Draft version June 11, 2018
Preprint typeset using LaTeX style emulateapj v. 04/17/13

PHD: PYTHON HYDRO-DYNAMICS CODE FOR ASTROPHYSICS

Ricardo Fernandez[1] and Greg L. Bryan[1,2]

Draft version June 11, 2018

ABSTRACT

We describes the structure and implementation of the open-source parallel moving-mesh hydrodynamic code, Python Hydro-Dynamics (`phd`). The algorithms are largely based on the `AREPO` code, described by Springel (2010). However, `phd` has been written from the ground up to be easy to use and facilitate future modifications. The code is written in a mixture of `Python` and `Cython` and makes extensive use of object-oriented programming. We outline the algorithm and describe the design philosophy and the reasoning of our choices during the code development. We end by validating the code through a series of test problems.

Keywords: methods: numerical — hydrodynamics

1. INTRODUCTION

The use of numerical simulations for modeling astrophysical compressible gas flows has steadily grown over the past few decades, rapidly becoming a key investigative tool for science domains ranging from planet formation to cosmology. This has come about due to a number of reasons. First, the decreasing cost and increasing performance of the hardware has played a key role, although in recent years the chip clock speed has remained constant, forcing performance improvements to move toward the use of large scale parallelization. In addition to technology improvements in hardware, remarkable progress has been made over the years through the development of algorithms that leverage computing resources to the problem at hand. Finally, the implementation of those algorithms in open source codes has also played a key role in bringing computational astrophysical fluid dynamics to a wider audience.

Despite these advances, there remain unresolved issues: computational tools are not as widely used as they could be, they require more effort to adapt they should, and are often used as black boxes due to their complexity and challenging software design.

The implementation of a particular algorithm can vary substantially due to choices of programming language and code design. Commonly, codes are born out of the necessity and/or a specific domain interest of the researcher or group. These codes are usually tested on a similar set of problems to ensure that they reproduce a basic set of known solutions, although, of course, differences – sometimes substantial – are still present. The result of this process is a wide range of different code bases, many of which have grown well beyond their original design specifications. Indeed, many were not formally designed to begin with and grew organically. As the range of physical processes the codes try to model grows, so has the complexity.

From the point of view of a newcomer, choosing and modifying an existing code for a specific research problem can be a daunting task. Codes are often developed with the mindset of optimization and scaling. Thus many are written to take full advantage of specific hardware or low level language details. This can generate code that requires an avalanche of changes to accommodate new algorithms or, even worse, a code that is opaque and inflexible to modifications. Compounding the situation is that most codes are written in `fortran` or `C`. Although these languages generally produce efficient, optimized code they come with the price of a steep learning curve. Moreover, these languages make it challenging or impossible to implement modern programming practices (in particular, object oriented design – note that, although C++ adds object-oriented design to C, arguably this layering of high-level tools on a low-level language is less than ideal).

For the reasons outlined above we have decided to produced a new code motivated by a philosophy that emphasizes simplicity, usability and extensibility. Our code is called `phd` and has been developed in the `Python`[3] programming language. The code makes substantial use of object-oriented design, allowing it to be easily extended and modified. As a first attempt, we have implemented the Arbitrary Langrangian-Eulerian (ALE) moving mesh scheme for hydrodynamics, outlined by Springel (2010), although our hope is that a wide variety of algorithms can be accommodated by the code framework.

The structure of this paper is as follows. In the rest of the Section 1, we go into the motivation of using ALE and Python, followed by summarizing the overall design philosophy of the code. Next, in Section 2 we briefly describe the main algorithms used in the code. Then in Section 3, we go into detail on how the algorithms are implemented and the resulting class structure. This is intended to give the reader an understanding and rational for the design choices we made, as well as to give examples on how to use and modify the code. Finally, in Section 4 we present the results of a number of test problems. We end with a discussion and an outline of future improvements in Section 5.

1.1. *Motivation for ALE*

[1] Columbia University, Department of Astronomy, New York, NY, 10025, USA
[2] Center for Computational Astrophysics, Flatiron Institute, 162 Fifth Avenue, New York, NY 10010
[3] `https//www.python.org`

Numerical simulations in astrophysics have become an essential tool and due to the wide ranges of scales, it is often useful to have an adaptive algorithm – that is, one for which the resolution of the method can vary depending on the solution. A variety of adaptive hydro methods have been used to solve the nonlinear hydrodynamics equations in cosmological problems. The two most notable are Smoothed Particle Hydrodynamics, or SPH (Lucy 1977; Gingold & Monaghan 1977; Monaghan 1992) and Adaptive Mesh Refinement, or AMR (Berger & Colella 1989; The Enzo Collaboration et al. 2013). However, even with the successes of these techniques, there are situations in which these methods behave very differently, even for basic problems that only involve non-radiative hydrodynamics (e.g., Agertz et al. 2007; Tasker et al. 2008; Mitchell et al. 2009). This is disconcerting since many of the more complex processes such as radiative cooling and star formation are not included.

SPH has a simple approach for numerical fluid dynamics in which the flow is represented by a set of discrete particles that interact with neighboring particles and long range forces. It Lagrangian nature and ability to automatically adapt to the flow, follow certain conservation laws, and its ease in incorporating gravity makes it well suited for gravitational collapse problems. However, SPH suffers in accurately capturing certain fluid mixing processes, such as Kelvin-Helmholtz instabilities which are usually found to be suppressed in growth. Alternative formulations of SPH have worked to remedy this problem (Monaghan 1997; Ritchie & Thomas 2001; Price 2008) by recognizing that the pressure at contact discontinuities must be single valued. However, many of these formulations either introduce new dissipation terms and/or violate conservation laws. Moreover, SPH can have relatively poor discontinuity capturing properties relative to grid based methods, including the spreading of shocks and contact discontinuities.

Unlike SPH, AMR is commonly used with a static discrete domain (i.e. it is Eulerian), over which the algebraic analogues of the fluid equations are solved. Frequently in astrophysics, gravity plays a dominant role in driving the flow to condense in certain regions. The result is that highly dense regions evolve on shorter time scales relative to low dense environments. For large multi-dimensional flows this becomes problematic because a fraction of the domain is dictating the evolution of the simulation at the expense of computing and memory resources. AMR addresses this situation, by starting with a coarse grid and only allowing regions flagged by a specified parameter, for example truncation error, to be refined until a maximum level has been reached or an error has been minimized to a desired level. Similar to SPH, AMR has its weakness too and most notable is that the solution can depend on the bulk velocity of the flow (Wadsley et al. 2008; Tasker et al. 2008). Different bulk velocities, with the spatial resolution held fixed, generates velocity-dependent numerical diffusion that leads to differing results. This is of concern since, in astrophysics, we can often have flows that are orders of magnitude larger than the sound speed. Robertson et al. (2010) has shown that the errors produced by numerical diffusion become negligible as the resolution of the simulation is increased to obtain a converged solution. Unfortunately we do not know what resolution scale will be convergent until the simulation is run.

Recently there has been a new-found interest in ALE methods (e.g., Whitehurst 1995; Springel 2010; Duffell & MacFadyen 2011) which combine both SPH and AMR methods harnessing the strengths of each method. The basic idea is to allow the mesh to move with the flow therefore having a Lagrangian character but solving the equations with grid-based methods commonly used in AMR techniques. Consequently this can reduce the bulk errors found in AMR methods while resolving shocks and instabilities. It is also relatively easy to parallelize and works well with modern, high-performance self-gravity techniques. Although ALE methods are not a panacea (in particular the grid motion and changing structure itself can be a source of noise), they are a promising technique with potentially wide applicability while only a very small number of open source codes currently exist.

1.2. *Motivation for Python*

Many of the popular codes used today have been developed in lower level languages (i.e. `fortran` and `C`). Although these languages have the benefit of explicit memory management and good performance, they can be cumbersome for early programming practitioners. The difficulty to set up new problems and modify or extend the code can become challenging and time consuming. Moreover, many codes are designed and optimized for a specific set of applications which may result to a code that is unwieldy to generalize. For these reasons, we wanted to create a code with a framework that would allow and encourage the user to experiment and perform modifications, and one that is written in a language widely used in the target domain.

To create such a framework we decided to use the `Python` programming language as the core language in our new hydrodynamics code. `Python` is an open source, interpreted, high-level, and general purpose dynamic programming language that centers around the philosophy of code readability. The readability aspect allows the user to focus on the concepts instead of language specific syntax such as types, pointers, and memory management found in compiled languages. The syntax of `Python` permits programmers to perform the same operations as compiled codes, albeit in fewer lines of code. Moreover, since `Python` is an interpreted language, it gives us the ability to quickly prototype and experiment, unlike traditional edit-compile-run languages. Additionally, `Python` offers an expansive library of open source tools and a broad based community that provides many introductory examples and resources.

Although `Python` has many desired qualities, it does come with a cost. `Python` executes instructions with the aid of an interpreter instead of using a compiler, resulting in decreased performance. There are many libraries that can remedy this situation. For example, `Numpy`[4] allows vectorized operations for array based data. However, our general approach is to move the most demanding parts of the computation to a lower level language, in fact, this is exactly what `Numpy` does, and use a wrapper to make it accessible in `Python`. Since creating a code strictly in `Python` would

[4] `www.numpy.org`

overly constrain the performance, we have elected to move the most demanding portions of the code into `Cython` [5]. `Cython` is a superset of the `Python` language which produces `C`-like performance for code that is mostly written in `Python`.

1.3. *Design Philosophy*

Out hope is that the overall design of the `phd` code is a framework that simplifies the initialization of new problems, allows for extension of new algorithms, and permits the user to easily tailor specific applications during runtime. To implement these ideas several designs choices were taken.

Since `Python` was chosen as the primary language, many of the complications of memory management and initialization of variables where eliminated. For large data applications, this can be problematic, therefore we did add back specific memory handling for our field arrays, but this is done within the class structure and so is hidden from users outside of that structure. This means that no meta data initialization, pre-functions or pre-calls are necessary. Instead, the user is solely responsible for setting primitive fields by element-wise assignment or taking advantage of `Python`s slicing and broadcasting operations. The instructions can be written in a file or compactly defined in a function.

To allow the code to be easily extended, we decided to encapsulate the core computations into classes. This allows the ability to easily modify or implement new algorithms by inheritance. Further, the base class creates an Application Programming Interface (API). This means that each base class creates the rules and behavior on how the class operates. Any future extension of the class must conform and abide by the API. In this way, additions to the code are properly placed without the need to modify several other portions of the code.

Reducing each core computation into a class allows the code to be modular. Therefore many parts of the code can be easily swapped out for different implementations. In affect, the code becomes mailable, defined by a series of classes. The classes can be extended and modified through the use of inheritance as long as the basic API can be maintained. Such a framework allows us to quickly prototype several simulations with varied algorithms.

2. PHYSICS AND ALGORITHMS

In this section, we describe the physical equations we wish to be able to solve with this code and the algorithms that we will use to solve them. As described earlier, the code is designed to be extremely flexible so that we can implement a range of methods within the existing code framework. This means that the algorithms we describe in this section should be seen as a starting point, and that we hope and expect that the code can handle a much general expression of the physical equations, including different ways to discretize the space and integrate the equations. In particular, much of the discussion below is based around the idea of a Voronoi tessellation of the spatial domain; however, as described in more detail in the code design section, this is flexible and can be replaced by another mesh at a later time.

2.1. *Physical Equations*

The spatial and temporal evolution of a compressible fluid is governed by the Euler equations. The Euler equations are a set of hyperbolic conservation laws governing the density ρ, velocity $\mathbf{v}$, and total specific energy e (specific kinetic energy $\frac{1}{2}\mathbf{v}^2$ plus specific internal energy u). These quantities $\mathbf{U}$ and their respective fluxes $\mathbf{F}(\mathbf{U})$, defined through the Euler equations, are conveniently expressed in vector form as

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ \rho e \end{pmatrix}, \quad \mathbf{F}(\mathbf{U}) = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v}^T + P \\ \rho e\mathbf{v} + P\mathbf{v} \end{pmatrix}, \tag{1}$$

where P is the pressure and a superscript T indicates a transpose. In this notation the Euler equations can be expressed in the form

$$\frac{\partial \mathbf{U}}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F} = 0. \tag{2}$$

As stated, the system of equations are not closed – there are more variables then equations. Thus, an additional constraint is needed and is supplied by the equation of state, which we typically take to be the ideal gas relation

$$P = \rho(\gamma - 1)u, \tag{3}$$

where γ is the ratio of specific heats.

These equations can be solved by the finite-volume approach, which is a discretization of the domain into finite sized disjoint cells and evolves their spatially averaged $\mathbf{U}$ values. Specifically, applying equation 2 to every cell i with volume V_i and performing Gauss' theorem to convert the volume integral to a surface integral results in

$$\frac{d\mathbf{Q}_i}{dt} = -\sum_j \int \mathbf{F}_{ij}(\mathbf{U}) \cdot \mathbf{A}_{ij}, \tag{4}$$

where $\mathbf{A}_{ij}$ is the cell's surface area normal and $\mathbf{Q}_i$ is the volume integral of $\mathbf{U}_i$,

$$\mathbf{Q}_i = \begin{pmatrix} m_i \\ \mathbf{p}_i \\ E_i \end{pmatrix} = \int_{V_i} \mathbf{U}_i dV, \tag{5}$$

[5] `cython.org`

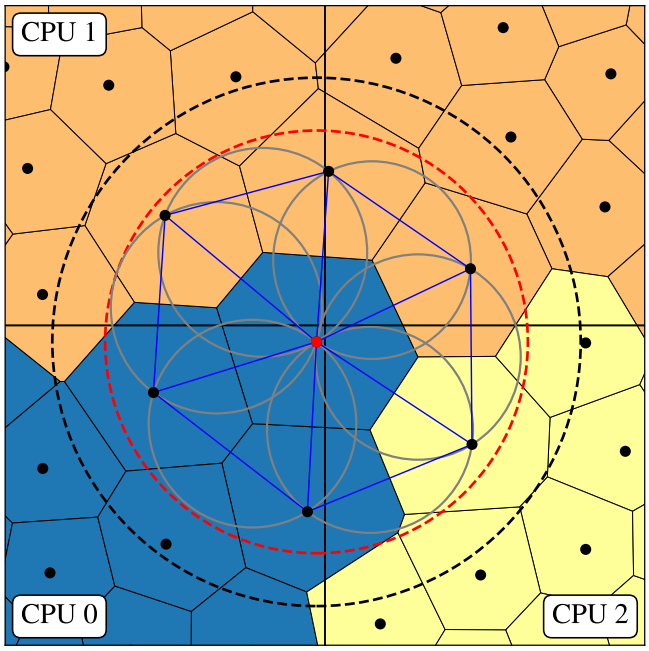

Figure 1. Voronoi tessellation performed on three processors. Particles are color coded by the processor in which they are stored. The Delaunay tessellation is shown for the red mesh generator in the center (in processor 0) as a set of blue lines. Each Delaunay line from the red mesh generator defines a neighbor and each pair of neighbours defines a triangle. The set of grey curves show the cirumcircles s_i constructed such that they intersect the central point and two Delaunay pairs The red dashed line is twice the maximum radius of all of these circumcircles, for the red mesh generator. The dashed black line is the search radius h_i that we adopt. Since $h_i > s_i$ the cell is complete, meaning all neighbors of the red generator has been accounted for.

where m_i, $\mathbf{p}_i$ and E_i are cell's total mass, momentum, and energy (kinetic plus thermal), respectively. Here, an assumption is taken that the cell's volume is a polyhedron such that the surface integral can become a sum over all polygon faces.

Additionally, time is discretized (with a superscript n indicating the timestep), leading to a finite difference update of the cell's quantities given by

$$\mathbf{Q}_i^{n+1} = \mathbf{Q}_i^n - \Delta t \sum_j \hat{\mathbf{F}}_{ij}^{n+1/2} A_{ij}. \tag{6}$$

In this expression, the flux $\hat{\mathbf{F}}^{n+1/2}$ is taken to be a time average and is constant across the cell face. To be able to use equation 6 we must make estimates of $\hat{\mathbf{F}}^{n+1/2}$ and A_{ij} to the proper order of accuracy. Sections 2.3 and 2.2, respectively, will go into detail on how these quantities are estimated.

2.2. *Tessellation*

In equation 6, we have written the flux update of a cell as a set of vector sums of fluxes across faces. Here we describe the (standard) tesselation that we will assume. There are many ways of partitioning space, but for this paper we will restrict our attention to the Voronoi tessellation. This partitions the space into a set of disjoint polyhedrons from a given set of points. The set of points are called mesh generators and for each generator there is a corresponding region consisting of all points that are closer to it than to any other mesh generator. Thus, for each neighboring mesh generating pair there lies a polygon that is equidistant between the two points (see Figure 1). For the rest of this paper we will refer to the polyhedra associated with the mesh generator as a cell and the polygons making up the polyhedra as faces. The creation of the Voronoi tessellation can be performed by first constructing the Delaunay triangulation. The Delaunay tessellation is the dual graph of the Voronoi tessellation and is defined as the triangulation of points such that no point lies inside the circumcircle of any of the triangles. Figure 1 depicts the Delaunay triangles for one mesh generating point (shown as a red dot). Notice that the mesh generators make up the vertices of the triangles and that no generator lies inside the circumcircles. Further, each edge defined by the the triangles associates a neighbor relation. Thus, all the neighbors of the red generator are connect by a blue edge. The dual aspect refers to the fact that each triangle edge corresponds to a Voronoi face and each Voronoi vertex corresponds to the center of the circumcircle of the corresponding triangle. In 3D the triangles and circumcircles become tetrahedrons and circumspheres, however, the same concept applies.

There are many suitable implementations for constructing the tessellation in serial (Bowyer (1981); Watson (1981); van de Weygaert (1994); Edelsbrunner & Shah (1996)). However, the challenge is to construct the tessellation when the points are distributed (i.e. in parallel). A common solution is to use a serial implementation coupled with a search

routine. In this procedure, particles local to the processor are used in the initial construction of the tessellation. At this point only interior cells of the tessellation are correct and cells at the surface are incorrect because they depend on particles that reside on neighboring processors. To complete the tessellation, particles must to be imported from neighboring processors. To find the correct neighboring particles a search radius h_i is used. The value of h_i is typically used from a previous time step or an initial value is given. All neighboring particles that fall in the union of h_is are imported to the processor and the tessellation is updated. To help determine when a cell is complete, every particle has a radius s_i calculated which is equal to twice the maximum radius of all the circumcircles. Due to the properties of the Delaunay tessellation, a particle can only influence another particle if it lies inside s_i. Thus, if the particle is not complete h_i is increased and the procedure is repeated until $s_i < h_i$. Figure 1 describes this process. The red mesh generator has h_i, shown by the black dashed line, and s_i given by the red dashed line. Notice that all the neighbors of the red point are contained within s_i. Since $s_i < h_i$ there are no particles that can influence the generator. Therefore the cell is complete.

In the procedure outlined above, notice that it involves a two way communication scenario. The search radius of every incomplete particle has to be exported to find the particles that need to be imported. Peterka et al. (2014) has simplified this process by exploiting the symmetry of the Delaunay tessellation. We highlight the main point but direct the reader to Peterka et al. (2014) and Morozov & Peterka (2016) for complete details. The idea is that if particle p on processor P_i needs to construct an edge with particle q on processor P_j, then it is also true that particle q needs to construct an edge with particle p (see Figure 1). Therefore, we only need to determine the particles that have to be sent to neighboring processors. This reduces the two way communication into a single direction communication, avoiding the need to search for neighboring particles contained in circumcircles. Therefore, every particle that has a search radius h_i that intersects a neighboring processor is exported to that processor. Then the local tessellations are updated and the process repeated until each cell is complete.

Once the cells are complete, geometric quantities (areas, volume, center of mass, ...) can be easily computed. Complete details can be found in Duffell & MacFadyen (2011) but for completeness we will outline some of the key computations.

Since the cells are complete, neighbors are defined for each mesh generator. Each neighbor defines a Delaunay edge, which its dual is a Voronoi face. This face is used in the Riemann solver (see Section 2.3.2) and the area can be calculated by summing half cross products of pair vertices along the convex hull. With the face areas calculated, the cell volume calculation is straightforward, since it can be decomposed as a sum of pyramids where the face area is the base and the height is the half distant between neighboring particles. Similarly, the center mass of the cell can be decomposed as a weighted sum of the center mass of each pyramid.

2.3. *Fluid Update*

Having described the procedure to generate a tesselation of the space and identify cells and their associated faces, we need a way to determine the fluxes at the cells faces. Our standard procedure is to adopt a Godunov approach (see Chapter 6 of Toro (1997) for a comprehensive description), in which the flux is computed in a two-step procedure. First, since the conserved quantities stored are cell-averaged values, we need to reconstruct the distribution of the primitive quantities within the cell and in particular at the cell faces (Section 2.3.1); second, we solve Riemann problems at the cell faces to estimate time-averaged fluxes (Section 2.3.2).

2.3.1. *Reconstruction*

To solve the Riemann problem, primitive values $\mathbf{W}^T = (\rho, \mathbf{v}, P)$ are needed at the center of mass of the faces. As a first choice, cell center values can be used. This procedure is known as constant reconstruction and can be viewed as the first term in the Taylor expansion of primitive values from the cell center of mass to center of mass of the face (gradients and higher order derivatives vanish; see Equation 7). For first order in space and time, the gradients can be included in the Taylor-series

$$\mathbf{W}' = \mathbf{W} + \frac{\partial \mathbf{W}}{\partial \mathbf{r}}(\mathbf{f} - \mathbf{s}_i) + \frac{\partial \mathbf{W}}{\partial t}\Delta t, \tag{7}$$

where $\mathbf{f}$ is the center of mass of the face, $\mathbf{s}_i$ is the center of mass of the cell, Δt is the time step, $\partial \mathbf{W}/\partial \mathbf{r}$ is the spatial gradient and $\partial \mathbf{W}/\partial t$ the time derivatives. This expansion has two unknowns, the spatial gradient $\partial \mathbf{W}/\partial \mathbf{r}$ and the time derivative $\partial \mathbf{W}/\partial t$. However, both derivatives are related through the Euler equations in primitive form

$$\frac{\partial \mathbf{W}}{\partial t} + \mathbf{A}(\mathbf{W})\frac{\partial \mathbf{W}}{\partial \mathbf{r}} = 0, \quad \mathbf{A}(\mathbf{W}) = \begin{pmatrix} \mathbf{v} & \rho & 0 \\ 0 & \mathbf{v} & 1/\rho \\ 0 & \gamma P & \mathbf{v} \end{pmatrix}. \tag{8}$$

Thus, only the spatial gradients need to be calculated. For the calculation we follow the method presented in Springel (2010), which is summarized below. Given a cell i the components of $\partial \mathbf{W}/\partial \mathbf{r}$ are given by

$$\nabla \phi_i = \frac{1}{V_i}\sum_j A_{ij}\left([\phi_i - \phi_j]\right) - \frac{\phi_i + \phi_j}{2}\frac{\mathbf{r}_{ij}}{r_{ij}}. \tag{9}$$

The sum is over all neighbors j of particle i, ϕ_i and ϕ_j are the scalar field values at each cell center respectively, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the separation vector with magnitude $r_{ij} = |\mathbf{r}_{ij}|$. As constructed, the gradients are second order

in space for smooth flows. However, in the presence of shocks, numerical instabilities may arise and therefore the reconstruction order must be reduced to prevent the creation of new extrema. To deal with this, a slope limiter is used. We have implemented two limiters. The first one is the method used by AREPO and begins with the calculation of

$$\psi_{ij} = \begin{cases} (\phi_i^{max} - \phi_i)/\Delta\phi & \text{for } \Delta\phi_{ij} > 0 \\ (\phi_i^{min} - \phi_i)/\Delta\phi & \text{for } \Delta\phi_{ij} < 0 \\ 1 & \text{for } \Delta\phi_{ij} = 0, \end{cases} \tag{10}$$

where $\phi_i^{max} = \max(\phi_j)$ and $\phi_i^{min} = \min(\phi_j)$ are the maximum/minimum values across all neighbors of i and $\Delta\phi_{ij} = \nabla\phi_i \cdot (\mathbf{f}_{ij} - \mathbf{s}_i)$. Then the minimum of all ψ_{ij}s, associated with each primitive field, is found, producing a single scalar value

$$\alpha_i = min(1, \psi_{ij}) \tag{11}$$

that is used to limit the gradient

$$\nabla\phi_i' = \alpha_i \nabla\phi_i \tag{12}$$

Once we have the gradient for each primitive variable, we can do a linear reconstruction to the face.

2.3.2. *Riemann Solver*

To update the conservative variables from Equation 6, we must estimate the time-averaged fluxes at the cell faces: $\hat{\mathbf{F}}^{n+1/2}$. These flux estimates are calculated from the the solution of the Riemann problem. The Riemann problem is defined by two constant states, $\mathbf{U}_L$ and $\mathbf{U}_R$, separated by a discontinuity at $t = 0$. An exact analytical solution exists that describes the time evolution of the system. The specific details of the solution can be found in many computational fluid dynamics textbook (e.g. Toro 1997). For clarity, we describe the main points of the solution. In Figure 2, we show the general structure of the solution. The vertical axis represents the time while the horizontal axis represents the position. As time progresses, three waves, labeled S_L, S_* and S_R, emanate from the position of the discontinuity. Each wave is associated with an eigenvalue from the Euler equations and carries a jump in the characteristic variables. The middle wave, called the contact discontinuity, carries a jump only in the entropy while leaving the pressure and velocity constant. The outer waves can be either shocks, if the characteristics converge, or rarefactions fans, if the characteristics diverge. The three waves define four states, which are left to right: $\mathbf{U}_L$ (left initial state), $\mathbf{U}_{*L}$, $\mathbf{U}_{*R}$ and $\mathbf{U}_R$ (right initial state). The unknown regions, $\mathbf{U}_{*L}$ and $\mathbf{U}_{*R}$, are called the star states and are separated by the contact discontinuity. The goal of the Riemann solver is to estimate these nonlinear waves and construct the flux at the characteristic $x/t = 0$.

The solution to the Riemann problem is given by the root of an algebraic equation which can be found, to a given tolerance, through a numerical iterative scheme. Toro (1997) describes an exact solver using an iterative Newton-Raphson approach. However, iterative approaches can be computationally expensive and time intensive. A more common approach is to use a non-iterative approximate solver. The Harten Lax van Leer (HLL, Toro 1997) solver is one of the simplest method by which the central contact discontinuity is ignored, leaving the star state to be an average of the outer waves. In practice the HLL solver is robust and efficient, however, ignoring the contact discontinuity makes the solver diffusive. The HLLC (Toro 1997) solver is an extension of the HLL method which includes the contact discontinuity. These three Riemann solvers (HLL, HLLC, and Exact) are implemented in `phd`.

2.4. *Grid Motion*

As currently constructed, the method outlined for solving the Euler equations are for static meshes only. In our case we also wish to allow the mesh to move, meaning that the mesh generators are given some velocity $\mathbf{w}_i$. Thus, equation 2 must be augmented to account for an advection term produced by the movement of the face. The updated Euler equations become

$$\mathbf{F}_m(\mathbf{U}) = \mathbf{F}_s(\mathbf{U}) - \mathbf{U}\mathbf{w}^T = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v}^T + P \\ \rho e\mathbf{v} + P\mathbf{v} \end{pmatrix} - \begin{pmatrix} \rho\mathbf{w} \\ \rho\mathbf{v}\mathbf{w}^T \\ \rho e\mathbf{w} \end{pmatrix}. \tag{13}$$

In practice this equation is not used because of its unstable numerical behavior (Pakmor et al. 2016). The equation can be numerically stabilized by solving the fluxes in the rest frame of the face. In this frame, the conserved variables and fluxes become (where we now use $\mathbf{w}$ to denote the face velocity).

$$\mathbf{U}' = \begin{pmatrix} \rho \\ \rho(\mathbf{v} - \mathbf{w}) \\ \rho e' \end{pmatrix}, \quad \mathbf{F}'(\mathbf{U}') = \begin{pmatrix} \rho(\mathbf{v} - \mathbf{w}) \\ \rho(\mathbf{v} - \mathbf{w})(\mathbf{v} - \mathbf{w})^T + P \\ \rho e'(\mathbf{v} - \mathbf{w}) + P(\mathbf{v} - \mathbf{w}) \end{pmatrix} = \begin{pmatrix} F'_\rho \\ F'_\mathbf{v} \\ F'_e \end{pmatrix}. \tag{14}$$

Notice ρ and P remain unchanged, however the energy transforms to $e' = e - \frac{1}{2}\mathbf{v}^2 + \frac{1}{2}(\mathbf{v} - \mathbf{w})^2$. Now the fluxes need to be transformed back to the lab frame

$$\mathbf{F}_m(\mathbf{U}) = \mathbf{F}_s(\mathbf{U}) - \mathbf{U}\mathbf{w}^T = \mathbf{F}'(\mathbf{U}') + \begin{pmatrix} 0 \\ \rho\mathbf{w}(\mathbf{v} - \mathbf{w})^T \\ \rho(\mathbf{v}\mathbf{w})(\mathbf{v} - \mathbf{w}) - \frac{\rho}{2}\mathbf{w}^2(\mathbf{v} - \mathbf{w}) + p\mathbf{w} \end{pmatrix}, \tag{15}$$

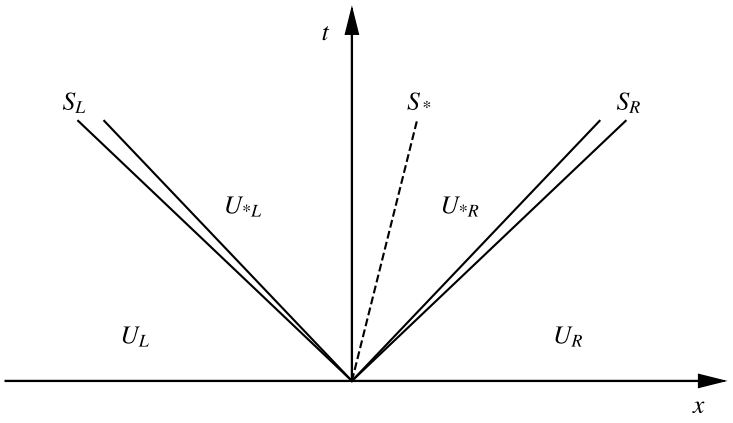

Figure 2. The wave structure of the Riemann problem. The vertical axis is time and the horizontal is the position. The three waves produce 4 distinct regions (U_L, U_{*L}, U_{*R} and U_R). The middle wave (dashed line) is called the contact discontinuity and has wave speed S_*. While the outer waves, S_L and S_R, can be any combination of a shock and rarefaction wave.

where the form of the terms are such that it makes equation 15 consistent with equation 13. This equation can be restated in terms of the rest frame fluxes,

$$\mathbf{F}_m(\mathbf{U}) = \mathbf{F}_s(\mathbf{U}) - \mathbf{U}\mathbf{w}^T = \begin{pmatrix} F'_\rho \\ F'_{\mathbf{v}} + \mathbf{w}F'^T_{\mathbf{v}} \\ \mathbf{w}F'_{\mathbf{v}} + \frac{1}{2}F'_\rho \mathbf{w}^2 \end{pmatrix}. \tag{16}$$

Thus, after solving the Riemann problem in the rest frame of the face the fluxes can be easily transformed back into the lab frame. Finally, to make use of equation 16 with equation 7, the primitive variables need to transformed by the following

$$\mathbf{W}' = \mathbf{W} - \begin{pmatrix} 0 \\ \mathbf{w} \\ 0 \end{pmatrix}. \tag{17}$$

This ensures that the correct values are used in the Riemann solver.

2.4.1. *Time Integration*

The time integration using equation 4 with equation 7 is a form of the MUSCL-Hancock scheme (Van Leer 1997; Toro 1997). For static meshes, the scheme is second order accurate in space and time. However, letting the mesh move introduces inaccuracies due to the neglect of mesh deformation during a time step Δt (Yalinewich et al. 2015; Steinberg et al. 2016). This can be corrected by adopting a Runge-Kutta type scheme, that uses information from the beginning and the end of a time step instead of mid point estimations. Specifically, we employ the method outlined by Pakmor et al. (2016) which updates the conservative variables by the following

$$\begin{aligned} \mathbf{W}'_i &= \mathbf{W}^n_i + \Delta t \frac{\partial \mathbf{W}}{\partial t}, \\ \mathbf{r}' &= \mathbf{r}^n + \Delta t \mathbf{w}^n, \\ \mathbf{Q}^{n+1}_i &= \mathbf{Q}^n_i - \frac{\Delta t}{2}\left(\sum_j A^n_{ij}\hat{\mathbf{F}}^n_{ij}(\mathbf{W}^n) + \sum_j A'_{ij}\hat{\mathbf{F}}'_{ij}(\mathbf{W}')\right), \\ \mathbf{r}^{n+1} &= \mathbf{r}'. \end{aligned} \tag{18}$$

Here we are taking an average of the fluxes from the beginning and the end of the time step. The flux at the beginning of the time step is constructed with the current state of the mesh with the primitive values extrapolated to the face.

Then the mesh generators move to their final position and the mesh is reconstructed. A new flux is constructed with new geometric quantities, however, the primitive values have been extrapolated in time from the beginning of the time step. At first glance, it seems that the mesh has to be constructed twice per time step. However, the generator velocity is assumed to be constant throughout the time step resulting in the final mesh to be equal to the beginning mesh of the next time step. Thus, the mesh need only to be constructed once per time step while the fluxes have to be calculated twice per time step. This method is not truly a Runge-Kutta scheme because of the time extrapolation but more a mixture of Runge-Kutta and MUSCL-Hancock scheme that has been shown to be second order accurate in space and time.

2.4.2. *Regularization*

Allowing the mesh generators to move with the local fluid velocity $\mathbf{w}_i$ can lead to cells that are elongated or mesh generators close to given face. This can lead to an unstable evolution of the cells because their faces can move rapidly relative to the generator velocity (Duffell & MacFadyen 2011). To counteract this issue Springel (2010) proposed a correction term that would steer the generator towards its center of mass. This effectively causes the cell to become rounder, thus mitigating the issue. The correction term is defined as

$$\mathbf{w}_i' = \mathbf{w}_i + \chi \begin{cases} 0, & \text{for } d_i/(\eta R_i) < 0.9 \\ c_i \frac{\mathbf{s}_i - \mathbf{r}_i}{d_i} \frac{d_i - 0.9\eta R_i}{0.2\eta R_i}, & \text{for } 0.9 \leq d_i/(\eta R_i) < 1.1 \\ c_i \frac{\mathbf{s}_i - \mathbf{r}_i}{d_i}, & \text{for } 1.1 \leq (d_i)/(\eta R_i). \end{cases} \tag{19}$$

Here R_i is the effective radius of the cell, $(V_i/\pi)^{1/2}$ for 2d and $(3V_i/4\pi)^{1/3}$ for 3d, d_i is the distance between the generators position $\mathbf{r}_i$ and center of mass $\mathbf{s}_i$, c_i is the local sound speed. Note that χ and η are tuning parameters which are typically set to 0.25 and 1.0, respectively.

2.5. *External Boundary Conditions*

For each simulation a boundary condition must be defined. At this moment, only reflecting and periodic boundaries are implemented. Our domains are restricted to to be rectangular with arbitrary aspect ratios (but this is not a fundamental constraint of the method). The implementation of both boundaries make use of ghost particles. These ghost particles are created during the mesh construction and carry all particle information to participate in the integration step. From the simulation perspective, they are treated as real particles, however, after a time step all ghost particles are discarded and new ghost particles are created with the relevant updated particle information.

2.5.1. *Periodic*

Periodic boundaries are formed by examining the circumcircle of each real particle in the tessellation. If this value intersects the domain boundary the particle is flagged for ghost construction. The flagged particle is then shifted periodically in all dimensions including corner cases and tested for boundary intersection. If intersection occurs a ghost particle is formed from the particle but with the appropriately shifted position data.

2.5.2. *Reflecting*

The reflection boundary parallels the periodic case with exception that particles are not periodically shifted. Instead, the flagged particle is mirrored across the minimum and maximum of each boundary dimension. If intersection occurs, again, a ghost particle is formed; however, the sign of the normal velocity component is flipped. This ensures that the mass flux vanishes on the surface of the boundary.

2.6. *Gravity*

In the presence of gravity the Euler equations 2 are modified by source terms:

$$\frac{\partial \mathbf{U}}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F} = \begin{pmatrix} 0 \\ -\rho \nabla \boldsymbol{\Phi} \\ -\rho \mathbf{v} \nabla \boldsymbol{\Phi} \end{pmatrix}. \tag{20}$$

Note that the gravitational potential $\boldsymbol{\Phi}$ only affects the momentum and energy. The source of the potential can be prescribed by an external source or by the self gravity of the gas. In the latter case the potential is given by Poisson's equation

$$\nabla^2 \boldsymbol{\Phi} = 4\pi G \rho \tag{21}$$

For the moment, we assume that $\boldsymbol{\Phi}$ is given. Then the equations 7 and 4 can be easily supplemented to include the gravitational force. First, the time derivatives in the reconstruction equation are replaced by

$$\frac{\partial \mathbf{W}}{\partial t} + \mathbf{A}(\mathbf{W}) \frac{\partial \mathbf{W}}{\partial \mathbf{r}} = \begin{pmatrix} 0 \\ -\nabla \boldsymbol{\Phi} \\ 0 \end{pmatrix}, \tag{22}$$

In this case the time extrapolated variables include a gravitational component. Second, the momentum and energy is updated during the flux update

$$\begin{aligned} \Delta \mathbf{p}_i &= \frac{\Delta t}{2}\left(\sum_j A^n_{ij}\hat{\mathbf{F}}^n_{ij,\mathbf{p}}(\mathbf{W}^n) + \sum_j A'_{ij}\hat{\mathbf{F}}'_{ij,\mathbf{p}}(\mathbf{W}')\right) - \frac{1}{2}\Delta t\left(m_i\nabla_i\mathbf{\Phi} + m'_i\nabla_i\mathbf{\Phi}'\right) \\ \Delta E_i &= \frac{\Delta t}{2}\left(\sum_j A^n_{ij}\hat{\mathbf{F}}^n_{ij,E}(\mathbf{W}^n) + \sum_j A'_{ij}\hat{\mathbf{F}}'_{ij,E}(\mathbf{W}')\right) - \frac{1}{2}\Delta t\left(m_i\mathbf{v}_i\nabla_i\mathbf{\Phi} + m'_i\mathbf{v}'_i\nabla_i\mathbf{\Phi}'\right). \end{aligned} \tag{23}$$

2.6.1. *Constant*

In the simplest case, the gravitational field can be defined as a constant external field. This assumes that the self gravity component of the gravitational field is negligible compared to the external field. In this case ρ can be set to zero and the solution to Equation 21 becomes trivial

$$-\nabla\mathbf{\Phi} = \mathbf{g}, \tag{24}$$

where $\mathbf{g}$ is a constant field. Once $\mathbf{g}$ is specified it can be coupled with the fluid equations through equations 22 and 23.

2.6.2. *Tree Gravity*

In the general case, self gravity should be taken into account. Because gravity is a long range force, the calculation of the gravitational forces can be particularly challenging. The root of the problem is the $N-1$ calculation per particle in the direct sum approach. Calculating the force for each particle produces a solver that scales as $\mathcal{O}(N^2)$ which is intractable. This scaling problem can be remedy by a Tree-based solver (Barnes & Hut (1986); hereafter BH). In this scenario the particles are broken into a hierarchy of groups, with their own multipole values, dictated by a recursive spatial subdivision scheme. The hierarchy, combined with a suitable criteria to transverse it, can be used to approximate the force calculation in $\log N$ operations per particle. Thus the $\mathcal{O}(N^2)$ scaling produced by the direct sum approach can be approximated by an operation that scales as $\mathcal{O}(N\log N)$.

The first part in constructing a Tree based solver is picking a grouping scheme. Our spatial decomposition scheme is based on the BH algorithm. In this scheme the domain is placed inside a cubic node, called the root. The root contains 8 daughter nodes in 3D (4 daughter nodes in 2D) that subdivides the space evenly. Particles are added to the root on a one by one basis. For each particle the root searches for the daughter node that spatially contains the particle. Once found, the daughter node will have three possible scenarios. First, if the node does not have any daughter nodes and does not store a particle then the particle is placed there and the node is termed a "leaf" node. Second, if the node already has a daughter node then the search continues recursively. Third, if the node already has a particle stored then the node is recursively subdivided until both particles exist in their own respective leaf nodes. Once all the particles are processed the multipole for each node can be calculated recursively.

With the tree constructed the force for a given particle can be approximated by "walking" the tree. This procedure is performed by starting at the root. Using a specified criteria a decision is made if including the node's multipole provides an accurate force calculation. If it does then the multipole value is used and the walk along this branch is terminated. If the multipole is not suitable then the node is "opened" and the procedure is repeated for each daughter node. The BH criteria to open a node is

$$\frac{l}{r} \geq \alpha, \tag{25}$$

where l is the side length of the node, r is the distance between the particle and the center mass of the node, and α is a free parameter constrained to $0 \leq \alpha \leq 1$. The error is effectively controlled by α because decreasing α produces walks with increasing lower level nodes. At the extreme case, setting $\alpha = 0$ opens every node and the computation becomes the original direct sum approach.

At this point, our description of the tree solver has been purely a serial implementation. In a parallel simulation the algorithm has to be slightly modified. We follow the approach of Springel (2005). In this approach each processor build its local BH Tree. Moreover, the local tree is augmented with "pseudo-particles" that serve as place holders for branches from different processors. Once the local multipole values are computed locally they are communicated across all processors. In doing so, each local tree can now construct the global "top-tree". The local tree is consistent in the sense that all internal nodes have the correct multipole. Now that the tree is consistent a tree walk can be performed. However, if particle opens a pseudo-particle it is flagged for exportation to the appropriate processor and placed in a communication buffer. Once the buffers are full the particles are exchanged and remote walks are performed. The particles are finally returned to their local processor with the correct force calculation.

2.7. *Parallelism*

2.7.1. *Partitioning*

For domain decomposition we use a Hilbert-Peano space filling curve (Springel 2005). A space filling curve maps 3D space onto a 1D curve. A Hilbert-Peano curve has the additional property that it preserves locality, meaning that points close along the 1D curve are generally close in 3D space. Due to the unique properties of the Hilbert-Peano curve a load balance scheme can be performed by the following. First, each particle is assigned a Hilbert-Peano key, generated by its 3D location. Second, the particles are sorted by their Hilbert-Peano key and third, the sorted particles

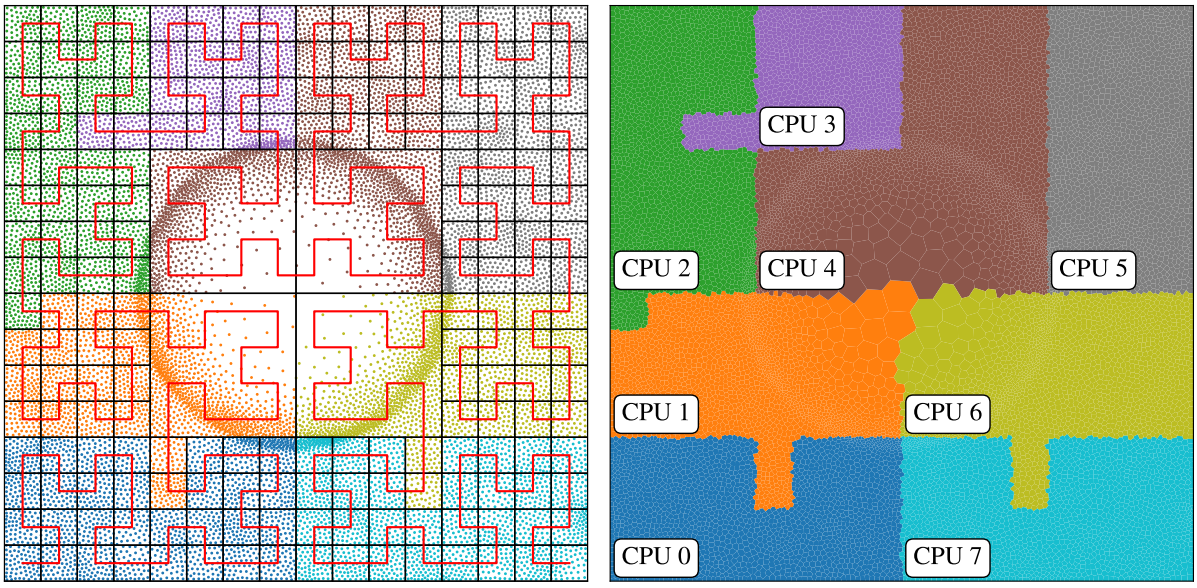

Figure 3. Load Balance performed on the Sedov-Taylor problem. Left: the tiling produced by the load balance and Hilbert-Peano key at level $n = 4$. Particle positions are plotted with color indicating the processor they reside on. Right: the resulting Voronoi tessellation comprised of local tessellations at each processor.

are partitioned across all processors with approximately constant work load. In doing so, each processor receives a set of particles defining a local compact domain.

In practice, the load balance procedure begins with the mapping of the particles into an integer grid of side length $2^n - 1$. The parameter n may be thought of as the resolution level of the domain, similar to AMR codes. For example, when $n = 1$ the domain is partitioned into $2^1 \times 2^1 = 4$ cells in 2D ($2^1 \times 2^1 \times 2^1 = 8$ cells in 3D). Likewise, if $n = 4$ the domain is partitioned into $2^4 \times 2^4 = 256$ cells. Therefore, every n corresponds to a level of refinement in powers of 2. Moreover, the total number of cells corresponds to the total number of possible Hilbert-Peano keys (2^{Dn}). At each value of n there is a corresponding Hilbert-Peano curve that intersects every cell. The relation between the grid and the Hilbert-Peano key is that the key encodes the information to traverse each level to the desired cell at a given level. The key can be created quickly through a series of bit shift operations and lookup arrays. We typically use values of $n = 20$ such that the key fits in a 64-bit integer with a dynamic range of $2^{20} \approx 10^6$ per dimension.

After the keys are computed they have to be sorted. However, performing a global parallel sort is not a trivial operation. Instead the keys are sorted locally and partitioned into small segments. The segments are then shared across all processors and are further refined by joining and splitting segments that overlap. The final result is a series of segments that can be used to partition the particles.

Geometrically, the segments create a tiling of the space, see Figure 3 Here we have performed the Sedov-Taylor simulation (see Section 4.2.5) on 8 processors. On the left hand side we plot the spatial tiling produced by the segments with particle positions colored by their residing processor. Further, we have plotted the the Hilbert-Peano keys (red line) at $n = 4$. Each processor receives a collection of tiles such that the work load is approximately constant. Clearly we can see the segment cuts that produce compact spaces of particles. On the right hand side, we show the resulting Voronoi mesh produced by this decomposition, labeled by processor number.

2.7.2. *Ghost particle copying*

For ghost particle we make a further distinction, interior and exterior ghost particles. Interior ghost particles are only used in parallel simulations while exterior ghost particles are associated with the boundary conditions of the simulation. In the following section we will go into the details on how the `DomainManager` creates ghost particles and transfers data.

3. CODE DESIGN

In this section, we will describe the design of the code for implementing the physical algorithms described above, in each case providing a rationale for the choices we made during the development of the code. Traditionally, codes in astrophysics have focused on the physical and mathematical character of the algorithms, with less focus on the particular implementation chosen. However, in this paper, we will stray from that path and instead spend much more time discussing our design philosophy and implementation choices. There are a number of reasons for this choice: first, it will provide a more coherent framework for those planning to use and adapt the code base; second, with the growing complexity of computational algorithms and the key role they increasingly play in scientific advances, it seems clear that the structure and design of a code such as this one should be placed under the same level of scrutiny as the algorithms themselves and therefore requires a more complete description than typically provided; finally, considerable

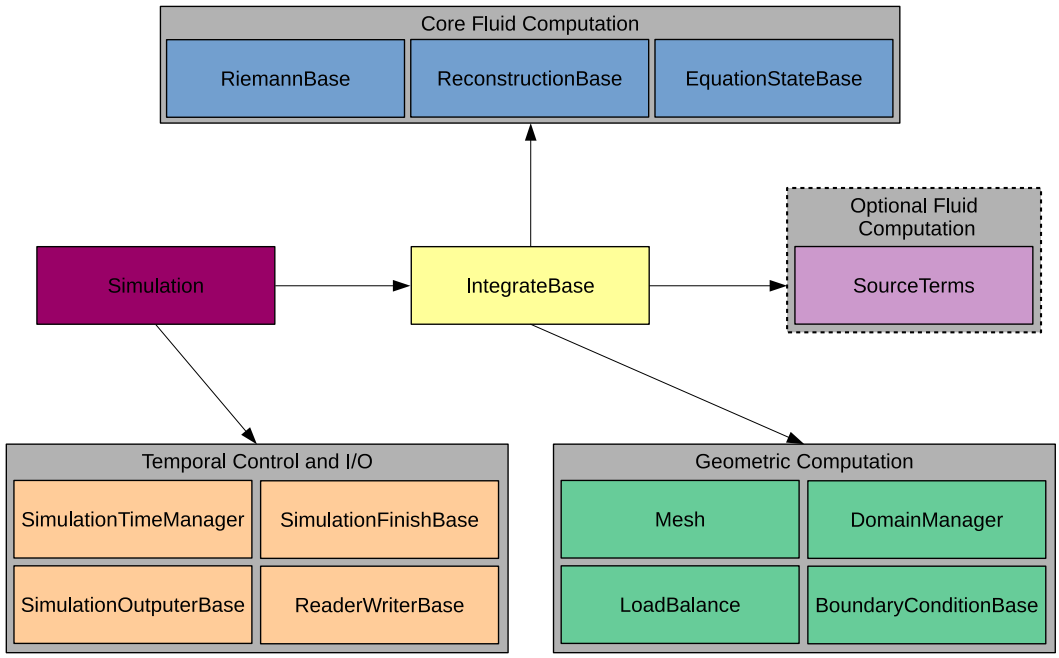

Figure 4. High level view of the the `phd` code. The code is compromised of six core components that have been color coded. Each component is comprised of one or more classes. Each class represent a designated computation. Classes that have "Base" in their name can be easily modified through inheritance. The arrows represent classes that can call other classes. The dashed box surrounding `SourceTerms` represents an optional component.

effort has been devoted to finding design choices that are efficient, compact, extensible and easy to understand and we believe these deserve to be described in some detail.

In several instances in this section, we will describe classes and methods in some detail to give concrete examples of the design philosophy in action. Moreover, we will give simple code fragments such that the reader can become more comfortable with the syntax. As a road map, Figure 4 shows a high level view of the code. As can be seen from this figure, the code consists of six core components: simulation, integration, temporal control, geometric computation, core fluid computation and optional fluid computation. This categorization is intended to provide a framework which is flexible enough to allow modifications without disrupting other portions of the code. In addition, each arrow represents a one-way association between components while each component can be composed of one or more subcomponents. As in example, `IntegrateBase` can control the actions of the classes in core fluid computation. In so, a new integrator can be devised that makes use of a different set and order of calls. In this way the code is modular, allowing to easily substitute different algorithms by swapping classes. In the following sections, we will go into detail of each component and subcomponent.

3.1. *Overall Class Design Considerations*

Before we go into specific details of the code, we outline some of the overarching traits that are common throughout the code. Specifically we will describe the following: field registration, inheritance and data structures.

3.1.1. *Field Registration*

Form our past experience, we have found that a challenging problem in maintaining codes is that of adding new fields. There are many reasons why we may need to add new fields, for example, the implementation of new physics module (MHD, radiation, chemistry etc.) or the addition of a passive scalar to track some portion of the fluid, or a modification of an existing algorithm that needs an auxiliary field which has not been previously stored. In any case, the addition of a new field can be challenging if the computation has been hard coded or the data structures, for example `C` structs do not allow for the addition of new fields. For this reason, we adopted a data structure that allows for the registration of fields (see also Section 3.3.2). In this manner fields can be easily added in the future. Registration provides a way to for a method to be aware of what fields are going to be used in a given simulation and ensuring that appropriate methods are applied to each field. In fact, there are two problems that need to be solved here: first, a new physics module may require that certain fields be followed (e.g. chemistry may require certain species fields to be defined on the grid); second, certain methods which operate on those quantities need to know what fields have been defined (e.g. the reconstruction method needs to know that it should reconstruct those species fields).

Based on these considerations, we have adopted a pair of common method which almost all top-level classes must implement:

- `register_fields()`: adds fields needed by the class to the simulation (in particular, to the particle container).

- `add_fields()`: inspects existing fields in the simulation (i.e. the particle container) and creates internal auxiliary fields needed for computation.

For registration, classes usually inspect the particle container, which holds the set of particles and their fields (see Section 3.3.2) and create the fields necessary for the simulation. As an example, gravity registers the acceleration fields in the proper dimension to the particle container. This way, we don't have unnecessary fields during the simulation (3D fields for 2D runs). Similarly, classes add fields internally by inspecting the particle container. For example, gravity inspects the particle container to internally register the correct dimension of the tree moments.

Since any number of fields can be registered, we have also allowed the ability to group fields by names. This has been extensively used to group collections of fields, for example, primitive and conservative fields (or all the components of a vector field). This allows the code to easily retrieve multiple fields that can be processed as a group.

3.1.2. *Class Dependencies*

Another source for future problems is when a class depends on another class. For example our gravity tree solver depends on our load balance implementation. If the algorithms are designed concurrently, then proper consideration is taken for their interaction. However, because of the extension of existing algorithms or the development of new ones, classes will eventually need to interact with others in a way that was not originally foreseen. In these situations common options are to re-factor the code or even rewrite the algorithm itself. Both methods can become cumbersome as the code develops and matures and changing one portion of the code may create a cascade of unwanted changes.

In our case we decided to write our classes as encapsulated computations (see Figure 4). This means we have tried to segregate our computations as independent tasks with interactions through an Application Programming Interface (API). For the most part, each computation has a base class (i.e `ReconstructionBase`, `RiemannBase`, `InetgrateBase` etc.) that defines the methods it can perform. Moreover, the base class only serves as a template and returns a warning if accessed without a properly inherited application. Thus, each implementation inherits the base class and must define the actual computation. Further, if a method depends on another class it has to be written in such a way that it confines to the API. Therefore, our algorithms are designed to conform to an API and any future development will have to abide by it. Our hope is that the design is sufficiently flexible that a wide variety of implementations can be described by it.

3.2. *Restarting the Code*

For restart capability, codes commonly use a parameter file approach. The file generally consists of several lines defining name-value pairs which is then read by a parser that reconstructs the simulation. Our approach will deviate, although, we do intend to create the parameter file approach. However, it will not be the primary method to restart a simulation.

Our new approach is inspired by `scikit-learn` (Pedregosa et al. 2011), a machine learning library for `Python`. In `scikit-learn`, many algorithms are comprised of one or more classes (i.e., scaling, imputation, dimension reduction, regression, ...). The collection of theses algorithms may have free parameters, known as hyperparameters, that cannot be derived from the training data. Instead a method like cross validation is used, which searches for the optimized values by training on multiple subsets of the data while varying every combination of hyperparmaters. For this reason, `scikit-learn` has the feature to recursively store all the parameters associated with the set of classes that make up the learning algorithm. In this way, once the optimized hyperparameters are found they can be returned and used to reconstruct the optimal learning algorithm. It is this functionality that we base our restart implementation.

Following this approach, during an output, the parameters and values of each class is saved into a dictionary. Additionally the name of the class is saved. At this point the dictionary represents a complete snapshot of all the classes and parameters used. The dictionary is then saved to disk, along with the particle data, as either a pickle[6] or json[7] file. To restart, a companion function will read the pickle/json file and query and set the parameters of each class. If the user wants to change a parameter or class it can be done by either editing the pickle/json file or manually overriding the value after the simulation has been reconstructed. This implementation is still in progress but should be available in the next revision.

3.3. *Particle Data Structures*

3.3.1. *Carray Class*

In choosing the underlying data structure several considerations were taken into account. First, the data has to be accessible in `Python` and `C` (or `Cython`). Second, the data structure has to accommodate several different data types. From these considerations, we wanted a data structure that closely resembles a `Numpy` array. `Numpy` arrays allocate raw data in `C` and allow the user to manipulate it in `Python` or `C`. With this design in mind, a decision had to be made on the underlying structure of the raw data. Two choices where considered: either the data would be held in `C` structs or arrays. The benefit of using structs is that it can encapsulate all the particle data. Thus, functions could operate on a particle by particle basis. Structs are also easily suited for passing and receiving data from other processors in parallel runs. Moreover, `Numpy` has an interface that treats arrays of structs as a record array. However, this form was abandoned early on as the attributes of the struct would have to be hard coded and therefore not allow the creation of dynamic fields at run time. With this consideration, the raw data was chosen to be `C` arrays.

[6] `https://docs.python.org/2/library/pickle.html`
[7] `https://www.json.org/`

The implementation we adopted was inspired by the `CarrayBase` class from the `pysph` code (Prabhu Ramachandran 2016). We have used this class as a starting point and have built functionality around it. This class can be initialized in `Python` or `Cython` and the interface closely resembles the `vector` template in `C++`. The `CarrayBase` comes in four different data types, `DoubleArray`, `IntArray`, `LongArray`, and `LongLongArray` with `CarrayBase` as the parent class for subtype polymorphism.

Like a `vector`, a `CarrayBase` allows for indexing and dynamic memory management. Internally the data is a contiguous `C` array and memory operations (i.e, append, resize, shrink, ...) are performed by `malloc` and `realloc`. Isolating the operations of memory management allows a clear path for future work to mitigate memory fragmentation and provides a direct way to assess the memory footprint of our data structures. Further, a `CarrayBase` can return a `Numpy` array, allowing the user to use all the `Numpy` functionality (i.e. slicing and fancy indexing). It is worth noting, that the `Numpy` array is not a copy but makes use of the array API by having a direct reference to the original data. Below is a simple example using a `DoubleArray`.

```
import phd

# allocate carray of size 10 doubles
# and assign values
x = phd.DoubleArray(10)
for i range(len(x)):
   x[i] = i**2

x.append(3.21) # append value at the end of carray
x.resize(5)   # resize to 5 doubles

xnp = x.get_npy_array()   # numpy array reference
xnp[:] = np.arange(x.length) # numpy slice
```

In this example a `DoubleArray` is created with 10 doubles, then assigned values by indexing. Notice that the length of the `DoubleArray` can be found by using the `len()` function or the `length` attribute. The `DoubleArray` then has a value appended to it followed by a resizing of length of 5. Finally, the `get_npy_array()` method is invoked returning a `Numpy` array to be used for slicing.

3.3.2. *CarrayContainer Class*

The use of the `CarrayBase` class allows us to easily manipulate contiguous data of a certain type. However, there are many circumstances that an algorithm needs to operate on several `CarrayBase`s collectively. For example, the Riemann solver needs all the primitive fields to estimate the fluxes. There is nothing wrong with having a function working on several `CarrayBase`s; however, it can become cumbersome to keep track of all `CarrayBase`s. Therefore, another data structure has been implemented that stores a collection of `CarrayBase`s of the same size. The data structure is called a `CarrayContainer` and like the `CarrayBase` it has many methods to manipulate the underlying data. Below are listed some of the most commonly used methods:

- `register_carray()`: register a new `CarrayBase` to the container.
- `get_carray_size()`: get the size of the `CarrayBase`s contained.
- `remove_items()`: remove elements at given indices from all `CarrayBase`s.
- `resize()`: resize each `CarrayBase` to a given size.
- `extract_items()`: return `CarrayContainer` with values from given indices.
- `append_container()`: append another `CarrayConatiner` to this one.

As can be seen from the list, most of the methods operate on the collection of `CarrayBase`s as a whole. The class only allows `CarrayBase`s of the same size and produces a runtime error otherwise. Additionally, each registered `CarrayBase` is associated with a string key such that it can be retrieved for later use. Below is an example of using the `CarrayContainer`.

```
import phd
import numpy as np

# create a container of 10 2d positions
carrys = {"x": "double", "y": "double"}
positions = phd.CarrayContainer(10, carrays)

# assign random values to each carray
size = positions.get_carray_size()
positions["x"][:] = np.random.rand(size)
```

```
positions["y"][:] = np.random.rand(size)

# add the z dimension
positions.register_carray(size, "z")
positions["z"][:] = np.random.rand(size)

# create 5 new random positions
carrays = {"x": "double", "y": "double", "z": "double"}
positions2 = phd.CarrayContainer(5, carrays)
size = ca_con.get_carray_size()
for ax in "xyz":
   positions2[ax][:] = np.random.rand(size)

# append new positions to old positions
positions.append_container(positions2)

# remove positions at selected indices
positions.remove(np.array([1, 3, 9])
assert(positions.get_carray_size() == 12)
```

In the above example, a `CarrayContainer` is created with two `DoubleArrays` labeled "x" and "y", of size 10 and stored in the `position` variable. Then each `DoubleArray` is retrieved by their key value and assigned random values. It is important to note that the `Numpy` array representation of `DoubleArray` was retrieved by using `Pythons __getitem__` special method. Thus, allowing to use `Numpys` slice feature. Next a new `DoubleArray` labeled "z" is registered to the container and assigned random values. Then a second `CarrayContainer` is created with random values and is append to `position`. Finally, vales at indices 1, 3 and 9 are removed from each `DoubleArray`.

3.4. *Simulation Class*

The `Simulation` class is the main driver for advancing the solution in time and coordinating outputs and logging information. The two most important methods of this class are listed below:

- `solve()`: advance `IntegrateBase` to its final state while outputting all necessary information.
- `compute_time_step()`: aggregate all time steps and enforce the smallest.

From its inception, the class was designed to be independent of the solvers. This was accomplished by viewing the simulation as a series of time advancements. Specifically, `IntegrateBase` can only perform a single time step from its given state (see Section 3.5 for details) while `Simulation` can dictate when and the number of time steps. Thus, `Simulation` controls the time advancement independently of the equations being solved. As of writing three integrators exist, however, adding a new integrator is relatively straightforward.

During the course of a simulation, the class is responsible for scheduling outputs and determining when the simulation has completed. Simulation outputs and termination are designated by the `SimulationOutputerBase` and `SimulationFinisherBase` classes respectively (see Section 3.6.1 for details). At the end of every time step the class calls `compute_time_step()` to modify the current time step and output any necessary data by cycling through all outputers. Likewise, at the beginning of the time step the class cycles through all `SimulationFinisherBase` classes in search of a termination signal.

Lastly, the simulation class also controls logging information (see Section 3.6.3 for details). Log information is currently printed to the terminal and saved to a log file. Additionally, the class allows the ability to chose the level of detail for logging. In parallel runs, the root processor takes responsibility for writing to the log file and (optionally) displaying to the terminal.

3.5. *Integrator Class*

To advance the the state of a fluid to a specific time, a suitable integration scheme must be provided. Although we have laid out the details of the MUSCL-Hancock scheme in Section 2.4.1, in practice, many schemes are available with different strengths and weaknesses. Therefore, we found it necessary to have an implementation that allows the ability to easily switch between integration schemes. To this end, we have created the `IntegrateBase` class which is responsible for advancing the state of the system by one time step. In this way, a simulation is viewed as a series of calls to `IntegrateBase` by the `Simulation` class. Note we say system, not the Euler equations. This distinction is made because the `IntegrateBase` is not limited to the Euler equations and, in principle, can be used to implement other equations. As an example, we have implemented a gravitational kick drift kick N-body solver (Springel 2005; Dehnen & Read 2011) under this framework. We find this framework to be versatile, allowing the user to chose from different schemes and allowing the ability to quickly create new schemes for experimentation. Currently we have implemented a static and moving mesh MUSCL-Hancock and a N-body integrator.

To implement an integrator, the `IntegrateBase` must be inherited. Below is the `IntegrateBase` API:

- `before_loop()` perform any needed initialization or initial computations.

- `compute_time_step()` compute from the current state, the maximum allowed time step.
- `evolve_time_step()` evolve the state of the system by one time step.
- `after_loop()` perform any clean up or needed final computations.

For each integrator implementation one must define each method from the API. The integrator has references to all the computation classes as well as state attributes (iteration counter, time step, and time). The most involved method is `evolve_time_step()` which defines the equations and method to be used. For clarity, we show the implementation of the static mesh MUSCL-Hancock integrator below.

```
def evolve_timestep(self):
    """Solve the compressible gas equations."""

    phdLogger.info("StaticMeshMUSCLHancock: Starting integration")

    # build left/right states at each face in the mesh
    self.reconstruction.compute_gradients(self.particles, self.mesh,
           self.domain_manager)
    self.reconstruction.compute_states(self.particles, self.mesh,
           self.equation_state.get_gamma(), self.domain_manager, 0.5*self.dt,
           self.riemann.boost)
    self.compute_source("primitive")

    # solve riemann problem, generate flux
    self.riemann.compute_fluxes(self.particles, self.mesh, self.reconstruction,
           self.equation_state)

    # update conserved quantities from fluxes
    self.mesh.update_from_fluxes(self.particles, self.riemann, self.dt)
    self.compute_source("flux")
    self.compute_source("compute")

    self.compute_source("conservative")
    self.domain_manager.update_ghost_fields(self.particles,
           self.particles.carray_named_groups["conservative"],
           True)

    # convert updated conservative to primitive
    self.equation_state.primitive_from_conservative(self.particles)
    self.iteration += 1; self.time += self.dt
```

As it can be seen the method is a series of core computations. The method begins by calculating the gradients followed by computing the left/right states for the riemann solver. Once the fluxes are calculated they are used to update the fields. Sprinkled in are calls to source terms (see Section 3.10.1), if any. Source terms are stored in a dictionary and each call to `compure_source` cycles through each source term registered and calls the respective computation signaled by the string argument. Finally the ghost particles, iteration counter and time are updated and the system is ready for the next computation.

3.6. *Temporal Control and I/O*

In the following sections we will go over the implementations of outputting data (`ReadWriterBase`), specifying when an output should occur (`SimulationOutputBase`), and completion of the simulation (`SimulationFinisherBase`). All these classes are controlled by a `SimulationTimeMangager` class which consists of any number of registered `SimulationOutputerBase`s and `SimulationFinisherBase`s. In this manner, any number number of conditions can be applied for signaling an output or the completion of the simulation.

3.6.1. *Outputer and Finisher Class*

The code has been designed to handle arbitrary criteria for determining when to output data (of various types) as when the simulation completes. This has been achieved by introducing the `SimulationOutputerBase` class for output and the `SimulationFinisherBase` class for completion. Each outputer and finisher instance must be registered to the `SimulationTimeManager`. The `SimulationTimeManager` can hold any number of them and is responsible for maintaining consistency across calls (in particular, ensuring timesteps taken do not violate and output or finishing criteria. For example, both `SimulationOutpuerBase` and `SimulationFinisherBase` have methods which return a maximum allowed timestep such that their respective times are not missed. Thus, it is the responsibility of the `SimulationTimeManager` to aggregate all time steps and enforce the smallest one such that no condition is overlooked. At the end of every time step (`SimulationOutputerBase` includes the beginning and ending of the simulation) the `SimulationTimeManager` calls all registered `SimulationOutputerBase`s and `SimulationFinisherBase`s.

During each call the class receives complete access of the integrator and associated data, including the time, time step, iteration, particle data, flux data, reconstruction data, etc. This allows the output and finisher methods to depend on features of the solution in potentially complicated and arbitrary ways. Such an implementation forgoes the hard coded approach of multiple `if else` statements and allows the flexibility to implement as many conditions as needed.

As an example of how to use this framework, we show how to implement an outputer and finisher to the Sedov problem from Section 4.2.5. We wish to output all time steps once the shock has reached a density value grater then $\rho = 2$. Also we want the simulation to complete once the shock reaches a value greater than or equal to $\rho = 3$. Of course, in this situation one may use the analytical solution to extrapolate the time when such values would occur. However, this assumes that the shock is tracked exactly. In our scenario we let the simulation do the work for us.

First we begin by defining an outputer, called `OutputSedovDensity`, with the objective to output all data once the density value of $\rho = 2$ has been reached (see code excerpt below). This is accomplished by inheriting `SimulationOutputerBase` and modifying `check_for_output()` and `modify_timestep()`. We introduce a new constructor (`__init__`) that extends the parents constructor by adding the new parameter `density_output`. The parameters `base_name`, `counter`, and `pad` are the parents parameters to generate a unique name and folder for each output. For overwriting `check_for_output()`, we first check if the call is made in the main loop and second if the maximum density is greater than `density_out`). In this case, we do not change the time step, therefore it is left unaltered.

```
...
class OutputSedovDensity(SimulationOutputerBase):
    def __init__(self, density_out=2, base_name="density_output",
                 counter=0, pad=4):
        super(OutputSedovDensity, self).__init__(base_name, counter, pad)
        self.density_out = density_out

    def check_for_output(self, simulation):
        """Return True to signal the simulation has reached
        sedov interval to ouput data."""
        integrator = simulation.integrator
        state = simulation._state == SimulationTAGS.MAIN_LOOP
        output_sedov = integrator.particles["density"].max() > self.density_out:

        if state and output_sedov:
            return True
        return False

    def modify_timestep(self, simulation):
        """Return consistent time step."""
        # not modifying
        return simulation.integrator.dt
```

We also need to add a criteria to signal the completion of the simulation. To do this we will create a finisher class. This class will signal completion when the simulation density is greater then or equal to $\rho = 3$. We will call it `SedovDensityFinisher`, and create it by inheriting `SimulationFinisherBase` (see code excerpt below). The `finished()` method must be overwritten. In our case we need only to compare the max density with $\rho = 3$.

```
...
class SedovDensityFinisher(SimulationFinisherBase):
    def __init__(self, density_stop=3):
      super(SimulationFinisherBase, self).__init__()
        self.density_stop = density_stop

    def finished(self, simulation):
        """Return True to signal the simulation is finished
        if reached max sedov density.
        """
        if simulation.integrator.particles["density"].max() >= self.density_stop:
            return True
        else:
            return False
```

With the creation of these two classes our simulation will output data at every time step when the density value is greater then $\rho = 2$ and terminate when the density is greater then or equal to $\rho = 3$. Although our example is simplistic, we hope that it shows the flexibility and power of what the outputers and finishers may achieve.

3.6.2. *Reader/Writer Class*

In the previous section, we mentioned that `SimulationOutputBase` is used to signal when an ouput is requested. However, we did not mention the details of how an output is produced. To allow the code to output in a variety of different data formats and make various choices about what data is actually recorded, we have implemented a `ReaderWriterBase` class. This allows the user the freedom to output the data to any desired format. Likewise, this also allows the data to be read in any format. Every instance of `ReaderWriterBase` is associated with a `SimulationOuputerBase` (see Section 3.6.1) which signals when output should be created. In this way, multiple outputs can be created at various moments during the simulation. The standard API of the `ReaderWriterBase` is listed below:

- `read()` read data in a specific format.
- `write()` write data in a specific format.

With such a framework, new formats can be easily implemented. Moreover, this allows a simple way to create a front end to read data from other codes. As of writing, we have implemented a class to read and write particle data in `hdf5`[8] format. We are currently in the process of adding new data formats.

Although this framework can be used to save all or some of the data in different formats, it can be used for a much wider set of tasks. Since the `ReaderWriterBase` has access to the integrator and its associated data, any component of the simulation can be considered for output. Also, output does not necessarily have to be particle data saved to disk. This framework can easily be used to create plots, send memory diagnosis through an email, or even transfer the output to a remote host.

As a simple example, we show below how one can easily track the energy of the system under this framework.

```
...
class EnergyTracker(ReaderWriterBase):
   def write(self, base_name, output_directory, integrator):
         particles = integrator.particles

         # compute square velocity
        v2 = particles["velocity-x"]**2 + particles["velocity-y"]**2
        if len(particles.carray_named_groups["position"]) == 3:
         v2 += particles["velocity-z"]**2

        # calculate kinetic and potential energy of system
        kinetic = 0.5*(particles["mass"]*v2).sum()
        potential = 0.5*(particles["mass"]*particles["potential"]).sum()

        message = "Energy values: Kinetic %.2E Gravity %.2E" % (kinetic, potential)
        phdLogger.info(message)
```

Our system of interest is a fluid simulation which includes gravity. First, a new class called `EnergyTracker` is created by inheriting `ReaderWriterBase` with the `write()` method overwritten. The new method simply calculates the total kinetic and potential energy of the system and then outputs the information to the logger. We don't implement a `read` method because it would not be consistent with our `write` method. If the `read` method is accidentally called, it wall call the parent implementation and raise an error. Finally, the class is registered to the appropriate `SimulationOutputerBase` and the calculation will be performed at the appropriate times.

Although the above example is in serial, it can be extended into parallel. Importing `phd` allows access to variables `_has_mpi`, `_in_parallal`, `_size`, `_rank`, and `_comm`. These variables are assigned at runtime and allow to check if MPI is available (`_has_mpi`), if the current simulation is running in parallel (`_in_parallel`) and the ability to perform MPI communications (`_comm`). Our `hdf5` output implementation produces serial or parallel outputs[9] using these variables. In this way, extending a serial implementation to parallel only requires the appropriate checks and placement of MPI calls (see the following the section for a parallel example).

3.6.3. *Logging*

Event logging during a simulation is performed through the python `logging`[10] library. The library has many capabilities but for our intentions we have focused on displaying and storing messages related to the state of the simulation. To that end, we use one logger called `phdLogger`. This logger can be imported to any file and used to log any information of interest.

For our purposes, the logger has four levels of logging. They are listed below:

- debug: detailed information or diagnosing.
- info: working as expected.
- success: a successful completion.

[8] `https://www.hdfgroup.org/solutions/hdf5/`

[9] In parallel each processors outputs it is own data to a separate folder. In the future we will add the ability to produce parallel outputs, meaning, aggregate the data from all processors and store it into one main file.

[10] `https://docs.python.org/2/library/logging.html`

- warning: unexpected result that may lead to future problem.

The log levels have been listed in order of inclusion. Meaning if "warning" is chosen, then all messages types are logged. However if "info" is chosen only messages of type "debug" and "info" are logged. At runtime the log level and file to store the messages can be specified through the `Simulation` class.

As an example we show a simple usage of the logger below:

```
import logging
phdLogger = logging.getLogger("phd")
...
phdLogger.info("Starting kinetic energy calculation")

ke = 0.5*particles["mass"]*(particles["velocity-x"]**2 + particles["velocity-y"]**2)
if dim == 3:
   ke += 0.5*particles["mass"]*particles["velocity-z"]**2

kinetic_energy = ke.sum()
if phd._in_parallel:
   phd._comm.Allreduce(kinetic_energy, glb_kinetic_engery, op=MPI.SUM)
else:
   glb_kinetic_energy = kinetic_energy

if glb_kinetic_energy > 0:
   phdLogger.debug("Kinetic energy greater than zero")
```

In this example we import the logger and print out a messages related to the calculation of the kinetic energy. It is important to note that this example is true if run in serial or parallel. The logger has been modified such that logging information and storage is always handled by the root processor.

3.7. *Geometric Computation*

In the following sections we will go over the classes associated with geometric properties of the code. These are the creation of the mesh (`Mesh`), boundary conditions and ghost particles (`BoundaryConditionBase`), domain information (`DomainManager`), and spatial decomposition (`LoadBalalnce`) for parallel simulations.

3.7.1. *Mesh Class*

The `Mesh` class is responsible for all geometric computations relating to the particle cell. For example the class computes each cell's volume and center of mass. The class works in tandem with the `DomainManager` to build the mesh. Internally the `Mesh` has a tessellation class. The tessellation class is wrapper for any scheme to produce a Voronoi diagram (or other tessellation). For our implementation, we follow FVMHD3D code (Gaburov et al. 2012) and use the CGAL[11] library (Fabri & Pion 2009) for Voronoi calculations.

The tessellation algorithm from Section 2.2 is performed through the `build_geometry()` method. It works with the `DomainManger`, building the mesh in rounds. In the first round, all the local particles are added to the tessellation. At this point there are no ghost particles. Particles that have infinite volume or a radius that intersects other processor boundaries (in parallel) or the simulation domain are flagged. These flagged particles are inspected for the creation of ghost particles. The mesh is then updated with new ghost particles and the process is repeated. This procedure continues until all particles have a finite volume and all neighbors have been accounted for.

When the tessellation is complete, cell values are computed. This involves calculating cell neighbors, volumes, center of masses, face areas, etc. The information regarding the faces is stored in an attribute named `faces`. For each face entry there exists two references to the particles that define the face. Likewise, each particle has references to all the faces that make up the particles cell. In this way, when a particle wants all of its neighbors, it first queries all of its faces and then uses each face to retrieve the second particle that makes up the face.

After the geometric quantities are calculated, they can be used for further computations. A computation of interest for the `Mesh` is the flux update (that is, updating conserved quantities based on the fluxes). It is somewhat arbitrary what class carries out this operations; however, we have elected to have the `Mesh` class perform this function. We argue that this is the natural choice particulary when we add meshes with symmetry (spherical and cylindrical). This can be added in straight forward fashion by adding the geometrical terms in the flux update. Thus, to add such meshes we only need modify the mesh class and constrain the particle movement.

3.7.2. *Domain Manager*

We consider a domain to be the spatial region where the computation is performed. In serial, this is the entire spatial region of the simulation. In parallel, this is the spatial region associated with each processor. In this way, for the most part, our domains are isolated computations and when data is needed from neighboring domains they are requested through the `DomainManager`. Below is some of the most frequent used methods. Our `DomainManager` class is designed to handle information associated with the domains of the simulation. Below is listed some of the main methods:

[11] `www.cgal.org`

- `create_ghost_particles`: create appropriate ghost particles (including exterior in parallel simulations) dictated by the boundary condition.
- `move_particles`: after conservative update move mesh generators.
- `update_ghost_fields`: transfer particle data from image particle (particle used to create ghost particle) to ghost particle.
- `update_ghost_gradients`: transfer particle data from image particle to ghost particle.

The `DomainManager` was initially designed to contain the limits and dimension of the problem. As development continued it became natural that the `DomainManager` would encompass the boundary condition, ghost particle information, and exchange of data across the boundaries. In its current state, the `DomainManager` only supports communication of ghost particle data and does not support the general forms of data communication (i.e reduction, gathers, broadcast etc). We plan in the next revision to implement an API that performs these tasks. In doing so, we would remove dependence on MPI calls in the rest of the code and all parallel communication of any sort would be done through the `DomainManager`.

In parallel, particles are decomposed into a disjoint set of spatial domains with each domain mapped to a unique processor. The construction of the global tessellation (e.g. Voronoi mesh) is then delegated to the construction of a disjoint set of local meshes. For the set of local constructions to be consistent with the global mesh, the appropriate boundary particles must be communicated across domains. The particles used to stitch together the disjoint meshes are called interior ghost particles. Their creation is handled by the `DomainManager` through the `create_interior_ghost_particles()` method. This method creates interior ghost particles by inspecting the search radius of each local particle. If a particle's search radius overlaps a processor boundary then it is flagged and a corresponding ghost particle is created and exported to that processor. The search method is made possible by the `DomainManager`'s ability to query all domains through geometrical searches. Care is taken such that no duplicates are created through the whole process. Note that rather than querying another processor for boundary information, patches determine which particles have a face on the other processor(s) and then send that particle (see Section 2.2 for more details and reference therein). Ghost particles created in this way ("owner sends") require only one-way communication and therefore should generally be more efficient.

After the creation of ghost particles, supplemental data has to be communicated. For example, the center of mass of a ghost particle cannot be computed because our implementation only guarantees that local particles have all of their neighbors. Certainly we could of opted to import all neighbors of ghost particles but instead we have decided to communicate that information instead. Thus, the `DomianManager` records all information associated with a ghost particle. This allows ghost particles to be easily updated with any data. These operations are implemented through `update_ghost_fields()` and `update_ghost_gradients()`.

The `DomainManager` is also responsible for movement of particles from one processor to another (for moving mesh integrators). After a flux update, the particles are moved. Depending on initial particle position and velocity, a particle may leave its processor or the simulation domain. In either case, the `DomainManager` is responsible for the destination of the particle. When a particle departs the simulation domain, in serial or parallel, the `DomainManager` flags that particle and then applies the appropriate boundary condition. For reflective boundaries, particles are checked for the possibility of leaving the domain. For such a case, the particles mesh generator velocity is set to zero, resulting the particle to stay in the domain. For periodic boundaries, particles that leave the domain are wrapped periodically back. If the simulation is in parallel, the `DomainManager` exports the wrapped particle to the correct processor. Likewise, if a particle leaves its processor patch the `DomainManager` will query the domains and find the respective domain for export.

3.7.3. *Boundary Conditions*

Similarly to interior ghost particles (see Section3.7.2), exterior ghost particles are used to complete local meshes. However, exterior ghost particles are not from neighboring domains but instead are created through a specified external boundary condition. To allow for different boundary conditions we have created the `BoundaryConditionBase` class that interacts with the `DomainManager`. The API is listed below:

- `create_ghost_particle()` create ghost particle from a flagged particle;
- `migrate_particles()` apply appropriate boundary conditions for particles that have left the domain;
- `update_gradients()` apply boundary condition to ghost particle gradients;
- `update_fields()` apply boundary condition to ghost particle fields.

Through this API, any boundary condition may be implemented. The boundary condition does not have to be uniform in each dimension. Mixed boundaries or even problem specific boundaries are allowed in this framework. Furthermore, the API allows the boundary condition to modify particle motion and field data. In this way, we have extracted all the boundary information from the `DomainManager`.

Currently, we have two implementations of external boundary conditions: reflective and periodic. In the reflective case, a flagged particle (i.e. one with a face that is overlaps an exterior domain) is mirrored across the boundary edge

and the velocity direction normal to the boundary surface is reversed in sign. In parallel, the ghost particle is then further inspected for intersection of neighboring processors. The ghost particle is then placed in a communication buffer to be exported to each flagged processor. For the periodic case, the procedure is similar, with the exception that the particle is periodically shifted instead of being mirrored.

3.7.4. *Load Balance*

Our load balance scheme, described in Section 2.7, is implemented through the `LoadBalance` class. The method `distribute()` performs the redistribution of particles across processors in a parallel run. The `LoadBalance` class is essentially an API. Internally, it has a reference to a `Tree` class that performs all the underling operations. This class was one of the early algorithms implemented before the core ideas of the code class structure were in place. Therefore, this class will be heavily modified to make it consistent with the rest of the code in the next revision. Points of modification will be: first, an API which is independent of the internal implementation and, second, we want to generalize the load balance scheme to allow more mature third party packages. As an example of a third party package is the `Zoltan` library (Devine et al. 2000) which offers geometric, graph- based, hypergraph-based partioning algorithms for load balancing. Additionally `Zoltan` has a `Python` front end[12], allowing it to be easily implemented in `Python` or `Cython`.

3.8. *Core Fluid Computation*

We will now detail the core fluid computations. This encompass the reconstruction method (`ReconstructionBase`), the Riemann solver (`RiemannBase`), and the equation of state (`EquationOfStateBase`). We believe this decomposition of classes allows the ability to easily implement a broad range of new algorithms. In our case we have implemented two reconstruction and three Riemann methods. Moreover, even though our current state of the code is specific to the moving mesh method, we believe these classes, along with the `IntgrateBase` (see Section 3.5), will permit the implementation of other fluid based methods such as SPH (Gingold & Monaghan 1977), mesh free (Hopkins 2015), and discontinuous Galerkin (Mocz et al. 2014) methods.

3.8.1. *Riemann Solver*

The fluxes at each face of the mesh are constructed by the `RiemannBase` class. As a common thread with all of our computation classes, the base class defines the API. The API is straight forward with only two methods to define:

- `riemann_solver()` solves for fluxes at each interface.
- `compute_time_step()` computes the maximum allowed time step.

The `compute_time_step()` is optional since the base class has a default CFL constrained time step calculation. However the option is there if a particular solver implementation needs to modify it. Furthermore, the base class defines a method for the fluxes to be transformed back into the lab frame. This allows, for the most part, us to implement a solver without worrying about the details of rotating and boosting back into the lab frame. As of the time of writing, we have implemented the HLL, HLLC, and Exact Riemann solvers.

3.8.2. *Reconstruction*

Once the mesh is created, cell-averaged values need to be reconstructed to face-centered values in order to calculate the fluxes across the face. The interface to implement the reconstruction scheme is dictated by the `ReconstructionBase` class which has the following API:

- `compute_gradients()` calculate the gradients of the fields for each particle.
- `add_spatial()` add gradients to expansion.
- `add_temporal()` add time derivatives to expansion.

The API embodies a general deconstruction of Equation 7 into three building blocks. First, is the calculation of the spatial derivatives (i.e gradients) by `compute_gradients()`. In this call the derivatives are computed through a specified scheme. For example, the gradients can be calculated by the least squares approach (Pakmor et al. 2016) or through geometrical properties (Springel 2010). With the derivatives calculated, cell-centered values can then be extrapolated spatially (`add_spatial()`) or temporally (`add_temporal()`) by Equation 8. In this manner the user has complete control of the terms in the expansion. This is important to note, as many integrator schemes make use of the reconstruction several times, for example, the Runge-Kutta solver variants (i.e Pakmor et al. (2016) and Duffell & MacFadyen (2011)).

[12] `http://pysph.readthedocs.io/en/latest/pyzoltan/overview.html`

3.9. *Equation of State*

Due to the implementation of the reconstruction and Riemann solvers, primitive fields must be computed. The computation of primitive fields requires an equation of state. Since there are multiple formulations of an equation of state we have decided to implement an `EquationStateBase` class with the following API:

- `conservative_from_primitive()`: compute conservative fields from primitive fields.
- `primitive_from_conservative()`: compute primitive fields from conservative fields.
- `sound_speed()`: compute the sound speed of the fluid.
- `get_gamma()`: compute the ratio of specific heats.

The `EquationStateBase` is responsible for converting the fields from conservative to primitive and vice versa. Furthermore, the equation of state can calculate the sound speed of the fluid. We have found this formulation to adequately remove the details of the state of the fluid from other calculations. Although our `EquationStateBase` is simplistic, we plan to extend its functionality to include chemistry species in the next revision to allow the use of a chemistry solver.

3.10. *Optional Fluid Computation*

3.10.1. *General Source Terms*

The inclusion of source terms used to model additional physical effects has been implemented through an API class. Examples of source terms are gravity, chemistry and radiation. In general, the API is associated with a specific integrator. Although this creates more work for a given source term, we have found this to be a better solution then continual refactoring a preexisting source term such that can be molded to be consistent with several integrators. Furthermore, having an API allows us to easily implement third party packages as source terms since the API is merely a wrapper to the main computation.

The current API formulation was chosen after considerable experimentation. One of the earlier attempts was a registration process. In this scenario a source term would link any computation to a class method. This information was stored in a dictionary inside `Simulation` and at runtime each class method was over written using `Python`s decorator scheme. This allowed source terms to "hook" its computation with other calculations. Although, this scheme makes use of more advance programming methods we found that this implementation was not transparent and made debugging difficult. Instead we found the current implementation to be more understandable and easier to generalize.

The API for the MUSCL-Hancock integrator is listed below:

- `apply_primitive()`: modify primitive variables.
- `apply_conservative()`: modify conservative variables.
- `compute_source()`: calculate source components.
- `compute_time_step()`: calculate time step from source term.

The API is inspired by Equations 2 and 8. Including a source term to the fluid equations results to the right hand side of Equation 2 no longer being zero. Moreover, since equation 2 is the basis of the conservative update the conservative variables have to be augmented from the source term (`apply_conservative()`). Similarly, the right hand side of Equation 8 is no longer zero and the expansion must also have a component from the source term (`apply_primitive()`). The two remaining methods are for calculating the source term and modifying the time step. Once the methods have been defined the class is registered to `Simulation` which can hold any number of source terms.

3.10.2. *Self Gravity Example*

For a more concrete example, we show how self gravity is implemented as a source term in the MUSCL-Hancock scheme. Our self gravity is a tree based implementation named `GravityTree`. Its main routine is `walk()` which is the calculation of gravitational accelerations from the current position of the particles. We created a new class called `SelfGravity` which inherited `MUSCLHancockSourceTerm`.

```
...
cdef class SelfGravity(MUSCLHancockSourceTerm):
    ...
    cpdef apply_primitive(self, object integrator):
      ...
        # loop over each face in the mesh
        for m in range(integrator.mesh.faces.get_carray_size()):
            ....
            # add gravity to velocity
            for k in range(dim):
                vl[k][m] += 0.5*dt*a[k][i]
```

```
            vr[k][m] += 0.5*dt*a[k][j]

    # add gravity acceleration from particle
    for i in range(integrator.particles.get_carray_size()):
        ...
            for k in range(dim):
                # update momentum
                e.data[i] += 0.5*dt*mv[k][i]*a[k][i]

                # update energy
                mv[k][i] += 0.5*dt*mass.data[i]*a[k][i]

cpdef compute_source(self, object integrator):
  ...
    phdLogger.info("SelfGravity: Calculating accelerations")
    self.gravity.build_tree(integrator.particles)
    self.gravity.walk(integrator.particles)
    ...

cpdef apply_conservative(self, object integrator):
    ...
    # add gravity acceleration from particle
    for i in range(integrator.particles.get_carray_size()):
        ...
            for k in range(dim):
                # update momentum
                mv[k][i] += 0.5*dt*mass.data[i]*a[k][i]

                # update energy
                e.data[i] += 0.5*dt*mv[k][i]*a[k][i]
```

From Section 2.6 the primitive and conservative variables are modified by gravity (see Equation 23). In the case of the primitive variables, we applied a velocity half update in `apply_primitive()`. For simplicity, we also added the initial half conservative update to bypass the need to store auxiliary values. In similar fashion, we added the post half conservative update in `apply_conservative()`. The actual computation of the gravity is performed in `compute_source()`. With these simple calls we have implemented a self gravity scheme in this framework.

3.11. *Units*

Astronomy deals with large scales and multi-physics which results to an overabundance of different units and scales. Moreover, observations may be summarized in units that are convenient or a product of instrumentation. This myriad use of scales must be placed in a coherent framework to produce consistent and correct results from a simulation. Commonly, the user must perform a unit analysis, thereby, defining unit conversions to place all variables in a simulation to the appropriate unit and scale. This approach, can lead to errors if the conversions are not carried correctly or if there are assumed units or conversions hard-coded in the source code. Therefore, we have decided to implement an unit based system in `phd` to allow the user to assign variables in any standard system and allow the software to perform all the internal scaling and conversions.

As of yet, the unit system has not been added as we are in the progress of designing and developing it. Our approach, is to follow the unit system implemented in `yt`[13] (Turk et al. 2011) which is an open source visualization and analysis package for simulated data. In their framework, they have sub-classed the `Numpy` array, as well other data structures, to carry units. This allows for all standard arithmetic operations to be performed while applying the appropriate conversion. However, as of writing the `yt` units system has become a standalone library named `unyt`[14] (Goldbaum et al. 2018). Thus we have abandoned our approach and have elected to use `unyt` as our unit-based system. We plan to incorporate `unyt` in our next release.

4. TESTS

4.1. *Unit Tests*

Many of our classes have unit tests that can be performed through the `nose`[15] library. We are currently implementing more test since due to the rapid development of the code many test have become obsolete.

4.2. *Hydro Tests*

In this section we present a series of tests to help verify the integrity and stability of the hydrodynamic algorithms used. These are so-called *answer* tests, for which we impose a specific set of initial and boundary conditions that have (generally) exact analytic solutions again which we compare. They test the end-to-end performance of the code.

[13] `https://yt-project.org/`
[14] `http://unyt.readthedocs.io/en/latest/`
[15] `http://nose.readthedocs.io/en/latest/`

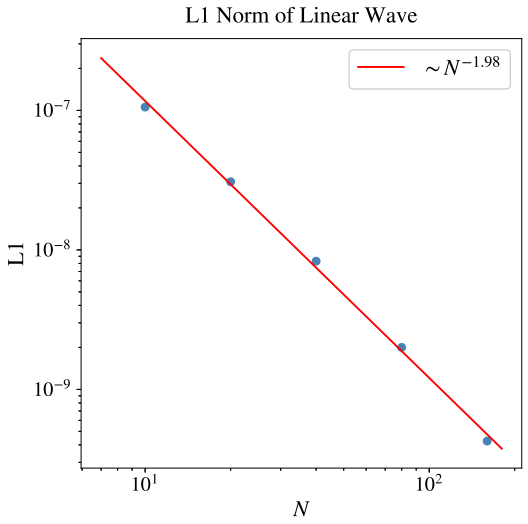

Figure 5. L1 norm of linear wave problem in two dimensions. Blue points are the results of simulations with different resolutions overlaid by a linear fit showing the convergence is approximately second order. This example was produced by **linear_wave_2d.py** and **l1_norm.py** scripts.

For each test it is assumed that linear reconstruction (Section 2.3.1), mesh movement (Section 2.4), motion correction (Section 2.4.2), and HLLC algorithm (Section 2.3.2) were used, unless stated otherwise. We have found the HLLC approximate Riemann solver, in most cases, to be nearly as accurate as the Exact solver and to produce solutions which are nearly indistinguishable. On the other hand, the HLL solver produces significantly lower fidelity answers and hence the HLLC solver is the natural choice as it is much less computationally intensive than the Exact solver. All scripts to generate the simulations and plots can be found in the **test_suite** directory.

4.2.1. *Linear Wave*

Sound waves provide a key mechanism to transport information through a fluid. An elementary test problem is the ability to maintain accurate wave propagation of small disturbances, both in terms of their amplitude and phase. Given a fluid in equilibrium with constant density ρ_0, pressure P_0 and zero velocity $\mathbf{v} = 0$ with perturbations of the form

$$\begin{aligned} \rho &= \rho_0 + \delta\rho(x,t) \\ P &= P_0 + \delta p(x,t) \\ \mathbf{v} &= \delta\mathbf{v}(x,t), \end{aligned} \tag{26}$$

and maintaining terms to first order in the Euler equations produce the wave equation for each variable with sound speed equal to the fluid's sound speed c_s. Thus, we can generate perturbations that should propagate with a finite velocity and maintaining its form as along as the initial disturbances are relatively small.

We set up a two-dimensional box of unit length with constant $\rho_0 = 1.0$, $P_0 = 3/5$, $\mathbf{v} = 0$, and $\gamma = 5/3$ with periodic boundary conditions. A sinusoidal wave in the x direction of the form $\delta\rho(x,t) = A\sin(kx + wt)$ with $k = w = 2\pi$ and $A = 10^{-6}$ is added at time $t = 0$. The remaining disturbances can be specified through $\delta\rho$ by the following

$$\begin{aligned} \delta\mathbf{v}(x,0) &= \left(\tfrac{w}{k}\right)\delta\rho(x,0)/\rho_0\hat{\mathbf{x}} \\ \delta P(x,0) &= \left(\tfrac{w}{k}\right)^2\delta\rho(x,0). \end{aligned} \tag{27}$$

The values chosen produces a wave traveling rightward with a velocity of 1. The simulation is evolved for 1 unit of time such that the wave returns to its initial position at time $t = 1$. Moreover, we study the convergence behavior by comparing the final state of the density to the initial density by computing the $L1$ norm,

$$L1 = \frac{1}{N}\sum_i \left|\rho_i - \rho(x_i)\right|, \tag{28}$$

where ρ_i is final density at position x_i and $\rho(x_i)$ is the density at t=0 at position x_i and N is the number of cells per dimension. Five simulations where evolved with varying resolution $N = 10, 20, 40, 80, 160$ with the initial particles laid out in a Cartesian grid. Figure 5 shows the $L1$ norm as a function of grid cells per dimension. The convergence rate is approximately second order in time and space for this smooth problem. However, we expect this to be in the case for situations where the mesh is heavily deformed. Pakmor et al. (2016) discovered that in situations where the mesh generator and center of mass significantly deviate the gradients deteriorate, leading to results that are not second order. The crux of this issue is that Equation 9 assumes the gradient calculation at the center of mass, however, we use cell center values. Pakmor et al. (2016) resolved this issue by introducing an iterative-scheme which we plan to incorporate in the next revision.

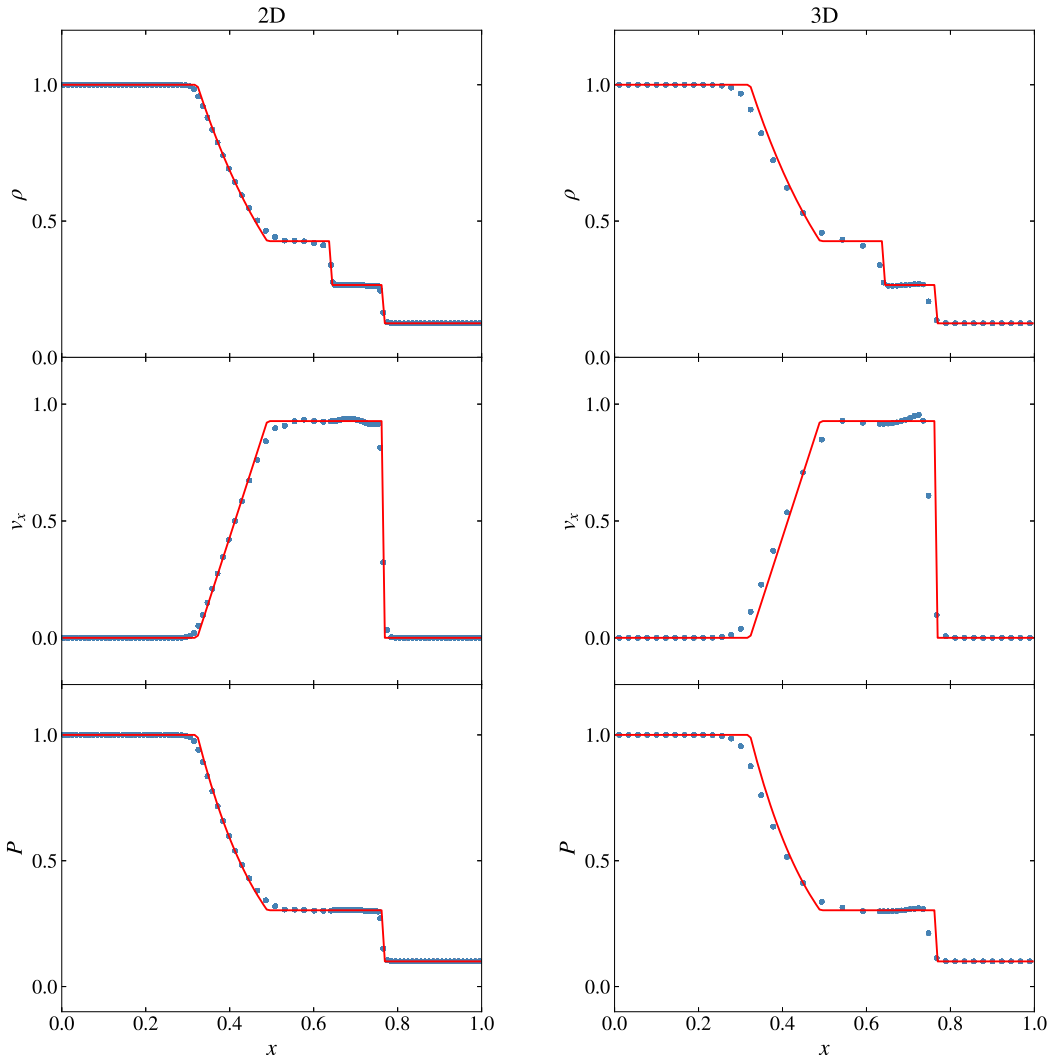

Figure 6. Profiles of density, x-component of velocity and pressure of the Sod shock-tube simulation. Left: 2D run using a total of 100×100 particles. Right: 3D run using a total of $45 \times 45 \times 45$, we only plot a slice of particles defined by $z = 0$. Light blue points are the simulation while the red line is the exact solution. This example was produced by **sod_2d_cartesian.py**, **sod_3d_cartesian.py**, **sod_2d_profiles.py**, and **sod_3d_profiles.py** scripts.

4.2.2. *Sod shock-tube*

To examine the ability of the code to handle shock propagation we perform the Sod shock-tube problem (Toro 1997). The problem consists of two different constant states at rest, separated at the midpoint of the x axis. A discontinuity exists in the density and pressure at that point. After $t = 0$ the high density region flows into the lower density region. The flow produces a rarefaction, contact discontinuity, and a shock wave emanating from the initial discontinuity. Thus, this problem creates a great test for the code's ability to capture the three wave types.

For our initial setup we use a unit box with reflecting boundary conditions with density and pressure defined as

$$\rho = \begin{cases} 1.0 & \text{for } x \leq 0.5 \\ 0.125 & \text{for } x > 0.5 \end{cases} \tag{29}$$

and

$$P = \begin{cases} 1.0 & \text{for } x \leq 0.5 \\ 0.1 & \text{for } x > 0.5 \end{cases} \tag{30}$$

with $\gamma = 1.4$. The particles are laid out in a Cartesian grid and the simulation is evolved until $t = 0.15$. The number of particles per dimension is chosen to be $N = 100$ and $N = 45$ for 2D and 3D runs, respectively. This allows a comparison of a high and low resolution run.

Figure 6 plots the particle values for density, the x-component of the velocity and the pressure; only particles with $z = 0$ are plotted in the 3D – the other particles simply overlay those plotted. The red line is the analytical solution.

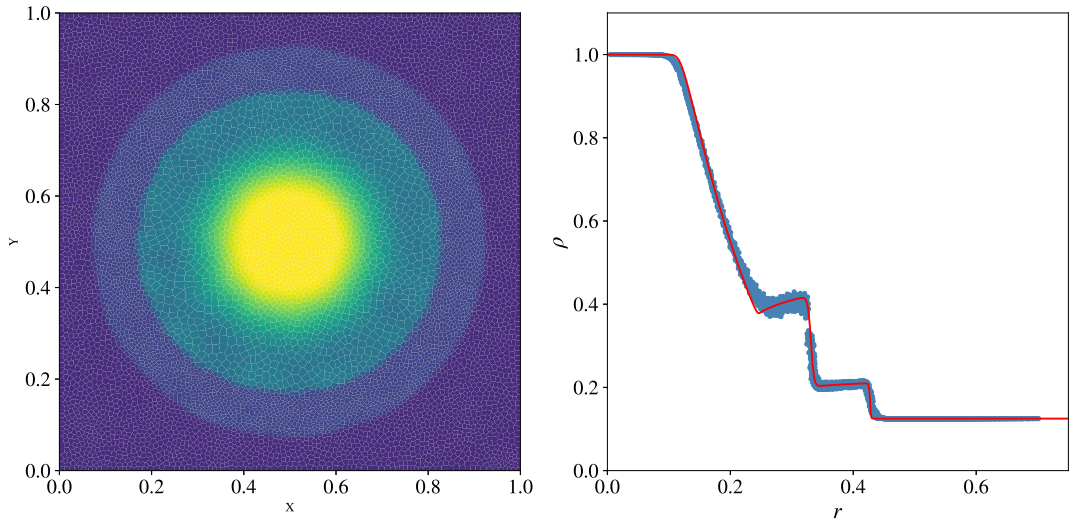

Figure 7. Density map and radial profile of the Explosion problem. Left density heatmap, the irregular cells can been seen from the random initialization. Right radial density profile is an agreement with the exact solution in red. This example was produced by **explosion_2d_random.py** and **explosion_density_panel.py** scripts.

For the 2D simulation we can see the shock is well resolved as is the contact discontinuity. Further, the Lagrangian nature of the code can be seen as many particles have been squeezed between the contact discontinuity and the shock front while particles in the rarefaction have been spread out. For the 3D, lower resolution run, the code still cleanly resolves all three waves although the contact discontinuity has been smoothed due to the lower number of particles.

4.2.3. *Cylindrical Shock*

An analog to the Sod problem is the 2D cylindrical shock (or explosion) problem, inspired by (Toro 1997). Like the Sod problem, the domain is partitioned into two constant states. However, the higher density region is now a circular region of radius r centered in a unit box. Similar to the Sod problem, the initial conditions generate a shock, contact discontinuity and rarefaction wave. However, in this case the waves are now a circular shock traveling radially outward, a circular contact discontinuity traveling in the same direction, and a rarefaction wave traveling towards the center.

We use the same values as the Sod problem except we restrict the higher density values to the center of domain with radius $r < 0.25$. Further, instead of using a Cartesian grid we sample particles uniformly for a unit square and perform 10 initial iterations of Lloyds algorithm to relax the grid, in order to reduce Poisson noise.

Figure 7 shows the density map and radial density profile. Clearly, there is a good match with the analytical solution, in red. Further, the solution captures all three waves even though the mesh was built in a random fashion. This demonstrates the flexibility of the technique since codes with moving meshes are not constrained to any initial particle placement; one can reach better accuracy by placing the particles in way that exploits the problem. Later, we will see an example of this in Evrard's problem (Section 4.3.4).

4.2.4. *Gresho vortex*

Our next problem will test the stability of the code in maintaining a dynamical equilibrium state. Gresho & Chan (1990) introduced an interesting problem to test for the conservation of angular momentum. A vortex in a unit 2D box with constant density $\rho = 1$ is set up with the following angular velocity

$$v_\phi(r) = \begin{cases} 5r & \text{for } 0 \le r < 0.2 \\ 2 - 5r & \text{for } 0.2 \le r < 0.4 \\ 0 & \text{for } \ge 0.4 \end{cases} \tag{31}$$

The angular velocity of the vortex grows linearly as one moves radially outward from the center until midway in the disk. Then the velocity decreases linearly until it vanishes at the rim of the disk. This produces a triangularly shaped velocity profile. The corresponding pressure is

$$P(r) = \begin{cases} 5 + 25/2r^2 & \text{for } 0 \le r < 0.2 \\ 9 + 25/2r^2 - 20r + 4\ln(r/0.2) & \text{for } 0.2 \le r < 0.4 \\ 3 + 4\ln(2) & \text{for } \ge 0.4. \end{cases} \tag{32}$$

The pressure is chosen such that the pressure gradient balances the centrifugal forces generated by the rotation, thus producing a solution that is independent of time. Figure 8 shows three snapshots at $t = 0.0, 0.5, 3.0$ of the azimuthal velocity. The top row is a two-dimensional density map while the bottom row is the corresponding radial profile. At

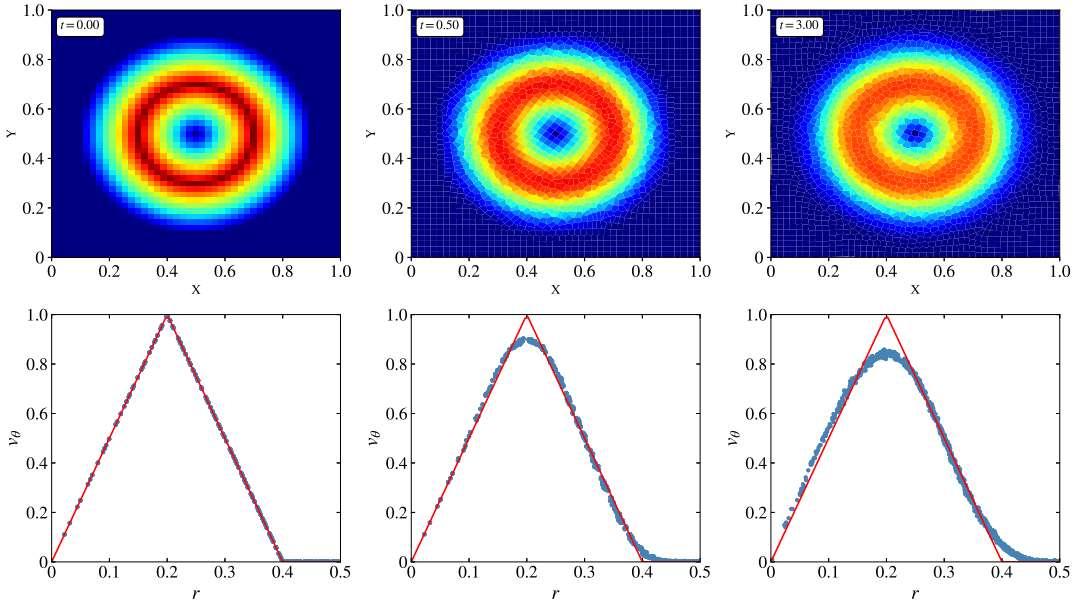

Figure 8. Density map and radial profile of azimuthal velocity for the Gesho test. Top row: time evolution of the cells at times $t = 0.0, 0.5, 3.0$. Bottom row: corresponding radial profile of azimuthal velocity. As the simulation evolves the systems remains in equilibrium. This example was produced by **gresho_2d_cartesian.py** and **gresho_density_panel.py** scripts.

time $t = 0$ all the cells are rectangular. As the system evolves the cells that are rotating become irregular polygons. There is a small amount of velocity smoothing at the radii with the highest velocities and at the rim of the vortex. However, it is evident that the system stays in equilibrium.

4.2.5. *Sedov-Taylor*

Another test that generates a shock is the Sedov-Taylor blast wave problem (Sedov et al. 1959). In this problem, a homogeneous gas is injected with a large amount of energy in a point-like region at the center of the domain. A spherical shock is generated and once the shock radius becomes much larger than the injection region, the system evolves toward a self-similar solution. The shock propagates radially outward, sweeping mass into a thin shell and creating a cavity behind the shock. The problem has a well known analytical self-similar solution: see Sedov et al. (1959) for details. Applying the Rankine-Hugoniot jump conditions at the shock front leads to a maximum density compression of $\rho_{\rm max}/\rho = (\gamma + 1)/(\gamma - 1)$, which, for $\gamma = 5/3$ results in a maximum value of 4.

We consider the 2D and 3D cases. A unit box is set up with particles in a Cartesian grid of resolution 45×45 and $45 \times 45 \times 45$ for 2D and 3D respectively. The stationary gas has a constant density of $\rho = 1.0$ and pressure $P = 10^{-6}$ with $\gamma = 5/3$. In the central cell, we set the total energy to $E = 1$. The simulation is allowed to evolve to time $t = 0.06$. Figure 9 shows the cell density as a function of radial distance from the center of the explosion. It is noted that shock is well resolved as the mesh has deformed in such a way that the shock front contains a large number of cells, as is evident in Figure 10. The center cell, where the energy is deposited, remains stationary while the cells around it move radially outward. The cells exterior to the shock remain stationary until they are swept up and compressed by the shock.

4.2.6. *Kelvin-Helmholtz instability*

For our last hydrodynamic test we consider the Kelvin-Helmholtz (KH) instability. This problem consist of a shear-flow where a single mode is excited by a velocity perturbation. Specifically, two layers with different densities are initially in pressure equilibrium. Each layer flows in the opposing direction and receives a velocity perturbation perpendicular to the interface. The perturbation grows exponentially and produces structures which are called KH instabilities. A difficulty of this problem is that numerical errors, noise, and resolution seed spurious small structure (Lecoanet et al. 2016) making direct convergence and comparisons a difficult endeavor. However, we can use this problem to visually verify the characteristics of the problem are maintained and leave a detailed comparison to future work.

We follow Springel (2010) and setup a unit periodic box with density

$$\rho = \begin{cases} 2 & \text{for } y < 0.25 \\ 1 & \text{for } 0.25 \leq y \leq 0.75 \\ 2 & \text{for } 0.75 < y, \end{cases} \tag{33}$$

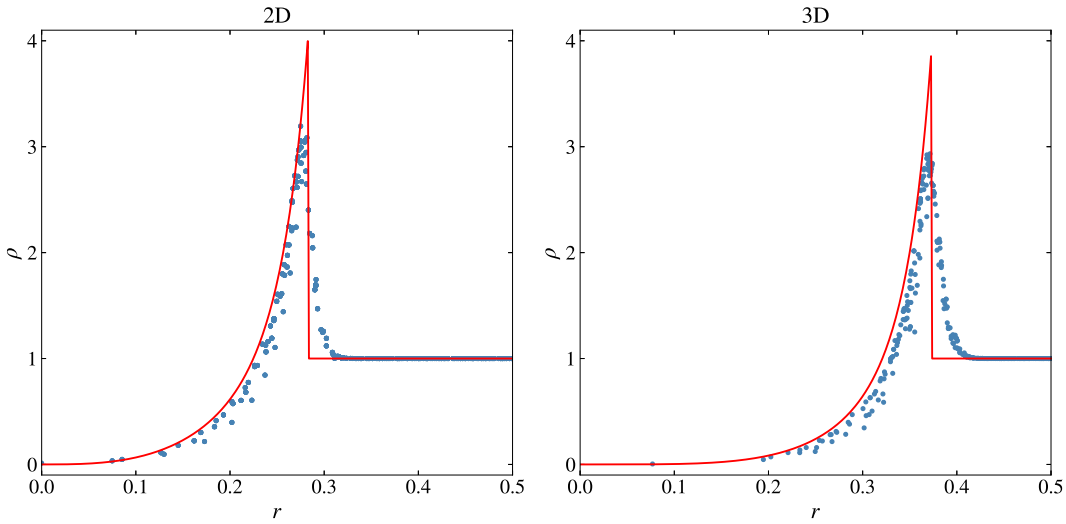

Figure 9. Density profile of Sedov-Taylor blast wave problem at $t = 0.06$. Left: 2D version with an initially Cartesian mesh of 45×45. Right: a 3D version with an initially Cartesian mesh of $45 \times 45 \times 45$; only a random sample of 45×45 cells are plotted for simplicity. Light blue points are the density at radius r from the center of the explosion while the red line is the exact solution. This example was produced by **sedov_2d_cartesian.py**, **sedov_3d_cartesian.py**, and **sedov_density_compare.py** scripts.

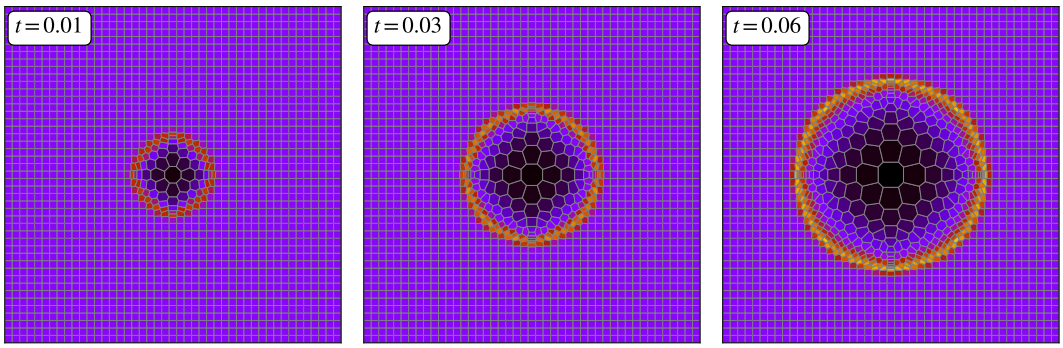

Figure 10. Evolution of the density in the Sedov test at several times. The initial cell with the energy imparted remains stationary as the cells around it move radially outward. The cells at the shock are compressed, allowing for better resolution. This example was produced by **sedov_2d_cartesian.py** and **sedov_density_panel.py** scripts.

x-component of velocity

$$v_x = \begin{cases} -0.5 & \text{for } y < 0.25 \\ 0.5 & \text{for } 0.25 \leq y \leq 0.75 \\ -0.5 & \text{for } 0.75 < y, \end{cases} \tag{34}$$

and y-component of velocity

$$v_y(x, y) = w_0 \sin(4\pi x) \left(\exp\left(-\frac{(y - 0.25)^2}{2\sigma^2} \right) + \exp\left(-\frac{(y - 0.75)^2}{2\sigma^2} \right) \right) \tag{35}$$

where $w_0 = 0.1$ and $\sigma = 0.05/\sqrt{2}$. The pressure is set to $P = 2.5$, $\gamma = 5/3$ and the simulation is evolved until time $t = 2$.

Figure 11 shows the density field for several selected times. Comparing with Springel, visually we conclude that the results are in close agreement. The formation of the Kevlin Helmholtz billows and mixing of both fluids at each time are similar.

4.3. *Gravity Tests*

In this section we continue the testing procedure by including tests with gravity. As in this previous section, tests that include hydrodynamics will incorporate linear reconstruction (Section 2.3.1), mesh movement (Section 2.4), motion correction (Section 2.4.2), and the HLLC algorithm (Section 2.3.2), unless stated otherwise. Moreover, for tests that only include gravity the kick drift kick integrator (Section 3.5) will be used. All scripts to generate the simulations and plots will be found in the **test_suite** directory.

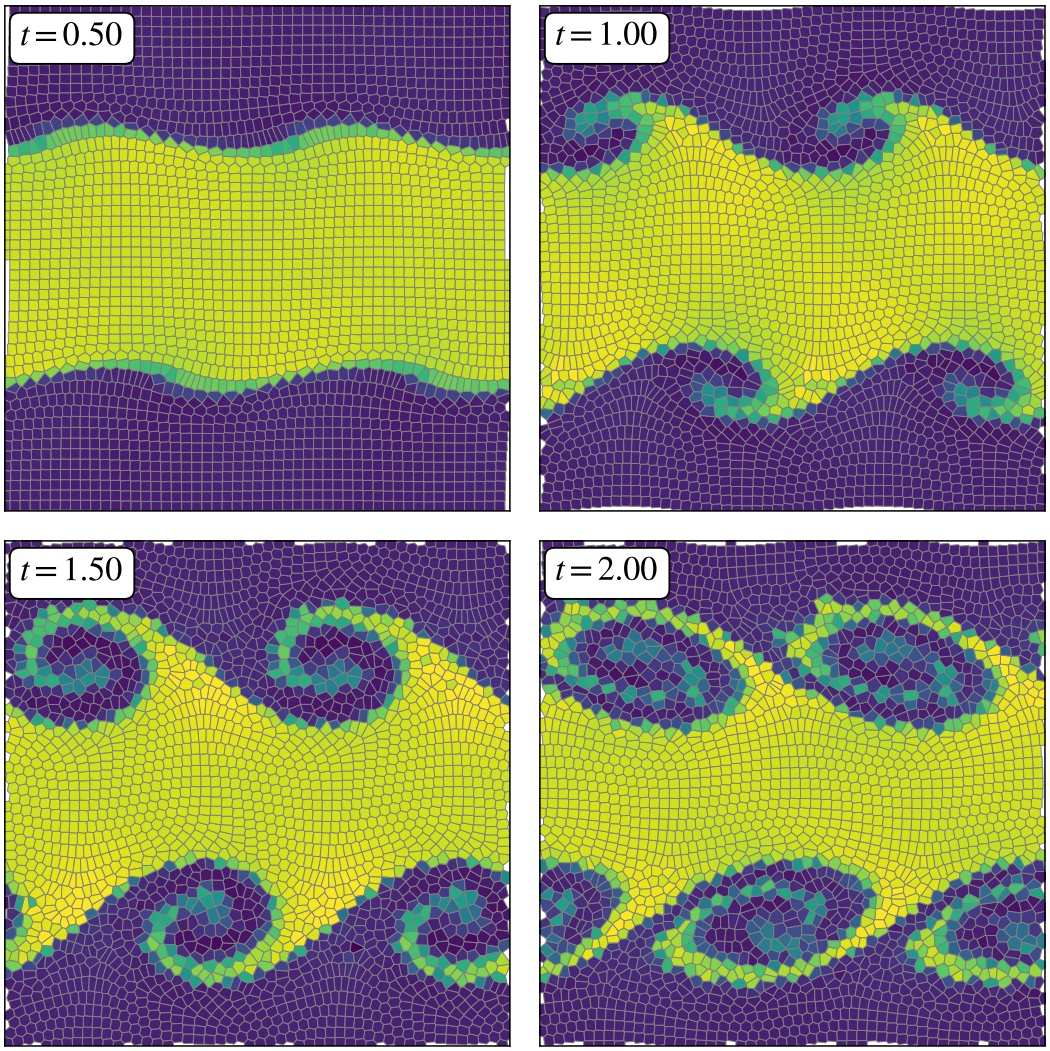

Figure 11. Evolution of the density at several times in the KH problem. We see the common traits of KH evolution, KH billows and mixing. This example was produced by **kelvin_helmholtz_2d_cartesian.py** and **kelvin_helmholtz_density_panel.py** scripts.

4.3.1. *Two body*

The first problem, in testing our gravity solver, is a simple two-body problem where two bodies interact with each other through their gravitational force. Although, this problem does not really test the implementation of the gravity tree, since only two leaves will be constructed and it is more likely that the leaves will interact with each other bypassing node moments, it does test the gravity kernel and stability and accuracy of the leap frog integrator.

For this problem an exact solution exists by reducing to a single body (Landau & Lifshitz 1969). Given two particles with masses m_1 and m_2 with positions $\vec{r}_1$ and $\vec{r}_2$ the equation of motion for the reduced mass $\mu = (1/m_1 + 1/m_2)^{-1}$ is

$$\mu \frac{d^2\vec{r}}{dt^2} = -\frac{Gm_1m_2}{r^2}\hat{r}, \tag{36}$$

where $\vec{r}$ is the separation vector $\vec{r}_1 - \vec{r}_2$. Equation 36 can be transformed to polar coordinates giving the solution

$$r = \frac{a(1-\epsilon^2)}{1 - \epsilon\cos(\theta)} \tag{37}$$

for initial conditions a and ϵ. The overall system evolves with a period of

$$T = \sqrt{\frac{4\pi^2 a^3}{Gm}}, \tag{38}$$

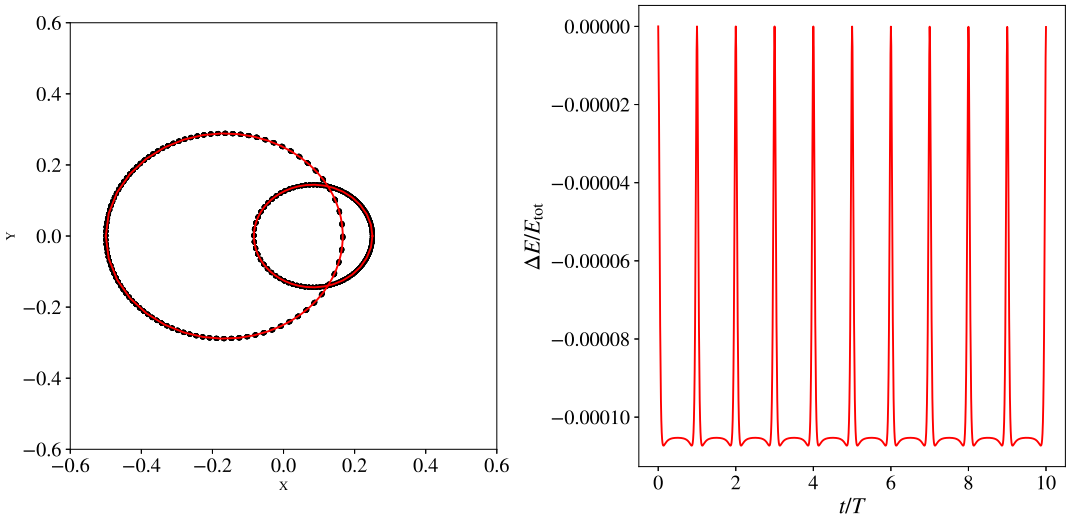

Figure 12. Left: trajectories of the two body problem for ten periods. Clearly both particles remain in their orbital path shown in red, demonstrating the stability of the leap frog integrator. Right: corresponding relative total energy error. The total energy remains accurately conserved as the worst relative error is -1.1×10^4. This example was produced by **two_body.py** and **two_body_panel.py**

where $m = m_1 + m_2$. To recover the particles positions, a final transformation of the form

$$\begin{aligned} \vec{r}_1 &= \frac{m_1}{m}\vec{r} \\ \vec{r}_2 &= -\frac{m_2}{m}\vec{r} \end{aligned} \tag{39}$$

is used. The initial position and velocity of the particles can be parameterized by a, ϵ and $q = m_1/m_2$

$$\begin{aligned} \vec{r}_1 &= a\frac{1-\epsilon}{1+q}\hat{x} \\ \vec{v}_1 &= \frac{1}{1+q}\sqrt{\frac{1+\epsilon}{1-\epsilon}}\sqrt{\frac{Gm}{a}}\hat{y} \\ \vec{r}_2 &= -q\vec{r}_1 \\ \vec{v}_2 &= -q\vec{v}_1. \end{aligned} \tag{40}$$

We set up the particles with parameter values $a = 0.5$, $\epsilon = 0.25/0.75$ and $G = 1$ and allow the simulation to evolve for 10 periods. The time step is held fixed with a value of $dt = T/1000$. In Figure 12 we show the trajectory for both particles as well as the evolution of the relative total energy error. We clearly see that both trajectories remain along the exact solution, in red, signifying the stability of the leap frog integrator. Further we see that the relative total energy error remains bounded by zero and -1.1×10^{-4} indicating that the total energy remains accurately conserved.

4.3.2. *Plummer sphere*

The Plummer sphere (Plummer 1911) is a model that can be used to describe the distribution of stars in a cluster and is commonly used to test gravity solvers. The Plummer sphere, i.e. a polytrope of index 5, has a density profile of the from

$$\rho(r) = \frac{3M}{4\pi R^3}\left(1 + (r/R)^2\right)^{-5/2}, \tag{41}$$

where M is the total mass and R is a scale parameter which sets the size of the cluster. The system is in steady state with an isotropic velocity distribution. To test our gravity solver, we initialize our particles with the given distributions and advance the system in time. We expect the system to stay in steady state, therefore we compare the initial density distribution with the final state.

For our test we chose the parameters of the Plummer sphere to be $M = 1000$ and $R = 1$ with $G = 1$. We then sampled 10,000 particles using the rejection technique outlined in Aarseth et al. (1974) to set the position and velocities. The system is allowed to evolve to time $t = 1$, which is roughly ten dynamical times, and a fixed time step of $dt = 0.001$. The gravitational tree parameters used were an opening angle 0.4 and smoothing parameter of 0.03. The left panel of Figure 13 shows the density profile at the initial and final time of the simulation with equation 41 overlaid as a reference. The density is calculated by dividing the space into spherical shells, binning and dividing by the volume. As can be clearly seen in that figure, the particles remain in a steady state, with their final distribution matching the initial distribution. The right panel of 13 shows the evolution of the relative error of the total energy of the system. The error stays well below 5×10^{-4} with a final error of 2×10^{-4}, indicating that the solver has accurately conserved the total energy of the system.

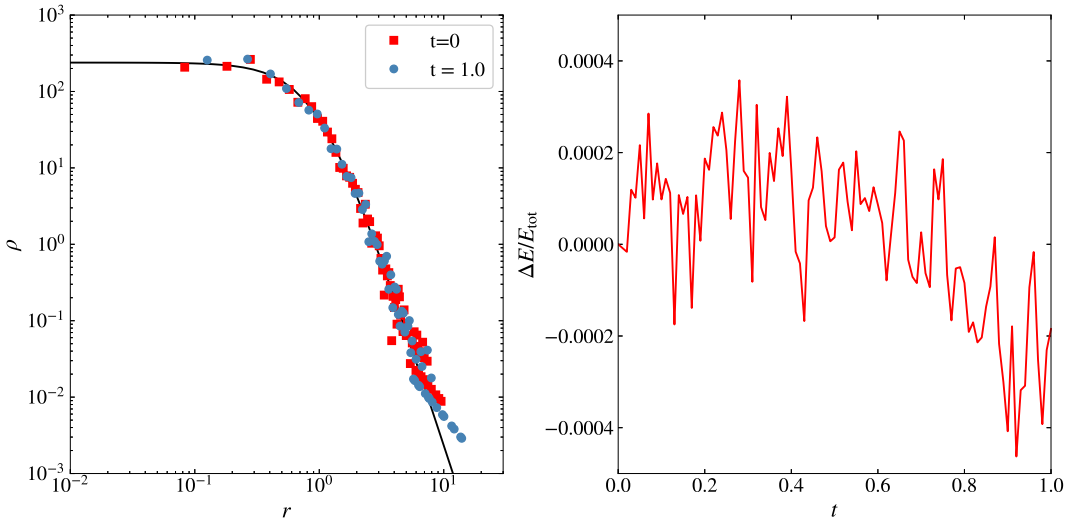

Figure 13. Density profile and relative total energy error evolution. Left: the initial and final density distribution overlaid by the Plummer density profile. The markers are shell density averages, even spaced in log r. At the the end of the evolution the particles remain in a Plummer distribution. Right: evolution of the relative total energy error. The error remains relatively small indicating the total energy remains accurately conserved. This example was produced by **plummer.py** and **plummer_panel.py**.

4.3.3. *Rayleigh-Taylor instability*

Our first hydrodynamic problem which includes gravity is the Rayleigh-Taylor instability problem. The problem consists of a dense fluid resting over a lighter fluid in the presence of an uniform vertical gravitational field. A velocity perturbation is placed in the vertical direction causing the dense field to sink while the lighter rises through buoyancy.

A rectangular Cartesian domain is chosen as $x \in [0, 1]$ and $y \in [0, 3]$ of size 50×150 with reflective boundary conditions. Typically this problem is performed with periodic boundary conditions in the x-direction and reflective in the y-direction. However, we currently don't have an implementation of mixed boundary conditions. Nonetheless, the reflective boundary in the x-direction will not affect our single mode evolution until relatively late times. The gravitational force is placed in the y-direction and has a strength of $g = 1$. The initial density is

$$\rho = \begin{cases} 1 & \text{for } y \leq 1.5 \\ 2 & \text{for } y > 1.5, \end{cases} \tag{42}$$

while the pressure is

$$P = \begin{cases} 10 - y & \text{for } y \leq 1.5 \\ 11.5 + 2(y - 1.5) & \text{for } y > 1.5. \end{cases} \tag{43}$$

The system, initially in hydrostatic equilibrium, is given a velocity perturbation in the y-direction

$$v_y = \cos(2\pi x) \exp\left(-(y - 1.5)^2 / 0.1^2\right). \tag{44}$$

We set $\gamma = 1$ and let the system evolve to a time $t = 3.0$.

In the absence of physical viscosity there is no solution that all codes will converge to (Stone et al. 2008). However, we can visually inspect our solution for common traits of this problem. Two simulations were performed, first constraining the mesh to be stationary and the second allowing the mesh to move.

In Figure 14, we see both simulations have the expected results, the lighter fluid rising around the denser fluid that sinks, with both developing billows. For the stationary grid case, the simulation maintains vertical symmetry, unlike the moving mesh counterpart that loses symmetry at the final output time. This is expected, as pointed out by Springel (2010) since the mesh correction motion (which steers the cells to become rounder) can trigger perturbations of its own. The symmetry can be maintained for longer times by increasing the resolution but ultimately both simulation lose their symmetry by round off noise.

4.3.4. *Evrard Collapse*

The final test case we explore is Evrard's collapse problem (Evrard 1988) which tests the coupling of self-gravity and hydrodynamics. The problem consists of an initially non-rotating isothermal gas sphere of mass $M = 1$ and radius $R = 1$ with density

$$\rho(r) = \begin{cases} 1/(2\pi r) & \text{for } r \leq 1 \\ 0 & \text{for } r > 1 \end{cases} \tag{45}$$

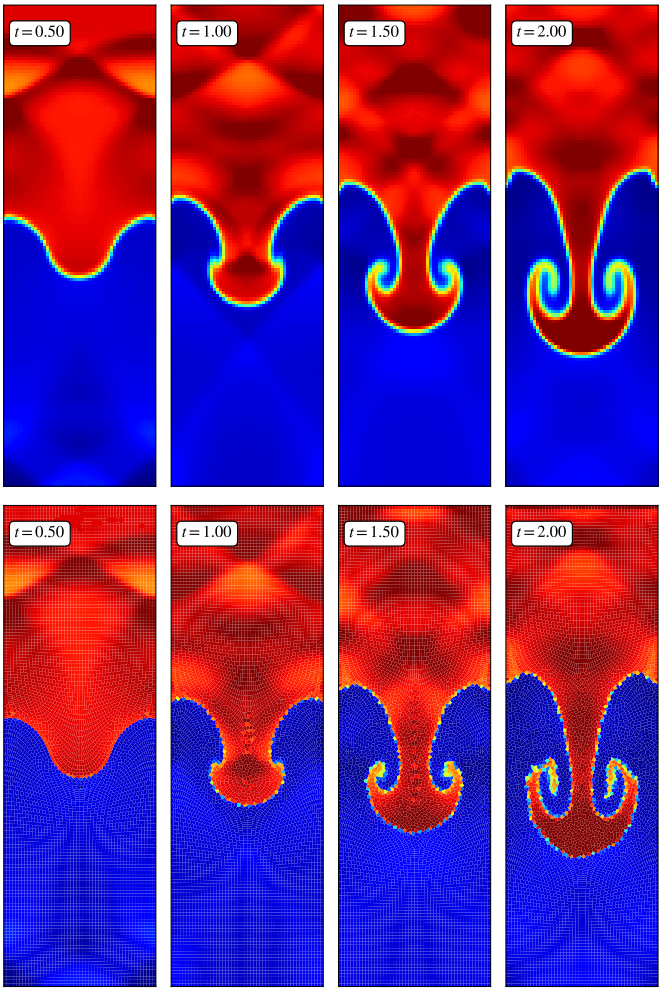

Figure 14. The evolution of the Rayleigh-Taylor instability problem with stationary mesh (top) and moving mesh (bottom). The initial mesh is a Cartesian of size 50×150. This example was produced by **rayleigh_2d_cartesian.py** and **rayleigh_density_panel.py**.

and pressure

$$P(r) = \begin{cases} 0.05/\left(3\pi r\right) & \text{for } r \leq 1 \\ 0 & \text{for } r > 1. \end{cases} \tag{46}$$

The evolution of the sphere begins with mass falling towards the center due to self-gravity. The pressure at the center rises and produces a shock traveling outward through the in-falling gas. The final state of the gas is a spherical distribution in hydrostatic virial equilibrium.

We setup an initial Cartesian grid of range $[0, 2.5]^3$ with $33 \times 33 \times 33$ particles. The gas sphere is centered at $(1.25, 1.25, 1.25)$. Due to the nature of the $1/r$ density profile, a Cartesian mesh will not resolve the high density values unless the resolution is sufficiently increased. However we have complete freedom on how to place the initial particles. Therefore, we transform particles radially inside the sphere by the following

$$r_{\text{new}} = r_{\text{old}}^{3/2}. \tag{47}$$

Such a transformation maps a grid of equally spaced particles with uniform density to a set of particles spaced in a way that the uniform density follows a $1/r$ profile. Performing this transformation produces particles with equal mass and $1/r$ density profile.

The radial averaged density, velocity and entropy are shown in Figure 15 at time $t = 0.81$ with a high resolution 1D solution for reference. At this time the shock is fully formed and is traveling outward. We see that each profile adequately follows the resolved solution in red for this low resolution run. Further we see significant error in the conservation of total energy Figure 16; a relative error of 27% at the final state. This is expected as it is pointed out by Springel (2010). The discrepancy arises from the gravitational work term which ignores the motion of mass exchanged by adjacent cells. Springel (2010) proposed a new formulation for then energy equation that results in better total energy conservation. This updated method will be added in the next revision of the code.

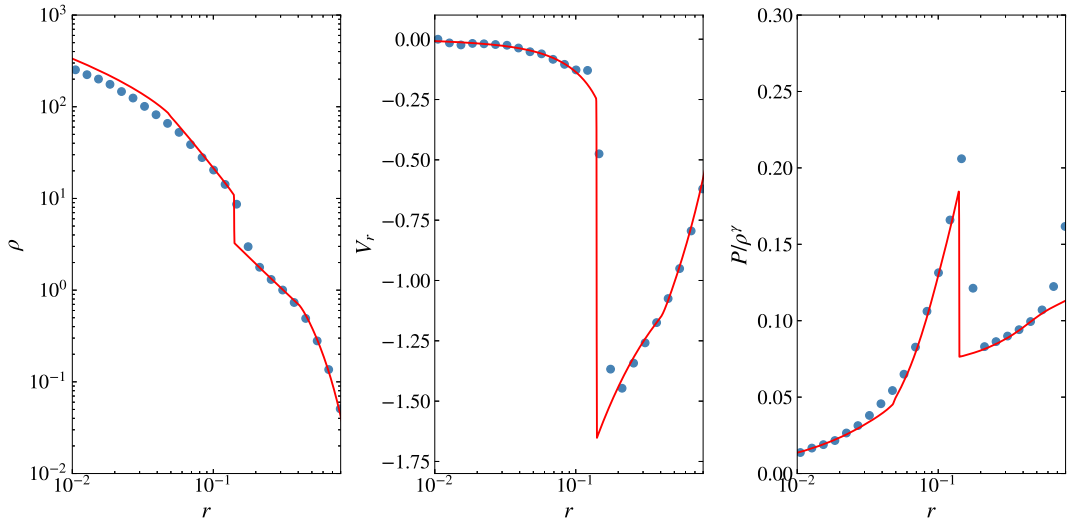

Figure 15. Profiles of the density, radial velocity and entropy at $t = 0.81$ overlaid with a high resolution 1D PPM solution. The profiles are shell averages divided by shell density. This example was produced by **evrard.py** and **evrard_profiles.py**.

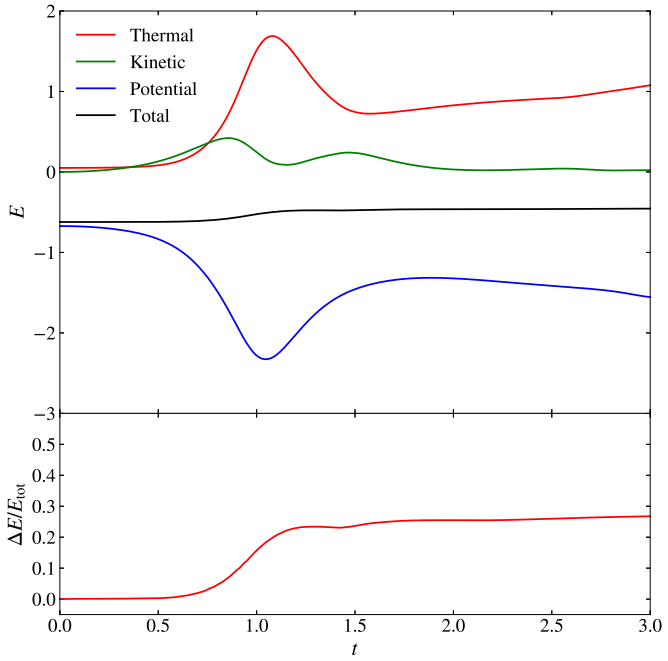

Figure 16. Evolution of total energy and each of its' component (top) and relative total energy error (bottom). Our current implementation of gravity generates considerable error in the total energy. This example was produced by **evrard.py** and **evrard_energy.py**.

5. CONCLUSIONS

In this paper, we have presented the algorithms, design and implementation used in `phd`, an object-oriented approach to a `Python` based moving mesh hydrodynamic code. The goal is to provide an easy-to-use, easy-to-modify open source Arbitrary Lagrange Eulerian toolkit. Moreover, we have described the development and thought process, and provided specific examples of using and enhancing the code. We have shown, through a series of test problems, the validity and capability of the code. Furthermore, the code, tests, and code to generate the figures can be found at `http://github.com/rickyfernandez/moving-mesh`. The code is still in its early stage and we hope that we can stimulate interest to support a development community with the aim to continually add functionality, documentation, scalability, and user support. In the future, we anticipate to add the following additions to the code:

- A chemistry and radiative cooling module
- Magnetohydrodynamics

- GPU backend, to export array based operations to the GPU.
- Individual time stepping, to save computational time in simulations with a large dynamic range.
- A general domain decomposition API to allow the inclusion of third party load-balance libraries.
- The addition of the fast multipole method and tree particle mesh gravity solvers.
- New infrastructure for problem initialization, including parallel initialization.
- Hybrid parallelization, MPI communication between nodes and OpenMP for thread based parallelism within the node.
- Unit aware computation

The structure of `phd` is relatively simple, making extensive use of object oriented programming. The use of the `Python` language allows the code to be more transparent and easier to follow, and selective use of `Cython` permits high performance. We have made considerable effort to encapsulate the algorithms. Our hope is that, in doing so, we allow the user to quickly modify or introduce new algorithms into the code. Thus, the user can spend more time and attention with the physics implementation then dealing with the side effects. Additionally, the modular approach allows the code to quickly substitute algorithms for a given simulation.

6. ACKNOWLEDGMENTS

We would like to thank Paul Duffell for his helpful insights dealing with the construction of the Voronoi mesh in the early development of the code.

REFERENCES

Aarseth, S. J., Henon, M., & Wielen, R. 1974, A&A, 37, 183
Agertz, O., Moore, B., Stadel, J., Potter, D., Miniati, F., Read, J., Mayer, L., Gawryszczak, A., Kravtsov, A., Nordlund, Å., Pearce, F., Quilis, V., Rudd, D., Springel, V., Stone, J., Tasker, E., Teyssier, R., Wadsley, J., & Walder, R. 2007, MNRAS, 380, 963
Barnes, J. & Hut, P. 1986, Nature, 324, 446
Berger, M. J. & Colella, P. 1989, Journal of Computational Physics, 82, 64
Bowyer, A. 1981, The Computer Journal, 24, 162
Dehnen, W. & Read, J. I. 2011, European Physical Journal Plus, 126, 55
Devine, K., Hendrickson, B., Boman, E., John, M. S., & Vaughan, C. 2000, in Proc. Intl. Conf. on Supercomputing, Santa Fe, New Mexico, 110–118
Duffell, P. C. & MacFadyen, A. I. 2011, ApJS, 197, 15
Edelsbrunner, H. & Shah, N. R. 1996, Algorithmica, 15, 223
Evrard, A. E. 1988, MNRAS, 235, 911
Fabri, A. & Pion, S. 2009, in Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '09 (New York, NY, USA: ACM), 538–539
Gaburov, E., Johansen, A., & Levin, Y. 2012, ApJ, 758, 103
Gingold, R. A. & Monaghan, J. J. 1977, MNRAS, 181, 375
Goldbaum, N. J., ZuHone, J. A., Turk, M. J., Kowalik, K., & Rosen, A. 2018, ArXiv e-prints
Gresho, P. M. & Chan, S. T. 1990, International Journal for Numerical Methods in Fluids, 11, 621
Hopkins, P. F. 2015, MNRAS, 450, 53
Landau, L. D. & Lifshitz, E. M. 1969, Mechanics
Lecoanet, D., McCourt, M., Quataert, E., Burns, K. J., Vasil, G. M., Oishi, J. S., Brown, B. P., Stone, J. M., & O'Leary, R. M. 2016, MNRAS, 455, 4274
Lucy, L. B. 1977, AJ, 82, 1013
Mitchell, N. L., McCarthy, I. G., Bower, R. G., Theuns, T., & Crain, R. A. 2009, MNRAS, 395, 180
Mocz, P., Vogelsberger, M., Sijacki, D., Pakmor, R., & Hernquist, L. 2014, MNRAS, 437, 397
Monaghan, J. J. 1992, ARA&A, 30, 543
—. 1997, Journal of Computational Physics, 138, 801
Morozov, D. & Peterka, T. 2016, in SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 728–738
Pakmor, R., Springel, V., Bauer, A., Mocz, P., Munoz, D. J., Ohlmann, S. T., Schaal, K., & Zhu, C. 2016, MNRAS, 455, 1134
Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. 2011, Journal of Machine Learning Research, 12, 2825
Peterka, T., Morozov, D., & Phillips, C. L. 2014, SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, 997
Plummer, H. C. 1911, MNRAS, 71, 460
Prabhu Ramachandran. 2016, in Proceedings of the 15th Python in Science Conference, ed. Sebastian Benthall & Scott Rostrup, 122 – 129
Price, D. J. 2008, Journal of Computational Physics, 227, 10040
Ritchie, B. W. & Thomas, P. A. 2001, MNRAS, 323, 743
Robertson, B. E., Kravtsov, A. V., Gnedin, N. Y., Abel, T., & Rudd, D. H. 2010, MNRAS, 401, 2463
Sedov, L., FRIEDMAN, M., & Holt, M. 1959, Similarity and Dimensional Methods in Mechanics. (London)
Springel, V. 2005, MNRAS, 364, 1105
—. 2010, MNRAS, 401, 791
Steinberg, E., Yalinewich, A., & Sari, R. 2016, MNRAS, 459, 1596
Stone, J. M., Gardiner, T. A., Teuben, P., Hawley, J. F., & Simon, J. B. 2008, ApJS, 178, 137
Tasker, E. J., Brunino, R., Mitchell, N. L., Michielsen, D., Hopton, S., Pearce, F. R., Bryan, G. L., & Theuns, T. 2008, MNRAS, 390, 1267
The Enzo Collaboration, Bryan, G. L., Norman, M. L., O'Shea, B. W., Abel, T., Wise, J. H., Turk, M. J., Reynolds, D. R., Collins, D. C., Wang, P., Skillman, S. W., Smith, B., Harkness, R. P., Bordner, J., Kim, J.-h., Kuhlen, M., Xu, H., Goldbaum, N., Hummels, C., Kritsuk, A. G., Tasker, E., Skory, S., Simpson, C. M., Hahn, O., Oishi, J. S., So, G. C., Zhao, F., Cen, R., & Li, Y. 2013, ArXiv e-prints
Toro, E. F. 1997, Riemann solvers and numerical methods for fluid dynamics : a practical introduction (Berlin, New York: Springer)
Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., & Norman, M. L. 2011, ApJS, 192, 9
van de Weygaert, R. 1994, A&A, 283, 361
Van Leer, B. 1997, On The Relation Between The Upwind-Differencing Schemes Of Godunov, Engquist—Osher and Roe, ed. M. Y. Hussaini, B. van Leer, & J. Van Rosendale (Berlin, Heidelberg: Springer Berlin Heidelberg), 33–52

Wadsley, J. W., Veeravalli, G., & Couchman, H. M. P. 2008, MNRAS, 387, 427
Watson, D. F. 1981, The Computer Journal, 24, 167
Whitehurst, R. 1995, MNRAS, 277, 655
Yalinewich, A., Steinberg, E., & Sari, R. 2015, ApJS, 216, 35