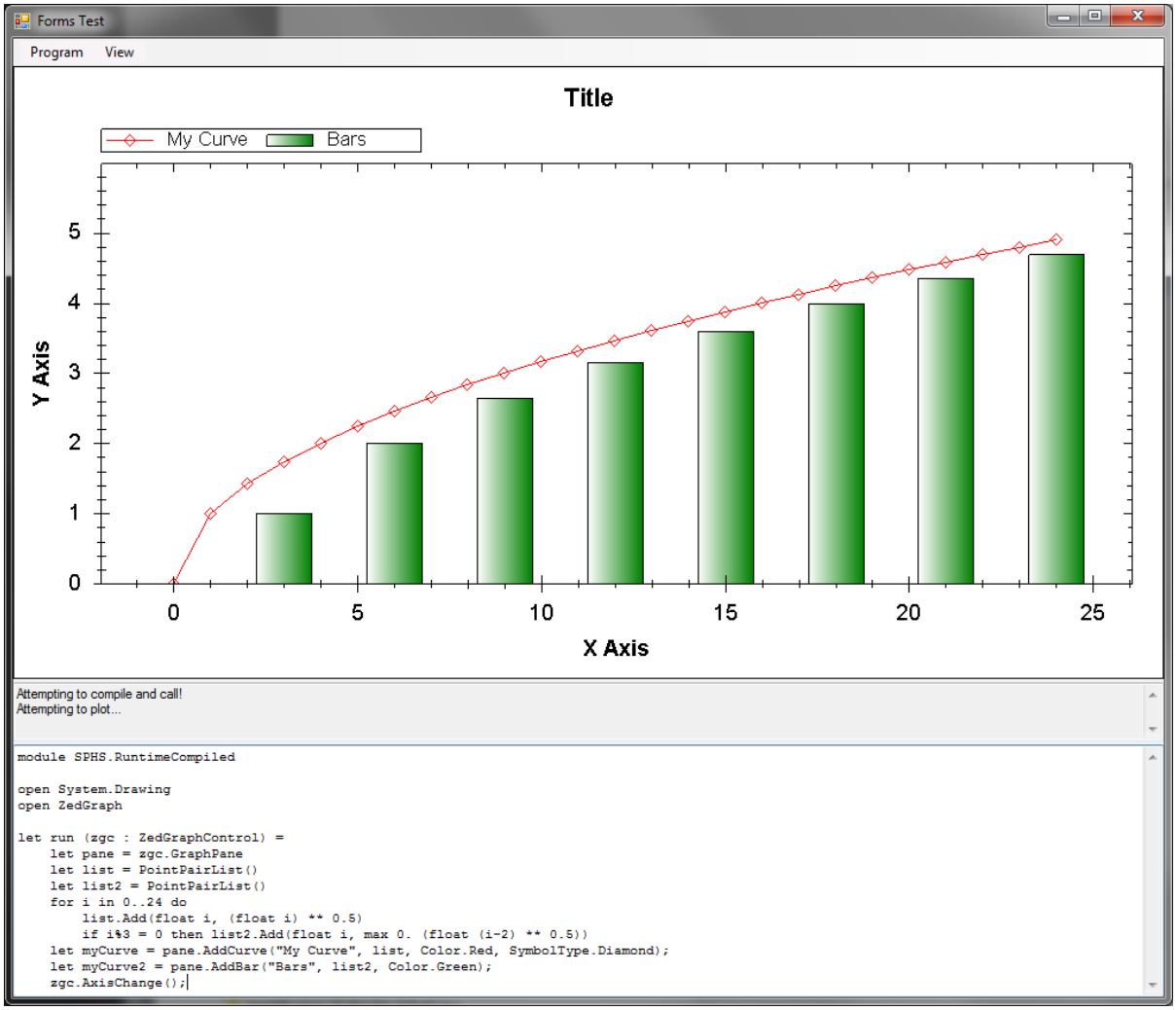


Figure 4.3: A forms-based cross-platform plotting window with a built-in F# compiler. This program was one of the plotting methods created for the SPHS library, but was abandoned since it offers little advantage over either a full IDE or the F# interactive console, while not benefiting from the features of either. It still serves to demonstrate the versatility of the library-focused programming methods that both created it and made it obsolete.

an F# compiler. It worked flawlessly, but was still discarded, because its user interface is redundant with the versatility F# programming offers anyway – and a plotting window can be launched from programs written in any F# IDE, including the F# *interactive console*, which is a simple console window operated by typing line-by-line F# code. Calling Chapter 2's SPHS library features from the interactive console worked immediately on both Windows .NET and Linux Mono. With a variety of alternatives, the plotting program was redundant.

4.8 Evaluation summary

Before we draw conclusions for the case of numerical physics, let us summarize the traits of F# in general. Amongst its prominent disadvantages are:

- Interacting with native libraries can cost performance and involve some additional effort.
- Its compilers and thereby run-time performance is not on par with the high-performance competition. (As seen in Section 4.4.)

- For very small and simple tasks, it can be slightly more verbose than dynamic languages, which can become the limiting factor in such use cases.
- F# feels alien to many and may be more difficult to learn than other languages.

General advantages of F# are:

- It is an expressive language that is well suited to represent mathematical models.
- Its design is directed at reducing the error rate when programming.
- The language is suited for live analysis and feedback from IDEs.
- Compiled programs, especially libraries, are automatically cross-platform usable, unless they rely on native code.
- It features very good interoperability to CLI libraries, including those written in other languages.
- It has a very lightweight syntax when compared to other strongly typed solutions to complex problems.

Distinguishing between high-level and high-performance code

Physical applications tend to face a slightly different situation compared to typical general-purpose programming projects. Some prominent differences are:

- Applications tend to be performance-intensive.
- Physicists spend much time programming and are willing to learn new concepts.
- Complicated, mathematically formulated problems are common.
- Units of measure are frequently used.

Of these four points, three arguably support the usage of F# and one, performance, does not. This suggests to distinguish between programming techniques for projects that are primarily limited by computer performance and those that are primarily limited by the writing of programs.

It may be a sensible decision to support a divide between programming languages that solve either problem, to then use two languages in combination. In most programs, the performance-intensive code is only a fraction of the program and increasingly executed on GPUs. Writing exactly these parts in a language that allows highly optimized routines, while using a safe and expressive high-level language for the remainder of the program, might be a very strong combination. Such a divide could have the consequence that languages forming a "middle ground" between the two could lose importance, as a case-by-case combination of the extremes may be hard to compete with.

The SPHS Library shows this approach when it interacts with Espresso: the SPH simulation in Espresso is executed using CUDA, which offers extremely high performance by running on parallel GPU multiprocessor units, but cannot compete with F# in other aspects.

4.9 Closing remarks

This thesis has demonstrated a modern programming style with functional emphasis by using it in multiple applications relevant to physical research. We have seen the abstraction of an inherently imperative high-performance GPGPU simulation of fluid dynamics for inclusion in an entirely separate analysis software that uses a strongly functional formalism. Example code demonstrates that the definition of a simulation can be of minimal length, retain type safety, and still be very clear and concise. We have also seen the implementation of a grid-based simulator in F# and the integration of various tools into the setup and analysis process.

The usefulness of many of the shown methods may depend on the scope of the projects they are used for. Strong typing, functional programming, library interoperability, cross-platform usage, the combination of multiple languages and compilers – implementing such concepts requires some extent of effort even if they seamlessly integrate into a project. The programming style used here is intended for large projects with high complexity; it likely would show greater effect in a project involving more than one to two people.

But this is where great potential may lie hidden. Many projects *do* involve a lot of people, high complexity, and an almost desperate demand for improvements in usability. There is much work to be done to improve workflow itself – and quite possibly, the effort would amortize.

A Appendix

A.1 EBT file format specification

EBT (Easy Block Tree) is a recursive format for structuring serialized data.

- The EBT format is a sequence of blocks.
- Integers defined in this section are serialized in little-endian format.

A block always consists of three parts:

- 2B (signed Int16) Block type ID; sign indicates whether the length field is 1B or 8B
- Length of content and possibly isEBT flag, formatted depending on the sign of the ID:
  - If the ID is positive, the content length field is an unsigned Int8 (1B) indicating content length in bytes.
  - If the ID is negative, the content length field is an Int64 (8B).
    - A negative value is used if the block content is again in EBT format.
    - The absolute value indicates the content length in bytes.
- Content

A.2 EBT index specification

To allow quick access, an index may be provided. Indexing of individual block contents is optional.

File addresses

File positions are stored as little-endian Int64 values (8B size). The beginning of the file and first valid position is at 0.

Index Address Block

- Is a top level block
- ID = +32512 = $+7F00_{hex}$
- Content length 8B
- Must be the first block in the file
- Contains the address of the Block Index Block that holds the top level index

Index Container Block

- Is a top level block
- ID = −32512 = $-7F00_{hex}$
- IsEBT: contents have EBT format
- Contains only “Block Index Blocks”

Block Index Block

- ID (local) = -1
- Holds the addresses of all elements in a block.
- Is to be used as index for its own file if and only if it can be navigated to from the Index Address Block
- Contains a sequence of indexer structures

Indexer

- 2B target block ID
- 8B target block address; points to the very beginning of the block (first byte of ID)
- 8B target index address *option*. If not 0, it contains the address of the index block for the target block. If 0, it is meaningless.

Notes

For performance and easy writing, it is often suitable to write one Index Container with all desired indices at the end of the file.

A.3 Implemented Rules

The following lists implemented settings the SPHS library could pass to the development version of Espresso. They are written in the format `identifier : type`. The `IOFlags` type will be explained later in this section.

Main simulation definition

- `bounds : Cuboid<cm>`: Boundaries of the simulation
- `d3 : bool`: Use three-dimensional simulation, false gives two-dimensional
- `dt : float<s>`: Time between two snapshots
- `enableGravity : bool`
- `enableHydro : bool`
- `isIsotherm : bool`: Set to true for isothermal computation
- `iterationsPerSnapshot : int64 option`
- `snapshots : int64`: Number of snapshots
- `stepmax : int64 option`: $\log_2$ of maximum time-step size in units of dtmin

Numerical constants

- `cfl_scale : float option`: Courant number. Typical value is 0.15.
- `constEnergy : float<erg/g> option`: If set, all particles will have this specific energy throughout the entire simulation.
- `constMass : float<g> option`: If set, all particles will have this mass throughout the entire simulation.
- `gamma : float option`: physical γ value
- `gravConstant : float<cm3̂/(g s2̂)> option`: Gravitational constant to use if gravity has been enabled
- `gravSoftening : float<cm> option`: Scale for the gravitational softening parameter

Other settings

- `boundX : byte`: Boundary behavior for X-direction: 0uy = Open; 1uy = Periodic; 2uy = Reflecting; 3uy = Streaming in/out; Extensions may introduce more
- `boundY : byte`: *see above*
- `boundZ : byte`: *see above*
- `dataFileOutput : bool`: Specifies whether an output in the standard data format is created.
- `export : IOFlags`: Which particle fields to export when writing to file
- `glassPass : int64`: How many passes of glass generation should be performed? Default is zero.
- `isscale : bool`: Toggles Espresso's internal unit scaling.
- `simOutput : IOFlags`: What per particle data should the simulator output?
- `waitOnEnd : bool`: If set, Espresso halts after the simulation instead of terminating.

Internal and temporary features

- `cRhoMin : float option`: Lower density limit for CIT-SPH output in data format units
- `cRhoMax : float option`: Upper density limit for CIT-SPH output in data format units
- `cEMin : float option`: Lower energy limit for CIT-SPH output in data format units
- `cEMax : float option`: Upper energy limit for CIT-SPH output in data format units
- `EType : byte`: *experimental Espresso setting*
- `HType : byte`: *experimental Espresso setting*
- `MType : byte`: *experimental Espresso setting*
- `VType : byte`: *experimental Espresso setting*

IOFlags

This type allows to specify which data should be transferred to or from Espresso via bit-flags. It is best described by its definition, which is in binary form:

```
[<Flags>]
type IOFlags =
    | Nothing   = 0b0000000000
    | All       = 0b1111111111
    | id        = 0b0000000001
    | x         = 0b0000000010
    | y         = 0b0000000100
    | z         = 0b0000001000
    | vx        = 0b0000010000
    | vy        = 0b0000100000
    | vz        = 0b0001000000
    | m         = 0b0010000000
    | rho       = 0b0100000000
    | e         = 0b1000000000
    | Position  = 0b0000001110
    | Velocity  = 0b0001110000
```

A.4 Unused feature: 3D Fast Fourier Transform

A three-dimensional multithreaded Fast Fourier Transform has been implemented in C# for this thesis. Due to a shift in the project's direction it has not been used.

The intended use case for the feature was the setup of turbulent velocity fields, which can be more easily defined by their representation in Fourier space. A backwards Fourier transformation would then yield the velocity field for the particles.

For such a part of the program, C# may be preferable due to its availability of unsafe code blocks, which allow manual code optimization using pointer manipulation, as common in C++ programs. Such operations are normally forbidden in C# and F# due to their breaches of security features and bad compatibility with garbage collection. With the current compilers, the usage of unsafe code still improves program speed, so an unsafe C# code block was used for the inner operations of the Fourier Transformation.

This feature appeared operational; referring to it from the SPHS library and writing a function that applies the velocity field should enable the ability to set up turbulent velocity fields.

Bibliography

[1] F.I. Pelupessy, A. van Elteren, N. de Vries, S.L.W. McMillan, N. Drost, S.F. Portegies Zwart. The Astrophysical Multipurpose Software Environment http://amusecode.org/

[2] Atacama Pathfinder Experiment (APEX Telescope). Funded by: Max Planck Institut für Radioastronomie (MPIfR) at 50%, Onsala Space Observatory (OSO) at 23%, and the European Southern Observatory (ESO) at 27%. http://www.apex-telescope.org/

[3] Courant, R.; Friedrichs, K.; Lewy, H., 1928: Über die partiellen Differenzengleichungen der mathematischen Physik". Mathematische Annalen 100 (1): 32–74.

[4] ECMA standard 335; ISO/IEC 23271:2012: Common Language Infrastructure http://www.ecma-international.org/publications/standards/Ecma-335.htm http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=58046

[5] Comma-seperated values (non-standardized convention) commonly: RFC4810 http://tools.ietf.org/html/rfc4180

[6] Compute Unified Device Architecture (CUDA). Parallel computing platform. Devloped by Nvidia Corporation. 2007–2013 http://www.nvidia.com/object/cuda_home_new.html

[7] Don Syme, Adam Granicz, Antonio Cisternino: Expert F# 2.0. ISBN-13 (pbk): 978-1-4302-2431-0. ISBN-13 (electronic): 978-1-4302-2432-7.

[8] Microsoft Research. The F# programming language project. Lead developer: Don Syme. Under development since 2005. http://research.microsoft.com/en-us/projects/fsharp/ F# Software Foundation: http://fsharp.org/

[9] Brent Fulgham: The Computer Language Benchmark Game. Performing n-body simulations on Ubuntu in multiple languages. http://benchmarksgame.alioth.debian.org/u64q/performance.php?test=nbody

[10] Volker Springel: The cosmological simulation code GADGET-2. Mon.Not.Roy.Astron.Soc. 364 (2005) 1105-1134

[11] Thomas Williams and Colin Kelley and many others. Gnuplot 4.4: an interactive plotting program. March 2010 http://gnuplot.sourceforge.net/

[12] Goodman, A. A., Barranco, J. A., Wilner, D. J., & Heyer, M. H. 1998: Coherence in Dense Cores. ApJ, 504, 223

[13] A. Hacar, M. Tafalla, 2011: Dense core formation by fragmentation of velocity-coherent filaments in L1517

[14] The HDF Group. Hierarchical data format version 5, 2000-2010 http://www.hdfgroup.org/HDF5

[15] Francesca Iannuzzi, Klaus Dolag: Adaptive gravitational softening in GADGET. Mon.Not.Roy.Astron.Soc. 07/2011

[16] IDL (programming language and visualization toolkit). Original design: David Stern. http://www.exelisvis.com/ProductsServices/IDL.aspx

[17] LLVM Developer Group: LLVM Compiler Infrastructure. (Formerly: Low Level Virtual Machine.) http://llvm.org

[18] R.A. Gingold and J.J. Monaghan: Smoothed particle hydrodynamics: theory and application to non-spherical stars, Mon. Not. R. Astron. Soc., Vol 181, pp. 375–89, 1977.

[19] Mono Project: A free and open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. http://www.mono-project.com

[20] The Microsoft Developer Network. Online documentation for F# and the .NET Framework. http://msdn.microsoft.com/

[21] Nagai, T., Inutsuka, S., & Miyama, S. M. 1998: An Origin of Filamentary Structure in Molecular Clouds. ApJ, 506, 306

[22] Octave community: GNU/Octave. 2012. http://www.gnu.org/software/octave/

[23] The Open Toolkit http://www.opentk.com

[24] J. Ostriker, 1964: The Equilibrium of Polytropic and Isothermal Cylinders

[25] S. Oxley: Free-fall derivation of a uniform density spherical distribution of matter initially at rest http://www.droxley.freeserve.co.uk/node15_ct.html

[26] Jarkko Piiroinen User of Wikimedia Commons http://commons.wikimedia.org/wiki/User:Jarkko_Piiroinen

[27] Quantalea GmbH. Wasserfuristrasse 42 CH-8542 Wiesendangen. https://www.quantalea.net/

[28] Anonymous Codeproject user "Qwertie" of Trapeze Software Inc. Head-to-Head benchmark: C++ vs .NET http://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vs-NET

[29] Markus Schmalzl, Jouni Kainulainen, Ralf Launhardt, Thomas Henning. Star formation in the Taurus filament L1495: From Dense Cores to Stars The Astrophysical Journal, Volume 725, Issue 1, pp. 1327-1336 (2010).

[30] Sommer-Larsen et al, 1996: The structure of isothermal, self-gravitating, stationary gas spheres for softened gravity

[31] V. Springel and C.P. Dullemond: Numerical fluid dynamics (lecture script 2012-2013) Universität Heidelberg

[32] Xamarin (company). Maintainer of the GPL-licensed CLI runtime Mono. http://xamarin.com

[33] John Champion and the ZedGraph development team: ZedGraph class library http://sourceforge.net/projects/zedgraph/

[34] Martin Zintl. CAST group. USM (Universitätssternwarte LMU München). zintl@usm.uni-muenchen.de

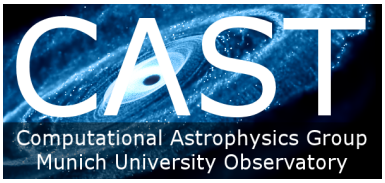
Danksagung

Ich möchte mich bei Prof. Dr. Andreas Burkert bedanken, der diese Arbeit betreut hat und mir viel Freiheit, gute Ratschläge, vielfältige Aufgaben und eine interessante Zeit beschert hat.

Ich bedanke mich bei Martin Zintl und Christian Alig, mit denen ich viel Spaß bei der Zusammenarbeit hatte, sowie bei der gesamten CAST-Gruppe für das großartige Arbeitsklima. Weiterer Dank geht an Christian Winnerlein für den Code zur effizienten Verwaltung der Zufallszahlengeneration und das absolut notwendige Mir-auf-die-Füße-Treten, damit ich mich rechtzeitig mit der Universitätsbürokratie auseinandersetze.

Ein extra Dankeschön geht an unsere Systemadministratoren Tadziu Hoffmann und Keith Butler, mit deren Hilfe ich allerlei Dinge auf Uni-Arbeitscomputern testen konnte und die ihn jederzeit wieder zum Laufen gebracht haben, wenn er einmal den Geist aufgegeben hat.

Zwischendurch entstanden:
Idee für ein neues CAST-Logo...



...mit einem Bild von Christian Alig. :)
(Falls ihr es nicht braucht, macht auch nichts.)

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

Unter den Hilfsmitteln befanden sich selbstverständlich Computer und ihre übliche Arbeitsumgebung, Software und sonstige gängige Werkzeuge. Einige bisher nicht explizit erwähnte solche Werkzeuge sind:

Additional tools used

Corel PaintShop Pro X4 (personal license, for creating infographics), the Microsoft Developer Network (major documentation resource for F# as well as the .NET CLI implementation and library), Wolfram Alpha (for calculations, integrations, and other mathematical tools), Git and Gitolite version control (to allow parallel updates on Espresso and convenient merging thereof), CIT-SPH (An earlier Espresso visualization and format of Martin Zintl used for preliminary analysis and consistency checks while implementing features for Espresso), Acadoid Developer's Lecture Notes on a stylus-enabled Galaxy Note 10.1 tablet PC (for drafting, sketching, and writing calculations)

And, last but not least, the LaTeX2e and the Memoir package, edited in WinShell and SumatraPDF, for the thesis's layout. (The interoperability between SumatraPDF and WinShell is neat.)

Georg Michna, September 30, 2013