PeleLM

An adaptive, parallel, hydrodynamics code for low Mach number chemically reacting flows

User's Guide

March 19, 2018

Preface

This documentation is a work in progress. PeleLM is rapidly evolving, so portions of the text are likely to be incomplete.

PeleLM is developed primarily at the Center for Computational Sciences and Engineering at Lawrence Berkeley National Laboratory. Development has been supported by the SciDAC Program of the DOE Office of Mathematics, Information, and Computational Sciences under the U.S. Department of Energy under contract No. DE-AC02-05CH11231.

Past and present contributors to the PeleLM code base include:

Ann Almgren
Andy Aspden
John Bell
Vince Beckner
Marc Day
Matthew Emmett
Candace Gilet
Ray Grout
Mike Lijewski
Andy Nonaka
Will Pazner
Chuck Rendleman
Weiqun Zhang

PeleLM uses the AMReX library, developed at the Center for Computational Sciences and Engineering (CCSE) at Lawrence Berkeley National Laboratory. AMReX is distributed separately from PeleLM (see `https://github.com/AMReX-Codes/amrex`)

The current version of the PeleLM User's Guide can be found in the PeleLM git repository (on ORNL GitLab) under `PeleLM/Docs/UsersGuide/`.

For questions about obtaining/using the code or this User's Guide, contact Marc Day `MSDay@lbl.gov` and Andy Nonaka `AJNonaka@lbl.gov` of CCSE.

Mailing list details coming soon.

Part I

Overview of the PeleLM Algorithm

CHAPTER 1

Introduction

1.1 History of PeleLM

PeleLM describes the evolution of chemically reacting low Mach number flows. The core libraries for managing the subcycling adaptive mesh refinement (AMR) grids and communication are found in the AMReX library (see `https://github.com/AMReX-Codes/amrex`).

PeleLM also uses source code and base classes for integrating the variable-density incompressible Navier-Stokes equations found in IAMR (see `https://github.com/AMReX-Codes/IAMR`).

The core algorithms in PeleLM are described in a series of papers:

- *A conservative, thermodynamically consistent numerical approach for low Mach number combustion. I. Single-level integration*, A. Nonaka, J. B. Bell, and M. S. Day. Submitted for publication, 2017. `https://ccse.lbl.gov/Publications/nonaka/LMC_Pressure.pdf`
- *A Deferred Correction Coupling Strategy for Low Mach Number Flow with Complex Chemistry*, A. Nonaka, J. B. Bell, M. S. Day, C. Gilet, A. S. Almgren, and M. L. Minion, Combust. Theory and Modelling, 16(6), 1053-1088, 2012. `http://www.tandfonline.com/doi/abs/10.1080/13647830.2012.701019` [12]
- *Numerical Simulation of Laminar Reacting Flows with Complex Chemistry*, M. S. Day and J. B. Bell, Combust. Theory Modelling 4(4) pp.535-556, 2000. `http://www.tandfonline.com/doi/abs/10.1088/1364-7830/4/4/309` [8]
- *An Adaptive Projection Method for Unsteady, Low-Mach Number Combustion*, R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee, Comb. Sci. Tech., 140, pp. 123-168, 1998. `http://www.tandfonline.com/doi/abs/10.1080/00102209808915770` [13]

- *A Conservative Adaptive Projection Method for the Variable Density Incompressible Navier-Stokes Equations*, A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome, J. Comp. Phys., 142, pp. 1-46, 1998. `http://www.sciencedirect.com/science/article/pii/S0021999198958909` [1]

1.2 Brief Overview of Low Speed Approximations

There are many low speed formulations of the equations of hydrodynamics in use, each with their own applications. All of these methods share in common a constraint equation on the velocity field that augments the equations of motion.

1.2.1 Incompressible Hydrodynamics

The simplest low Mach number approximation is incompressible hydrodynamics. This approximation is formally the $M \rightarrow 0$ limit of the Navier-Stokes equations. In incompressible hydrodynamics, the velocity satisfies a constraint equation:

$$\nabla \cdot \mathbf{U} = 0 \tag{1.1}$$

which acts to instantaneously equilibrate the flow, thereby filtering out soundwaves. The constraint equation implies that

$$D\rho/Dt = 0 \tag{1.2}$$

(through the continuity equation) which says that the density is constant along particle paths. This means that there are no compressibility effects modeled in this approximation.

1.2.2 Low-Mach Number Combustion

In the low Mach number combustion model, the pressure is decomposed into a dynamic, π, and thermodynamic component, p_0, the ratio of which is $O(M^2)$. The total pressure is replaced everywhere by the thermodynamic pressure, except in the momentum equation. This decouples the pressure and density and filters out the sound waves. Large amplitude density and temperature fluctuations are allowed. The only requirement is that the total pressure stay close to the background pressure, which is assumed constant. This requirement can be expressed as:

$$p = p_0 \tag{1.3}$$

and differentiating this along particle paths leads to a constraint on the velocity field:

$$\nabla \cdot \mathbf{U} = S \tag{1.4}$$

This looks like the constraint for incompressible hydrodynamics, but now we have a source term, S, representing the local compressibility effects due to the energy generation and thermal diffusion. Since the background pressure is taken to be constant, we cannot model flows that cover a large fraction of a pressure scale height. However, this method is ideal for exploring the physics of flames.

1.2.3 Other Models

We defer discussion of anelastic and pseudo-incompressible methods for now, as they are more relevant large-gravity or large-scale flows with density and pressure stratification. For more detailed discussion, refer to our Maestro code (`http://amrex-astro.github.io/MAESTRO/index.html`).

1.3 Projection Methods 101

Most combustion hydrodynamics codes (e.g. S3D [10]) solve the compressible Navier-Stokes equations, which can be written in the form:

$$\mathbf{U}_t + \nabla \cdot F(\mathbf{U}) = S \tag{1.5}$$

where $\mathbf{U}$ is a vector of conserved quantities, $\mathbf{U} = (\rho, \rho u, \rho E)$, with ρ the density, u the velocity, E the total energy per unit mass, and S are source terms. This system can be expressed as a coupled set of advection, diffusion, reaction equations:

$$\mathbf{q}_t + A(\mathbf{q})\nabla\mathbf{q} + D = \mathcal{S} \tag{1.6}$$

where $\mathbf{q}$ are called the primitive variables, A is the advective flux Jacobian, $A \equiv \partial F/\partial U$, D are diffusion terms, and $\mathcal{S}$ are the transformed sources. The eigenvalues of the matrix A are the characteristic speeds—the real-valued speeds at which information propagates in the system, u and $u \pm c$, where c is the sound speed. Solution methods for the compressible equations that are strictly conservative make use of this wave-nature to compute advective fluxes at the interfaces of grid cells. Diffusive fluxes can be computed either implicit or explicit in time, and are added to the advective fluxes, and used, along with the source terms to update the state in time. An excellent introduction to these methods is provided by LeVeque's book [11]. The timestep for these methods is limited by all three processes and their numerical implementation. Typically, advection terms are treated time-explicitly, and the time step will be constrained by the time it takes for the maximum characteristic speed to traverse one grid cell. However, in low speed flow applications, it can be shown the acoustics transport very little energy in the system. As a result, the time-step restrictions arising from numerical treatement of the advection terms can be unnecessarily limited, even if A-stable methods are used to incorporate the diffusion and source terms.

In contrast, solving low Mach number systems (including the equations of incompressible hydrodynamics) typically involves a stage where one or more advection-like equations are solved (representing, e.g. conservation of mass and momentum), and coupling that advance with a divergence constraint on the velocity field. For example, the equations of invicid constant-density incompressible flow are:

$$\mathbf{U}_t = -\mathbf{U} \cdot \nabla\mathbf{U} + \nabla p \tag{1.7}$$

$$\nabla \cdot \mathbf{U} = 0 \tag{1.8}$$

Here, $\mathbf{U}$ represents the velocity vector[1] and p is the dynamical pressure. The time-evolution equation for the velocity (Eq. 1.7) can be solved using techniques similar to those developed for

[1] Here we see an unfortunate conflict of notation between the compressible hydro community and the incompressible community. In papers on compressible hydrodynamics, $\mathbf{U}$ will usually mean the vector of conserved quantities. In incompressible / low speed papers, $\mathbf{U}$ will mean the velocity vector.

compressible hydrodynamics, updating the old velocity, $\mathbf{U}^n$, to the new time-level, $\mathbf{U}^\star$. Here the '$\star$' indicates that the updated velocity does not, in general, satisfy the divergence constraint. A projection method will take this updated velocity and force it to obey the constraint equation. The basic idea follows from the fact that any vector field can be expressed as the sum of a divergence-free quantity and the gradient of a scalar. For the velocity, we can write:

$$\mathbf{U}^\star = \mathbf{U}^d + \nabla\phi \tag{1.9}$$

where $\mathbf{U}^d$ is the divergence free portion of the velocity vector, $\mathbf{U}^\star$, and ϕ is a scalar. Taking the divergence of Eq. (1.9), we have

$$\nabla^2\phi = \nabla\cdot\mathbf{U}^\star \tag{1.10}$$

(where we used $\nabla\cdot\mathbf{U}^d = 0$). With appropriate boundary conditions, this Poisson equation can be solved for ϕ, and the final, divergence-free velocity can be computed as

$$\mathbf{U}^{n+1} = \mathbf{U}^\star - \nabla\phi \tag{1.11}$$

Because soundwaves are filtered, the timestep constraint now depends only on $|\mathbf{U}|$.

Extensions to variable-density incompressible flows [6] involve a slightly different decomposition of the velocity field and, as a result, a slightly different Poisson equation. There is also a variety of different ways to express what is being projected [2], and different discretizations of the divergence and gradient operators lead to slightly different mathematical properties of the methods (leading to "approximate projections" [3]). Finally, for second-order methods, two projections are typically done per timestep. The first (the 'MAC' projection [5]) operates on the half-time, edge-centered advective velocities, making sure that they satisfy the divergence constraint. These advective velocities are used to construct the fluxes through the interfaces to advance the solution to the new time. The second/final projection operates on the cell-centered velocities at the new time, again enforcing the divergence constraint. The PeleLM algorithm performs both of these projections.

The PeleLM algorithm builds upon these ideas, using a different velocity constraint equation that captures the compressibility due to local sources and large-scale stratification.

CHAPTER 2

Equation Set

2.1 The low Mach number flow equations

TODO: This is just a place holder for a more generic description of the model, which will have to discuss the CHEMKIN assumptions, EGLib, etc.

PeleLM solves the reacting navier-Stokes flow equations in the low Mach number regime. The methodology treats the fluid as a mixture of perfect gases; the corresponding conservation equations for an open domain are

$$\begin{aligned}
\frac{\partial(\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u} + \tau) &= -\nabla\pi + \rho\,\boldsymbol{F}, \\
\frac{\partial(\rho Y_i)}{\partial t} + \nabla \cdot (\rho Y_i \boldsymbol{u} + \boldsymbol{\mathcal{F}}_i) &= \rho\,\dot{\omega}_i, \\
\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho h \boldsymbol{u} + \boldsymbol{\mathcal{Q}}) &= 0,
\end{aligned}$$

where ρ is the density, $\boldsymbol{u}$ is the velocity, h is the mass-weighted enthalpy, T is temperature. Y_i is the mass fraction and $\dot{\omega}_i$ is the molar production rate for species i. A mixture-averaged model is assumed for diffusive transport, ignoring Dufour and Soret effects; τ is the stress tensor; $\boldsymbol{\mathcal{Q}} = h_i\boldsymbol{\mathcal{F}}_i - \lambda\nabla T$ is the heat flux and $\boldsymbol{\mathcal{F}}_i = -\rho\boldsymbol{\mathcal{D}}_i\nabla X_i$ is the species diffusion. The momentum source term, $\boldsymbol{F}$, is a long-wavelength forcing term designed to establish and maintain turbulence with the desired properties.

The chemical kinetics and transport are modelled using the dodecane mechanism of You *et al.* [14], consisting of 56 species with 289 fundamental reactions. These evolution equations are supplemented by CHEMKIN-compatible databases for thermodynamic quantities. Transport properties are computed using EGLIB [9].

The basic discretisation combines a spectral deferred corrections (SDC) coupling of chemistry and transport [12] with a density-weighted approximate projection method for low Mach number flow

[8]. The projection method implements a constrained evolution on the velocity field via the SDC iterations, which ensures that the update simultaneously satisfies the equation of state and discrete conservation of mass and total enthalpy. A time-explicit approach is used for advection; faster diffusion and chemistry processes are treated time-implicitly, and iteratively coupled together within the defferred corrections strategy. Since the low Mach system does not support acoustic waves, the time step size is governed by a CFL constraint based on advective transport. The integration algorithm is second-order accurate in space and time.

The performance of the numerical scheme for direct numerical simulation of premixed flame systems in regimes comparable to the present study was examined in [4]. An *effective* Kolmogorov length scale, η_{eff}, was formulated, which measures the actual Kolmogorov length scale realised in a simulation at a given resolution. Here, the most computationally demanding simulation, having the highest turbulence levels, has a computational cell width that is approximately 1.27 times the Kolmogorov length scale, η. In this case, the numerical scheme produces $\eta_{\mathit{eff}}/\eta < 1.03$. All other cases were better resolved.

Part II

Using PeleLM

CHAPTER 3

Getting Started

3.1 Downloading the Code

PeleLM is built on top of the IAMR code, which itself built on top of the AMReX framework. In order to run PeleLM you must download separate git modules for PeleLM, IAMR, and AMReX.

First, make sure that `git` is installed on your machine—we recommend version 1.7.x or higher.

1. Download the AMReX repository by typing:

   ```
   git clone https://github.com/AMReX-Codes/amrex.git
   ```

   This will create a folder called `amrex/` on your machine. Set the environment variable, `AMREX_HOME`, on your machine to point to the path name where you have put AMReX. You can add this to your `.bashrc` as:

   ```
   export AMREX_HOME="/path/to/amrex/"
   ```

2. Download the IAMR repository by typing:

   ```
   git clone https://github.com/AMReX-Codes/IAMR.git
   ```

   This will create a folder called `IAMR/` on your machine. Set the environment variable, `IAMR_HOME`.

3. Obtain an ORNL GitLab account and clone the Pele repositories:

   ```
   git clone https://<username>@code.ornl.gov/Pele/PeleLM.git
   ```

   ```
   git clone https://<username>@code.ornl.gov/Pele/PelePhysics.git
   ```

This will create folders called PeleLM/ and PelePhysics/ on your machine. Set the environment variables, PELELM_HOME and PELE_PHYSICS_HOME.

4. You will want to periodically update each of these repositories by typing `git pull` within each repository.

3.2 Building the Code

The PelePhysics/ repository contains chemistry models and the encapsulating source code that is used by both PeleLM and PeleC (the compressible framework). More information on the source and data files for the chemistry models is coming soon.

In PeleLM each different problem setup is stored in its own sub-folder under PeleLM/Exec/, and a local version of the PeleLM executable is built directly in that folder. The name of the executable (generated by the make system) encodes several of the build characteristics, including dimensionality of the problem, compiler name, and whether MPI and/or OpenMP were linked with the executable. Thus, several different build configurations may coexist simultaneously in a problem folder.

The build system is based on GNU make and is relatively self-contained. We have accumulated specialized building setups over the years for a wide variety of hardware configurations, and the system is quite easy to modify to add new machine/OS/compiler types and site-specific options (optimizations, cross-compiles, user-maintained includes, module-based strategies, etc). The system currently supports a wide range of Linux, OSX, Windows (via CYGWIN), AIX and BGL configurations. With some luck, your machine will be supported "out of the box" – however, if you run into problems and need assistance building on your hardware, contact Marc Day (MSDay@lbl.gov).

In the following, we step through building a representative PeleLM executable.

1. We will work in the folder containing setup for the "Flame In A Box" problem in 2-d and 3-d (PeleLM/Exec/FlameInABox). In this setup, cold fuel enters the domain bottom and passes through a flame sheet. Hot products exit the domain at the top. The sides of the domain are periodic, and the coordinates are cartesian. From the folder in which you checked out the PeleLM git repo, type

   `cd PeleLM/Exec/FlameInABox`

2. In FlameInABox/, edit the GNUmakefile, and set

   `DIM = 2` (for example)

   `COMP = gnu` (or your favorite C++/F90 compiler suite)

   `DEBUG = FALSE`

   `USE_MPI = TRUE`

   `USE_OMP = FALSE`

   If you want to try compilers other than those in the GNU suite, and you find that they don't work, please let us know.

   To build a serial (single-processor) code, set `USE_MPI = FALSE`. This will compile the code without the MPI library. If you want to do a parallel run, set `USE_MPI = TRUE`. In this case,

the build system will need to know about your MPI installation. This can be done by editing the makefiles in the AMReX tree.

The resulting executable will look something like `PeleLM2d.gnu.MPI.ex`, suggesting that this is a 2-d version of the code, made with `COMP=gnu` and `USE_MPI=TRUE`.

3.3 Running the Code

1. PeleLM takes an input file as its first command-line argument. The file may contain a set of parameter definitions that will overrides defaults set in the code. To run PeleLM with an example inputs file, type:

   ```
   ./PeleLM2d.gnu.MPI.ex inputs.2d-regt
   ```

2. PeleLM typically generates subfolders in the current folder that are named `plt00000/`, `plt00020/`, etc, and `chk00000/`, `chk00020/`, etc. These are "plotfiles" and "checkpoint" files. The plotfiles are used for visualization of derived fields; the checkpoint files are used for restarting the code.

   The output folders contain a set of ASCII and binary files. The field data is generally written in a self-describing binary format; the ASCII header files provide additional metadata to give AMR context to the field data.

3.4 Visualization of the Results

There are several options for visualizing the data. The popular VisIt package supports the AMReX file format natively, as does the yt python package. The standard tool used within the AMReX-community is Amrvis, a package developed and supported by CCSE that is designed specifically for highly efficient visualization of block-structured hierarchical AMR data.

For more information on how to use Amrvis and VisIt, refer to the AMReX User's guide in the AMReX git repository (see Section 3.1 for download instructions). For more information on how to use yt, refer to Section 8.2

You have now completed a brief introduction to PeleLM.

3.5 Other Distributed Problem Setups

`PeleLM/Exec/DiffFlame` - More info soon!

CHAPTER 4

Inputs Files

The PeleLM executable uses two inputs files at runtime to set and alter the behavior of the algorithm and initial conditions.

The main inputs file, typically named `inputs` is used to set AMReX parameters and the control flow in the C++ portions of the PeleLM code. Each parameter here has a namespace (like `amr.`*`optionname`* or `ns.`*`optionname`*). Parameters set here are read using the AMReX `ParmParse` class infrastructure.

The second inputs file, typically named `probin` is used by the Fortran code that initializes the problem setup. It is read at problem initialization (via a Fortran `namelist`) and the problem-specific quantities are stored in a Fortran module `probdata_module` defined in the problem's `probdata.f90` file.

Only the `inputs` file is specified on the commandline. The associated `probin` file is specified in the `inputs` file using the `amr.probin_file` parameter, e.g.,

```
amr.probin_file = my_special_probin
```

for example, has the Fortran code read a file called `my_special_probin`.

4.1 Working with `probin` Files

There are three different Fortran `namelist`s that can be defined in the `probin` file:

- `&fortin` is the main `namelist` read by the problem's `probinit` subroutine in the `Prob_?d.f90` file.
- `&heattransin` is used to set the ambient pressure and various algorithmic options.

- `&flctin` is used to read in turbulence files to set the space and time-dependent inflow.
- `&control` is used dynamically control the inflow velocity.

4.2 Common `inputs` Options

Important: because the `inputs` file is handled by the C++ portion of the code, any quantities you specify in scientific notation, must take the form `1.e5` and not `1.d5`—the '`d`' specifier is not recognized.

Additionally, note that in PeleLM, all quantities are in MKS units.

4.2.1 Problem Geometry

The `geometry.` namespace is used by AMReX to define the computational domain. The main parameters here are:

- `geometry.prob_lo`: physical location of low corner of the domain (type: `Real`; must be set)

  Note: a number is needed for each dimension in the problem

- `geometry.prob_hi`: physical location of high corner of the domain (type: `Real`; must be set)

  Note: a number is needed for each dimension in the problem

- `geometry.coord_sys`: coordinate system, 0 = Cartesian, 1 = r-z (2-d only), 2 = spherical (1-d only) (must be set)

- `geometry.is_periodic`: is the domain periodic in this direction? `0` if false, `1` if true (default: `0 0 0`)

  Note: an integer is needed for each dimension in the problem

As an example, the following:

```
geometry.prob_lo = 0 0 0
geometry.prob_hi = 1.e8 2.e8 2.e8
geometry.coord_sys = 0
geometry.is_periodic = 0 1 0
```

defines the domain to run from $(0, 0, 0)$ at the lower left to $(10^8, 2 \times 10^8, 2 \times 10^8)$ at the upper right in physical space, specifies a Cartesian geometry, and makes the domain periodic in the y-direction only. The problem center is set to be halfway in between the lower left and upper right corners.

4.2.2 Domain Boundary Conditions

Boundary conditions are specified using integer keys that are interpreted by AMReX. The runtime parameters that we use are:

- `ns.lo_bc`: boundary type of each low face (must be set)
- `ns.hi_bc`: boundary type of each high face (must be set)

The valid boundary types are:

0 – Interior / Periodic	3 – Symmetry
1 – Inflow	4 – Slip Wall
2 – Outflow	5 – No Slip Wall

Note: ns.lo_bc and ns.hi_bc must be consistent with geometry.is_periodic—if the domain is periodic in a particular direction then the low and high bc's must be set to 0 for that direction.

As an example, the following:

```
ns.lo_bc = 1 4 0
ns.hi_bc = 2 4 0

geometry.is_periodic = 0 0 1
```

would define a problem with inflow (1) in the low-x direction, outflow (2) in the high-x direction, slip wall (4) on the low and high y-faces, and periodic in the z-direction.

4.2.3 Resolution

The grid resolution is specified by defining the resolution at the coarsest level (level 0) and the number of refinement levels and factor of refinement between levels. The relevant parameters are:

- amr.n_cell: number of cells in each direction at the coarsest level (Integer > 0; must be set)
- amr.max_level: number of levels of refinement above the coarsest level (Integer ≥ 0; must be set)
- amr.ref_ratio: ratio of coarse to fine grid spacing between subsequent levels (2 or 4; must be set)
- amr.regrid_int: how often (in terms of number of steps) to regrid (Integer; must be set)
- amr.regrid_on_restart: should we regrid immediately after restarting? (0 or 1; default: 0)

Note: if amr.max_level = 0 then you do not need to set amr.ref_ratio or amr.regrid_int.

Some examples:

```
amr.n_cell = 32 64 64
```

would define the domain to have 32 cells in the x-direction, 64 cells in the y-direction, and 64 cells in the z-direction *at the coarsest level*. (If this line appears in a 2D inputs file then the final number will be ignored.)

```
amr.max_level = 2
```

would allow a maximum of 2 refined levels in addition to the coarse level. Note that these additional levels will only be created only if the tagging criteria are such that cells are flagged as needing refinement. The number of refined levels in a calculation must be $\leq$ amr.max_level, but can change in time and need not always be equal to amr.max_level.

```
amr.ref_ratio = 2 4
```

would set factor of 2 refinement between levels 0 and 1, and factor of 4 refinement between levels 1 and 2. Note that you must have at least `amr.max_level` values of `amr.ref_ratio` (Additional values may appear in that line and they will be ignored).

```
amr.regrid_int = 2 2
```

tells the code to regrid every 2 steps. Thus in this example, new level 1 grids will be created every 2 level-0 time steps, and new level 2 grids will be created every 2 level-1 time steps. If `amr.regrid_int` < 0 for any level, then regridding starting at that level will be disabled. If `amr.regrid_int = -1` only, then we never regrid for any level. Note that this is not compatible with `amr.regrid_on_restart = 1`.

4.2.4 Regridding

The details of the regridding strategy are described in a later section; here we cover how the input parameters can control the gridding.

As described later, the user defines Fortran subroutines which tag individual cells at a given level if they need refinement. This list of tagged cells is sent to a grid generation routine, which uses the Berger-Rigoutsos algorithm [7] to create rectangular grids that contain the tagged cells.

The relevant runtime parameters are:

- `amr.regrid_file`: name of file from which to read the grids (text; default: no file)

  If set to a filename, e.g. `fixed_girds`, then list of grids at each fine level are read in from this file during the gridding procedure. These grids must not violate the `amr.max_grid_size` criterion. The rest of the gridding procedure described below will not occur if `amr.regrid_file` is set.

- `amr.grid_eff`: grid efficiency (Real > 0 and < 1; default: 0.7)

- `amr.n_error_buf`: radius of additional tagging around already tagged cells (Integer ≥ 0; default: 1)

- `amr.max_grid_size`: maximum size of a grid in any direction (Integer > 0; default: 128 (2-d), 32 (3-d))

  Note: `amr.max_grid_size` must be even, and a multiple of `amr.blocking_factor` at every level.

- `amr.blocking_factor`: grid size must be a multiple of this (Integer > 0; default: 2)

  Note: `amr.blocking_factor` at every level must be a power of 2 and the domain size must be a multiple of `amr.blocking_factor` at level 0.

- `amr.refine_grid_layout`: refine grids more if # of processors $>$ # of grids (0 if false, 1 if true; default: 1)

Note also that amr.n_error_buf, amr.max_grid_size and amr.blocking_factor can be read in as a single value which is assigned to every level, or as multiple values, one for each level.

As an example, consider:

```
amr.grid_eff = 0.9
amr.max_grid_size = 64
amr.blocking_factor = 32
```

The grid efficiency, amr.grid_eff, means that during the grid creation process, at least 90% of the cells in each grid at the level at which the grid creation occurs must be tagged cells. A higher grid efficiency means fewer cells at higher levels, but may result in the production of lots of small grids, which have inefficient cache and OpenMP performance and higher communication costs.

The amr.max_grid_size parameter means that the final grids will be no longer than 64 cells on a side at every level. Alternately, we could specify a value for each level of refinement as: amr.max_grid_size = 64 32 16, in which case our final grids will be no longer than 64 cells on a side at level 0, 32 cells on a side at level 1, and 16 cells on a side at level 2. The amr.blocking_factor means that all of the final grids will be multiples of 32 at all levels. Again, this can be specified on a level-by-level basis, like amr.blocking_factor = 32 16 8, in which case the dimensions of all the final grids will be multiples of 32 at level 0, multiples of 16 at level 1, and multiples of 8 at level 2.

4.2.4.1 Getting good performance

These parameters can have a large impact on the performance of PeleLM, so taking the time to experiment with is worth the effort. Having grids that are large enough to coarsen multiple levels in a V-cycle is essential for good multigrid performance.

4.2.4.2 How grids are created

The gridding algorithm proceeds in this order:

1. Grids are created using the Berger-Rigoutsos clustering algorithm modified to ensure that all new fine grids are divisible by amr.blocking_factor.
2. Next, the grid list is chopped up if any grids are larger than max_grid_size. Note that because amr.max_grid_size is a multiple of amr.blocking_factor the amr.blocking_factor criterion is still satisfied.
3. Next, if amr.refine_grid_layout = 1 and there are more processors than grids, and if amr.max_grid_size / 2 is a multiple of amr.blocking_factor, then the grids will be redefined, at each level independently, so that the maximum length of a grid at level ℓ, in any dimension, is amr.max_grid_size[ℓ] / 2.
4. Finally, if amr.refine_grid_layout = 1, and there are still more processors than grids, and if amr.max_grid_size / 4 is a multiple of amr.blocking_factor, then the grids will be redefined, at each level independently, so that the maximum length of a grid at level ℓ, in any dimension, is amr.max_grid_size[ℓ] / 4.

4.2.5 Simulation Time

There are two paramters that can define when a simulation ends:

- `max_step`: maximum number of level 0 time steps (Integer ≥ 0; default: -1)
- `stop_time`: final simulation time (Real ≥ 0; default: -1.0)

To control the number of time steps, you can limit by the maximum number of level 0 time steps (`max_step`) or by the final simulation time (`stop_time`), or both. The code will stop at whichever criterion comes first.

Note that if the code reaches `stop_time` then the final time step will be shortened so as to end exactly at `stop_time`, not past it.

As an example:

```
max_step  = 1000
stop_time  = 1.0
```

will end the calculation when either the simulation time reaches 1.0 or the number of level 0 steps taken equals 1000, whichever comes first.

4.2.6 Time Step

The following parameters affect the timestep choice:

- `ns.cfl`: CFL number (Real > 0 and ≤ 1; default: 0.8)
- `ns.init_shrink`: factor by which to shrink the initial time step (Real > 0 and ≤ 1; default: 1.0)
- `ns.change_max`: factor by which the time step can grow in subsequent steps (Real ≥ 1; default: 1.1)
- `ns.fixed_dt`: level 0 time step regardless of cfl or other settings (Real > 0; unused if not set)
- `ns.dt_cutoff`: time step below which calculation will abort (Real > 0; default: 0.0)

As an example, consider:

```
ns.cfl = 0.9
ns.init_shrink = 0.01
ns.change_max = 1.1
ns.dt_cutoff = 1.e-20
```

This defines the `cfl` parameter in Eq. **??** to be 0.9, but sets (via `init_shrink`) the first timestep we take to be 1% of what it would be otherwise. This allows us to ramp up to the hydrodynamic timestep at the start of a simulation. The `change_max` parameter restricts the timestep from increasing by more than 10% over a coarse timestep. Note that the time step can shrink by any factor; this only controls the extent to which it can grow. The `dt_cutoff` parameter will force the code to abort if the timestep ever drops below 10^{-20}. This is a safety feature—if the code hits such a small value, then something likely went wrong in the simulation, and by aborting, you won't burn through your entire allocation before noticing that there is an issue.

If we know what we are doing, then we can force a particular timestep:

```
ns.fixed_dt = 1.e-4
```

sets the level 0 time step to be 1.e-4 for the entire simulation, ignoring the other timestep controls. Note that if `ns.init_shrink` $\neq 1$ then the first time step will in fact be `ns.init_shrink` $\cdot$ `ns.fixed_dt`.

4.2.7 Restart Capability

PeleLM has a standard sort of checkpointing and restarting capability. In the inputs file, the following options control the generation of checkpoint files (which are really directories):

- `amr.check_file`: prefix for restart files (text; default: `chk`)
- `amr.check_int`: how often (by level 0 time steps) to write restart files (Integer > 0; default: -1)
- `amr.check_per`: how often (by simulation time) to write restart files (Real > 0; default: -1.0)

  Note that `amr.check_per` will write a checkpoint at the first timestep whose ending time is past an integer multiple of this interval. In particular, the timestep is not modified to match this interval, so you won't get a checkpoint at exactly the time you requested.
- `amr.restart`: name of the file (directory) from which to restart (Text; not used if not set)
- `amr.checkpoint_files_output`: should we write checkpoint files? (0 or 1; default: 1)

  If you are doing a scaling study then set `amr.checkpoint_files_output = 0` so you can test scaling of the algorithm without I/O.
- `amr.check_nfiles`: how parallel is the writing of the checkpoint files? (Integer ≥ 1; default: 64)

  See the Software Section for more details on parallel I/O and the `amr.check_nfiles` parameter.
- `amr.checkpoint_on_restart`: should we write a checkpoint immediately after restarting? (0 or 1; default: 0)

Note:

- You can specify both `amr.check_int` or `amr.check_per`, if you so desire; the code will print a warning in case you did this unintentionally. It will work as you would expect – you will get checkpoints at integer multiples of `amr.check_int` timesteps and at integer multiples of `amr.check_per` simulation time intervals.
- `amr.plotfile_on_restart` and `amr.checkpoint_on_restart` only take effect if `amr.regrid_on_restart` is in effect.

As an example,

```
amr.check_file = chk_run
amr.check_int = 10
```

means that restart files (really directories) starting with the prefix "chk_run" will be generated every 10 level-0 time steps. The directory names will be chk_run00000, chk_run00010, chk_run00020, etc.

If instead you specify

```
amr.check_file = chk_run
amr.check_per = 0.5
```

then restart files (really directories) starting with the prefix "chk_run" will be generated every 0.1 units of simulation time. The directory names will be chk_run00000, chk_run00043, chk_run00061, etc, where $t = 0.1$ after 43 level-0 steps, $t = 0.2$ after 61 level-0 steps, etc.

To restart from chk_run00061, for example, then set

```
amr.restart = chk_run00061
```

4.2.8 Controlling Plotfile Generation

The main output from PeleLM is in the form of plotfiles (which are really directories). The following options in the inputs file control the generation of plotfiles:

- amr.plot_file: prefix for plotfiles (text; default: "plt")
- amr.plot_int: how often (by level-0 time steps) to write plot files (Integer > 0; default: -1)
- amr.plot_per: how often (by simulation time) to write plot files (Real > 0; default: -1.0)

  Note that amr.plot_per will write a plotfile at the first timestep whose ending time is past an integer multiple of this interval. In particular, the timestep is not modified to match this interval, so you won't get a checkpoint at exactly the time you requested.
- amr.plot_vars: name of state variables to include in plotfiles (valid options: ALL, NONE or a list; default: ALL)
- amr.derive_plot_vars: name of derived variables to include in plotfiles (valid options: ALL, NONE or a list; default: NONE
- amr.plot_files_output: should we write plot files? (0 or 1; default: 1)

  If you are doing a scaling study then set amr.plot_files_output = 0 so you can test scaling of the algorithm without I/O.
- amr.plotfile_on_restart: should we write a plotfile immediately after restarting? (0 or 1; default: 0)
- amr.plot_nfiles: how parallel is the writing of the plotfiles? (Integer ≥ 1; default: 64)

  See the Software Section for more details on parallel I/O and the amr.plot_nfiles parameter.

All the options for amr.derive_plot_vars are kept in derive_lst in Pelelm_setup.cpp. Feel free to look at it and see what's there.

Some notes:

- You can specify both amr.plot_int or amr.plot_per, if you so desire; the code will print a warning in case you did this unintentionally. It will work as you would expect – you will get plotfiles at integer multiples of amr.plot_int timesteps and at integer multiples of amr.plot_per simulation time intervals.

As an example:

```
amr.plot_file = plt_run
amr.plot_int = 10
```

means that plot files (really directories) starting with the prefix "plt_run" will be generated every 10 level-0 time steps. The directory names will be plt_run00000, plt_run00010, plt_run00020, etc.

If instead you specify

```
amr.plot_file = plt_run
amr.plot_per = 0.5
```

then restart files (really directories) starting with the prefix "plt_run" will be generated every 0.1 units of simulation time. The directory names will be plt_run00000, plt_run00043, plt_run00061, etc, where $t = 0.1$ after 43 level-0 steps, $t = 0.2$ after 61 level-0 steps, etc.

4.2.9 Screen Output

There are several options that set how much output is written to the screen as PeleLM runs:

- amr.v: verbosity of Amr.cpp (0 or 1; default: 0)
- ns.v: verbosity of NavierStokesBase.cpp (0 or 1; default: 0)
- diffusion.v: verbosity of Diffusion.cpp (0 or 1; default: 0)
- mg.v: verbosity of multigrid solver (for gravity) (allow values: 0,1,2,3,4; default: 0)
- amr.grid_log: name of the file to which the grids are written (text; not used if not set)
- amr.run_log: name of the file to which certain output is written (text; not used if not set)
- amr.run_log_terse: name of the file to which certain (terser) output is written (text; not used if not set)
- amr.sum_interval: if > 0, how often (in level-0 time steps) to compute and print integral quantities (Integer; default: -1)

  The integral quantities include total mass, momentum and energy in the domain every ns.sum_interval level-0 steps. The print statements have the form

  ```
  TIME= 1.91717746 MASS= 1.792410279e+34
  ```

  for example. If this line is commented out then it will not compute and print these quanitities.

As an example:

```
amr.grid_log = grdlog
amr.run_log = runlog
```

Every time the code regrids it prints a list of grids at all relevant levels. Here the code will write these grids lists into the file `grdlog`. Additionally, every time step the code prints certain statements to the screen (if `amr.v` = 1), such as:

```
STEP = 1 TIME = 1.91717746 DT = 1.91717746
PLOTFILE: file = plt00001
```

The `run_log` option will output these statements into *runlog* as well.

Terser output can be obtained via:

```
amr.run_log_terse = runlogterse
```

This file, `runlogterse` differs from `runlog`, in that it only contains lines of the form

```
10  0.2  0.005
```

in which "10" is the number of steps taken, "0.2" is the simulation time, and "0.005" is the level-0 time step. This file can be plotted very easily to monitor the time step.

4.2.10 Other parameters

There are a large number of solver-specific runtime parameters. We describe these together with the discussion of the physics solvers in later chapters.

CHAPTER 5

Units and Constants

5.1 Units and Constants

We currently support only MKS units in PeleLM. All inputs and problem initialization should be specified in MKS. No internal conversions of units occur within the code, so the output must be interpreted appropriately.

Location	Variable	MKS
inputs file	**geometry.prob_lo** and **geometry.prob_hi**	m
Hydro Initialization	density	kg / m^3
Hydro Initialization	velocities	m/s
Hydro Initialization	temperature	K
Hydro Initialization	energies	J = kg $(m/s)^2$
Output	Pressure	kg $(m/s)^2$ / m^3
Output	Time	s

CHAPTER 6

Software Framework

6.1 Code structure

The code structure in the `PeleLM/` directory is as follows:

- `Exec/`: various problem implementations, including:
  - `FlameInABox/`: run directory for the flame-in-a-box
  - `DiffFlamex/`: run directory for the diffusion flame
- `Source/`: source code
- `doc/`: documentation, including:
  - `UsersGuide/`: you're reading this now!
- `tools/`: a catch-all for additional things you may need

6.2 An Overview of PeleLM

PeleLM is built upon the AMReX C++ framework. This provides high-level classes for managing an adaptive mesh refinement simulation, including the core data structures required in AMR calculations.

The PeleLM simulation begins in `IAMR/Source/main.cpp` where an instance of the AMReX `Amr` class is created:

```
Amr* amrptr = new Amr;
```

The initialization, including calling a problem's `initdata()` routine and refining the base grid occurs next through

```
amrptr->init(strt_time,stop_time);
```

And then comes the main loop over coarse timesteps until the desired simulation time is reached:

```
while ( amrptr->okToContinue()                                &&
       (amrptr->levelSteps(0) < max_step || max_step < 0) &&
       (amrptr->cumTime() < stop_time || stop_time < 0.0) )

{
   //
   // Do a timestep.
   //
   amrptr->coarseTimeStep(stop_time);
}
```

This uses the AMReX machinery to do the necessary subcycling in time, including synchronization between levels, to advance the level hierarchy forward in time.

6.2.1 Geometry class

6.2.2 ParmParse class

6.2.3 PeleLM Data Structures

6.2.3.1 State Data

The `StateData` class structure defined by AMReX is the data container used to store the field data associated with the state on a single AMR level during an PeleLM run. The entire state consists of a dynamic union, or hierarchy, of nested `StateData` objects. Periodic regrid operations modify the hierarchy, changing the shape of the data containers at the various levels according to user-specified criteria; new `StateData` objects are created for the affected levels, and are filled with the "best" (finest) available data at each location. Instructions for building and managing `StateData` are encapsulated in the AMReX class, `StateDescriptor`; as discussed later, a `StateDescriptor` will be created for each type of state field, and will include information about data centering, required grow cells, and instructions for transferring data between AMR levels during various synchronization operations.

In `IAMR/Source/NavieStokesBase.H`, the `enum StateType` defines the different state descriptors for PeleLM. These are setup during the run by code in `PeleLM_setup.cpp`, and include (but are not limited to):

- `State_Type`: the cell-centered thermodynamic state variables.
- `Press_Type`: the node-centered dynamic pressure field.
- `RhoYdot_Type`: the mass production rates for the chemical species.

Each StateData object has two MultiFabs, one each for old and new times, and can provide an interpolated copy of the state at any time between the two. Alternatively, can also access the data containers directly, for instance:

```
MultiFab& S_new = get_new_data(State_Type);
```

gets a pointer to the multifab containing the hydrodynamics state data at the new time (here State_Type is the enum defined in NavierStokesBase.H) (note that the classes NavierStokes and PeleLM are derived classes of NavierStokesBase).

MultiFab data is distributed in space at the granularity of each Box in its BoxArray. We iterate over MultiFabs using a special iterator, MFIter, which knows about the locality of the data—only the boxes owned by the processor will be included in the loop on each processor. An example loop (for the initialization, taken from code in PeleLM.cpp):

```
for (MFIter mfi(S_new); mfi.isValid(); ++mfi)
  {
     const Box& bx       = mfi.validbox();
     const int* lo       = bx.loVect();
     const int* hi       = bx.hiVect();

     if (! orig_domain.contains(bx)) {
        BL_FORT_PROC_CALL(CA_INITDATA,ca_initdata)
          (level, cur_time, lo, hi, ns,
           BL_TO_FORTRAN(S_new[mfi]), dx,
           gridloc.lo(), gridloc.hi());
     }
  }
```

Here BL_TO_FORTRAN is a special AMReX macro that converts the C++ multifab into a Fortran array, and BL_FORT_PROC_CALL is a AMReX macro that is used to interface with Fortran routines. ++mfi iterates to the next FArrayBox owned by the MultiFab, and mfi.isValid() returns false after we've reached the last box contained in the MultiFab, terminating the loop.

The corresponding Fortran function will look like: Need to write the Fortran version here Need to modernize C/F interface description/usages

6.2.4 Derived Variables

6.2.5 Error Estimators

6.2.6 Fortran Helper Modules

6.3 Setting Up Your Own Problem

To define a new problem, we create a new directory under bin/, and place in it a Prob_2d.f90 file (or Prob_3d.f90, depending on the dimensionality of the problem), a probdata.f90 file, the inputs and probin files, and a Make.package file that tells the build system what problem-specific

routines exist. The simplest way to get started is to copy these files from an existing problem. Here we describe how to customize your problem.

The purpose of these files is:

- `probdata.f90`: this holds the `probdata_module` Fortran module that allocates storage for all the problem-specific runtime parameters that are used by the problem (including those that are read from the `probin` file.
- `Prob_?d.f90`: this holds the main routines to initialize the problem and grid and perform problem-specific boundary conditions:
  - `probinit()`:

    This routine is primarily responsible for reading in the `probin` file (by defining the `&fortin` namelist and reading in an initial model (usually through the `model_parser_module`—see the `toy_convect` problem setup for an example). The parameters that are initialized here are those stored in the `probdata_module`.
  - `initdata()`:

    This routine will initialize the state data for a single grid. The inputs to this routine are:
    - `level`: the level of refinement of the grid we are filling
    - `time`: the simulation time
    - `lo()`, `hi()`: the integer indices of the box's *valid data region* lower left and upper right corners. These integers refer to a global index space for the level and identify where in the computational domain the box lives.
    - `nscal`: the number of scalar quantities—this is not typically used in PeleLM.
    - `state_l1`, `state_l2`, (`state_l3`): the integer indices of the lower left corner of the box in each coordinate direction. These are for the box as allocated in memory, so they include any ghost cells as well as the valid data regions.
    - `state_h1`, `state_h2`, (`state_h3`): the integer indices of the upper right corner of the box in each coordinate direction. These are for the box as allocated in memory, so they include any ghost cells as well as the valid data regions.
    - `state()`: the main state array. This is dimensioned as:

      ```
      double precision state(state_l1:state_h1,state_l2:state_h2,NVAR)
      ```

      (in 2-d), where `NVAR` comes from the `meth_params_module`.

      When accessing this array, we use the index keys provided by `meth_params_module` (e.g., `Density`) to refer to specific quantities
    - `delta()`: this is an array containing the zone width (Δx) in each coordinate direction: `delta(1)` $= \Delta x$, `delta(2)` $= \Delta y$, ...
    - `xlo()`, `xhi()`: these are the physical coordinates of the lower left and upper right corners of the *valid region* of the box. These can be used to compute the coordinates of the cell-centers of a zone as:

```
do j = lo(2), hi(2)
   y = xlo(2) + delta(2)*(dble(j-lo(2)) + 0.5d0)
   ...
```

(Note: this method works fine for the problem initialization stuff, but for routines that implement tiling, as discussed below, `lo` and `xlo` may not refer to the same corner, and instead coordinates should be computed using `problo()` from the `prob_params_module`.)

- `Prob_?d.f90`:

  These routines handle how PeleLM fills ghostcells *at physical boundaries* for specific data. These routines are registered in `HT_setup.cpp`, and called as needed. By default, they just pass the arguments through to `filcc`, which handles all of the generic boundary conditions (like reflecting, extrapolation, etc.). The specific '`fill`' routines can then supply the problem-specific boundary conditions, which are typically just Dirichlet boundary conditions (usually this means looking to see if the `bc()` flag at a boundary is `EXT_DIR`. The problem-specific code implementing these specific conditions should *follow* the `filcc` call.

  - `velfill`: This handles the boundary filling for velocity fields.
  - `denfill`: This handles the boundary filling for density field.
  - `rhohfill`: This handles the boundary filling for ρh field.
  - `tempfill`: This handles the boundary filling for temperature field.
  - `chemfill`: This handles boundary filling for ρY_i, for $i \in (1, n)$ and n is the number of chemical species.

  These routines take the following arguments:

  - `adv_l1`, `adv_l2`, (`adv_l3`): the indicies of the lower left corner of the box holding the data we are working on. These indices refer to the entire box, including ghost cells.
  - `adv_h1`, `adv_h2`, (`adv_h3`): the indicies of the upper right corner of the box holding the data we are working on. These indices refer to the entire box, including ghost cells.
  - `adv()`: the array of data whose ghost cells we are filling. Depending on the routine, this may have an additional index refering to the variable.

    This is dimensioned as:

    ```
    double precision adv(adv_l1:adv_h1,adv_l2:adv_h2)
    ```

  - `domlo()`, `domhi()`: the integer indices of the lower left and upper right corners of the valid region of the *entire domain*. These are used to test against to see if we are filling physical boundary ghost cells.

    This changes according to refinement level: level-0 will range from `0` to `amr.max_grid_size`, and level-n will range from `0` to `amr.max_grid_size` $\cdot \prod_n$ `amr.ref_ratio(n)`.

  - `delta()`: is the zone width in each coordinate direction, as in `initdata()` above.

- `xlo()`: this is the physical coordinate of the lower left corner of the box we are filling—including the ghost cells.

  Note: this is different than how `xlo()` was defined in `initdata()` above.

- `time`: the simulation time

- `bc()`: an array that holds the type of boundary conditions to enforce at the physical boundaries for `adv`.

  Sometimes it appears of the form `bc(:,:)` and sometimes `bc(:,:,:)`—the last index of the latter holds the variable index, i.e., density, pressure, species, etc.

  The first index is the coordinate direction and the second index is the domain face (`1` is low, `2` is hi), so `bc(1,1)` is the lower x boundary type, `bc(1,2)` is the upper x boundary type, `bc(2,1)` is the lower y boundary type, etc.

  To interpret the array values, we test against the quantities defined in `bc_types.fi` included in each subroutine, for example, `EXT_DIR`, `FOEXTRAP`, ... The meaning of these are explained below.

6.4 Boundaries

In AMReX, we are primarily concerned with enabling structured-grid computations. A key aspect of this is the use of "grow" cells around the "valid box" of cells over which we wish to apply stencil operations. Grow cells, filled properly, are conveniently located temporary data containers that allow us to separate the steps of data preparation (including communication, interpolation, or other complex manipulation) from stencil application. The steps that are required to fill grow cells depends on where the cells "live" in the computational domain.

6.4.1 Boundaries Between Grids and Levels

Most of our state data is cell-centered, and often the grow cells are as well. When the cells lie directly over cells of a neighboring box at the same AMR refinement level, these are "fine-fine" cells, and are filled by direct copy (including any MPI communication necessary to enable that copy). Note that fine-fine boundary also include grow cells that cover valid fine cells through a periodic boundary.

When the boundary between valid and grow cells is coincident with a coarse-fine boundary, these coarse-fine grow cells will hold cell-centered temporary data that generated by interpolation (in space and time) of the underlying coarse data. This operation requires auxiliary metadata to define how the interpolation is to be done, in both space and time. Importantly, the interpolation also requires that coarse data be well-defined over a time interval that brackets the time instant for which we are evaluating the grow cell value – this places requirements on how the time-integration of the various AMR levels are sequenced relative to eachother. In AMReX, the field data associated with the system state, as well as the metadata associated with inter-level transfers, is bundled (encapsulated) in a class called "StateData". The metadata is defined in `HT_setup.cpp` – search for `cell_cons_interp`, for example – which is "cell conservative interpolation", i.e., the data is cell-based (as opposed to node-based or edge-based) and the interpolation is such that the average

of the fine values created is equal to the coarse value from which they came. (This wouldn't be the case with straight linear interpolation, for example.) A number of interpolators are provided with AMReX and user-customizable ones can be added on the fly.

6.4.2 Physical Boundaries

The last type of grow cell exists at physical boundaries. These are special for a couple of reasons. First, the user must explicitly specify how they are to be filled, consistent with the problem being run. AMReX provides a number of standard condition types typical of PDE problems (reflecting, extrapolated, etc), and a special one that indicates external Dirichlet. In the case of Dirichlet, the user will supply a coded function to fill grow cells (discussed elsewhere in this document).

It is important to note that Dirichlet boundary data is to be specified as if applied on the edge of the cell bounding the domain. The array passed into the user boundary condition code is filled with cell-centered values in the valid region and in fine-fine, and coarse-fine grow cells. Additionally, grow cells for standard extrapolation and reflecting boundaries are pre-filled. The differential operators throughout PeleLM are aware of the special boundaries that are Dirichlet and wall-centered, and the stencils are adjusted accordingly.

For convenience, PeleLM provides a limited set of mappings from a physics-based boundary condition specification to a mathematical one that the code can apply. This set can be extended by adjusting the corresponding translations in `HT_setup.cpp`, but, by default, includes (See `AMReX/Src/C_AMRLib/amrlib/BC_TYPES.H` for more detail):

- *Outflow*:
  - velocity: `FOEXTRAP`
  - temperature: `FOEXTRAP`
  - scalars: `FOEXTRAP`
- *No Slip Wall with Adiabatic Temp*:
  - velocity: `EXT_DIR`, $u = v = 0$
  - temperature: `REFLECT_EVEN`, $dT/dt = 0$
  - scalars: `HOEXTRAP`
- *No Slip Wall with Fixed Temp*:
  - velocity: `EXT_DIR`, $u = v = 0$
  - temperature: `EXT_DIR`
  - scalars: `HOEXTRAP`
- *Slip Wall with Adiabatic Temp*:
  - velocity: `EXT_DIR`, $u_n = 0$; `HOEXTRAP`, u_t
  - temperature: `REFLECT_EVEN`, $dT/dn = 0$
  - scalars: `HOEXTRAP`

- *Slip Wall with Fixed Temp*:
  - velocity: `EXT_DIR`, $u_n = 0$
  - temperature: `EXT_DIR`
  - scalars: `HOEXTRAP`

The keywords used above are defined:

- `INT_DIR`: data taken from other grids or interpolated
- `EXT_DIR`: data specified on EDGE (FACE) of bndry
- `HOEXTRAP`: higher order extrapolation to EDGE of bndry
- `FOEXTRAP`: first order extrapolation from last cell in interior
- `REFLECT_EVEN`: $F(-n) = F(n)$ true reflection from interior cells
- `REFLECT_ODD`: $F(-n) = -F(n)$ true reflection from interior cells

6.4.3 The `FillPatchIterator`

A `FillPatchIterator` is a AMReX object tasked with the job of filling rectangular patches of state data, possibly including grow cells, and, if so, utilizing all the metadata discussed above that is provided by the user. Thus, a `FillPatchIterator` can only be constructed on a fully registered `StateData` object, and is the preferred process for filling grown platters of data prior to most stencil operations (e.g., explicit advection operators, which may require several grow cells). It should be mentioned that a `FillPatchIterator` fills temporary data via copy operations, and therefore does not directly modify the underlying state data. In the code, if the state is modified (e.g., via an advective "time advance", the new data must be copied explicitly back into the `StateData` containers.

For example, the following code demonstrates the calling sequence to create and use a `FillPatchIterator` for preparing a rectangular patch of data that includes the "valid region" plus `NUM_GROW` grow cells. Here, the valid region is specified as a union of rectangular boxes making up the box array underlying the `MultiFab S_new`, and `NUM_GROW` cells are added to each box in all directions to create the temporary patches to be filled. This is a parallel loop (the constructor is blocking over all processors while the data is filled); each processor then gets platters of data associated with the boxes from `S_new` that are local to it.

```
for (FillPatchIterator fpi(*this, S_new, NUM_GROW,
                           time, State_Type, strtComp, NUM_STATE);
      fpi.isValid(); ++fpi)
{
  // Get a reference to the temporary platter of grown data
  FArrayBox &state = fpi();

  // work on "state"
  ...
}
```

Here the FillPatchIterator fills the patch with data of type "State_Type" at time "time", starting with component strtComp and including a total of NUM_STATE components. state is a completely local data structure, and will be processed serially by the owning procesor. When the loop is terminated, the FillPatchIterator and temporary data platters are destroyed (though much of the metadata generated during the operation is cached internally for performance). Notice that since NUM_GROW can be any positive integer (i.e., that the grow region can extend over an arbitrary number of successively coarser AMR levels), this key operation can hide an enormous amount of code and algorithm complexity.

6.5 Parallel I/O

Both checkpoint files and plotfiles are actually folders containing subfolders: one subfolder for each level of the AMR hierarchy. The fundamental data structure we read/write to disk is a MultiFab, which is made up of multiple FAB's, one FAB per grid. Multiple MultiFabs may be written to each folder in a checkpoint file. MultiFabs of course are shared across CPUs; a single MultiFab may be shared across thousands of CPUs. Each CPU writes the part of the MultiFab that it owns to disk, but they don't each write to their own distinct file. Instead each MultiFab is written to a runtime configurable number of files N (N can be set in the inputs file as the parameter amr.checkpoint_nfiles and amr.plot_nfiles; the default is 64). That is to say, each MultiFab is written to disk across at most N files, plus a small amount of data that gets written to a header file describing how the file is laid out in those N files.

What happens is N CPUs each opens a unique one of the N files into which the MultiFab is being written, seeks to the end, and writes their data. The other CPUs are waiting at a barrier for those N writing CPUs to finish. This repeats for another N CPUs until all the data in the MultiFab is written to disk. All CPUs then pass some data to CPU 0 which writes a header file describing how the MultiFab is laid out on disk.

We also read MultiFabs from disk in a "chunky" manner, opening only N files for reading at a time. The number N, when the MultiFabs were written, does not have to match the number N when the MultiFabs are being read from disk. Nor does the number of CPUs running while reading in the MultiFab need to match the number of CPUs running when the MultiFab was written to disk.

Think of the number N as the number of independent I/O pathways in your underlying parallel filesystem. Of course a "real" parallel filesytem should be able to handle any reasonable value of N. The value -1 forces N to the number of CPUs on which you're running, which means that each CPU writes to a unique file, which can create a very large number of files, which can lead to inode issues.

6.6 Parallelization

AMReX uses a hybrid MPI + OpenMP approach to parallelism. The basic idea is that MPI is used to distribute individual boxes across nodes while OpenMP is used to distribute the work in local boxes to the cores within a node. The OpenMP approach in AMReX is optionally based on *tiling*

the box-based data structures. Both the tiling and non-tiling approaches to work distribution are discussed below.

6.6.1 AMReX's Non-Tiling Approach In C++

At the highest abstraction level, we have `MultiFab` (mulitple `FArrayBox`es). A `MultiFab` contains an array of `Box`es (a `Box` contains integers specifying the index space it covers), including `Box`es owned by other processors for the purpose of communication, an array of MPI ranks specifying which MPI processor owns each `Box`, and an array of pointers to `FArrayBox`es owned by this MPI processor. The real floating point data are stored for each `FArrayBox` as one-dimensional arrays, and can thus be passed across languages, e.g. to Fortran subroutines for processing. A typical usage of `MultiFab` is as follows,

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi) // Loop over boxes
{
  // Get the index space of this iteration
  const Box& box = mfi.validbox();

  // Get a reference to the FAB, which contains data and box
  FArrayBox& fab = mf[mfi];

  // Get double* of the FAB
  double* a = fab.dataPtr();

  // Get the index space for the data pointed by the double*
  // Note "abox" may have ghost cells, and is thus larger than
  // or equal to "box" obtained using mfi.validbox().
  const Box& abox = fab.box();

  // We can now pass the information to a Fortran routine,
  // which reshapes double* a into a multi-dimensional array
  // with dimensions specified by the information in "abox".
  // We will also pass "box", which specifies our "work" region.
}
```

A few comments about this code

- Here the iterator, `mfi`, will perform the loop only over the boxes that are local to the MPI task. If there are 3 boxes on the processor, then this loop has 3 iterations.

- `box` as returned from `mfi.validbox()` does not include ghost cells. We can get the indices of the valid zones as `box.loVect` and `box.hiVect`.

- Instead of getting the data pointer explicitly (via `fab.dataPtr()`), PeleLM often uses the preprocessor macro `BL_TO_FORTRAN(x)` (defined in `ArrayLim.H`) to substitute in the data pointer and the `lo` and `hi` indices of the multidimensional Fortran array (including ghost cells).

6.6.2 AMReX's Current Tiling Approach In C++

There are two types of tiling that people discuss. In *logical tiling*, the data storage in memory is unchanged from how we do things now in pure MPI. In a given box, the data region is stored contiguously). But when we loop in OpenMP over a box, the tiling changes how we loop over the data. The alternative is called *separate tiling*—here the data storage in memory itself is changed to reflect how the tiling will be performed. This is not considered in AMReX.

We have recently introduced logical tiling into parts of AMReX. Examples that demonstrate the syntax and usage can be found at `Tutorials/Tiling_C`, and `Src/LinearSolvers/C_CellMG/`. In our logical tiling approach, a box is logically split into tiles, and a `MFIter` loops over each tile in each box. Note that the non-tiling iteration approach can be considered as a special case of tiling with the tile size equal to the box size.

```
bool tiling = true;
for (MFIter mfi(mf,tiling); mfi.isValid(); ++mfi) // Loop over tiles
{
  // Get the index space of this iteration
  const Box& box = mfi.tilebox();

  // Get a reference to the FAB, which contains data and box
  FArrayBox& fab = mf[mfi];

  // Get double* of the FAB
  double* a = fab.dataPtr();

  // Get the index space for the data pointed by the double*.
  const Box& abox = fab.box();

  // We can now pass the information to a Fortran routine.
}
```

Note that the code is almost identical to the one in § 6.6.1. Some comments:

- The iterator now takes an extra argument to turn on tiling (set to `true`). There is another interface fo `MFIter` that can take an `IntVect` that explicitly gives the tile size in each coordinate direction.

  If we don't explictly specify the tile size at the loop, then the runtime parameter `fabarray.mfiter_tile_size` can be used to set it globally.

- `.validBox()` has the same meaning as in the non-tile approach, so we don't use it. Instead, we use `.tilebox()` to get the `Box` (and corresponding `lo` and `hi`) for the *current tile*, not the entire data region.

- When passing into the Fortran routine, we still use the index space of the entire fab (including ghost cells), as seen in the `abox` construction.

  The Fortran routine will declare a multidimensional array that is of the same size as the entire box, but only work on the index space identified by the tile-box (`box`).

Let us consider an example. Suppose there are four boxes—see Figure 6.1. The first box is divided into 4 logical tiles, the second and third are divided into 2 tiles each (because they are small), and

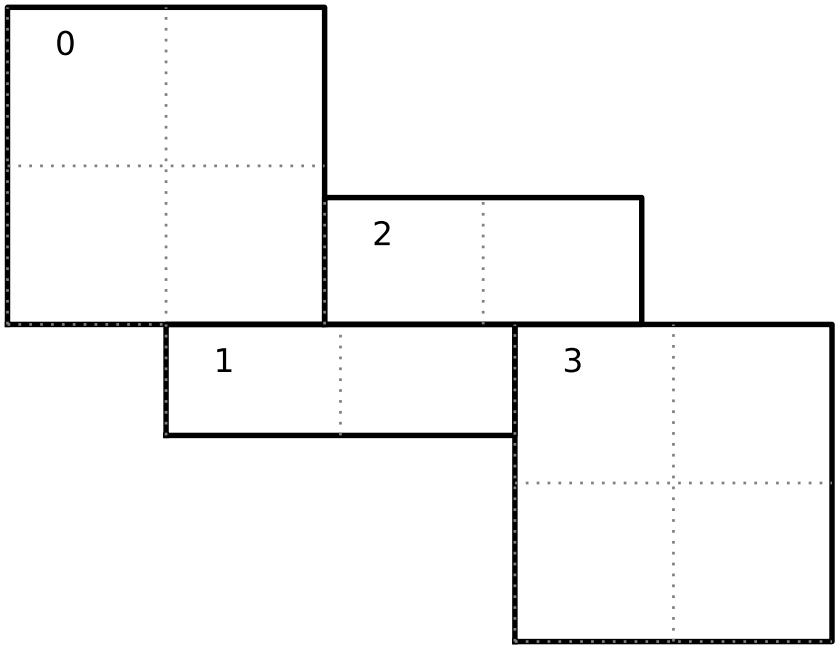

Figure 6.1: A simple domain showing 4 `Box`es labeled 0–3, and their tiling regions (dotted lines)

the fourth into 4 tiles. So there are 12 tiles in total. The difference between the tiling and non-tiling version are then:

- In the tiling version, the loop body will be run 12 times. Note that `tilebox` is different for each tile, whereas `fab` might be referencing the same object if the tiles belong to the same box.
- In the non-tiling version (by constructing `MFIter` without the optional second argument or setting to `false`), the loop body will be run 4 times because there are four boxes, and a call to `mfi.tilebox()` will return the traditional `validbox`. The non-tiling case is essentially having one tile per box.

Tiling provides us the opportunity of a coarse-grained approach for OpenMP. Threading can be turned on by inserting the following line above the `for (MFIter...)` line.

```
#pragma omp parallel
```

Assuming four threads are used in the above example, thread 0 will work on 3 tiles from the first box, thread 1 on 1 tile from the first box and 2 tiles from the second box, and so forth. Note that OpenMP can be used even when tiling is turned off. In that case, the OpenMP granularity is at the box level (and good performance would need many boxes per MPI task).

We also note that, independent of whether or not tiling is on, OpenMP threading can also be started within the function called inside the `MFIter` loop, rather than at the `MFIter` loop level. This was the original way that threading was done in PeleLM, but we are in the process of removing this approach in favor of tiling.

The tile size for the three spatial dimensions can be set by a parameter, e.g., `fabarray.mfiter_tile_size = 1024000 8 8`. A huge number like `1024000` will turn off tiling in that direction. As noted above, the `MFIter` constructor can also take an explicit tile size: `MFIter(mfi(mf,IntVect(128,16,32)))`.

Note that tiling can naturally transition from all threads working on a single box to each thread working on a separate box as the boxes coarsen (e.g., in multigrid).

The `MFIter` class provides some other useful functions:

```
mfi.validbox()       : The same meaning as before independent of tiling.
mfi.growntilebox(int): A grown tile box that includes ghost cells at box
                       boundaries only.  Thus the returned boxes for a
                       Fab are non-overlapping.
mfi.nodaltilebox(int): Returns non-overlapping edge-type boxes for tiles.
                       The argument is for direction.
mfi.fabbox()         : Same as mf[mfi].box().
```

Finally we note that tiling is not always desired or better. This traditional fine-grained approach coupled with dynamic scheduling is more appropriate for work with unbalanced loads, such as chemistry burning in cells by an implicit solver. Tiling can also create extra work in the ghost cells of tiles.

6.6.3 Practical Details in Working with Tiling

With tiling, the OpenMP is now all in C++, and not in Fortran for all modules except reactions and initdata. Note that the OpenMP pragma does not have a `for`—this is not used when working with an iterator.

It is the responsibility of the coder to make sure that the routines within a tiled region are safe to use with OpenMP. In particular, note that:

- tile boxes are non-overlapping
- the union of tile boxes completely cover the valid region of the fab
- Consider working with a node-centered MultiFab, `ugdnv`, and a cell=centered MultiFab, `s`:
  - with `mfi(s)`, the tiles are based on the cell-centered index space. If you have an 8×8 box, then and 4 tiles, then your tiling boxes will range from $0 \rightarrow 3$, $4 \rightarrow 7$.
  - with `mfiugdnv`, the tiles are based on nodal indices, so your tiling boxes will range from $0 \rightarrow 3$, $4 \rightarrow 8$.
- When updating routines to work with tiling, we need to understand the distinction between the index-space of the entire box (which corresponds to the memory layout) and the index-space of the tile.
  - In the C++ end, we pass (usually via the `BL_TO_FORTRAN()` macro) the `loVect` and `hiVect` of the entire box (including ghost cells). These are then used to allocate the array in Fortran as:

```
    double precision :: a(a_l1:a_h1, a_l2:a_h2, ...)
```

When tiling is used, we do not want to loop as `do a_l1, a_h1`, but instead we need to loop over the tiling region. The indices of the tiling region need to be passed into the Fortran routine separately, and they come from the `mfi.tilebox()` statement.

– In Fortran, when initializing an array to 0, do so only over the tile region, not for the entire box. For a Fortran array `a`, this means we cannot do:

```
a = 0.0
a(:,:,:,:) = 0.0
```

but instead must do:

```
a(lo(1):hi(1),lo(2):hi(2),lo(3):hi(3),:) = 0.0
```

where `lo()` and `hi()` are the index-space for the tile box returned from `mfi.tilebox()` in C++ and passed into the Fortran routine.

– Look at `r_old_s` in `Exec/DustCollapse/probdata.f90` as an example of how to declare a `threadprivate` variable—this is then used in `sponge_nd.f90`.

CHAPTER 7

AMR

Our approach to adaptive refinement in PeleLM uses a nested hierarchy of logically-rectangular grids with simultaneous refinement of the grids in both space and time. The integration algorithm on the grid hierarchy is a recursive procedure in which coarse grids are advanced in time, fine grids are advanced multiple steps to reach the same time as the coarse grids and the data at different levels are then synchronized.

During the regridding step, increasingly finer grids are recursively embedded in coarse grids until the solution is sufficiently resolved. An error estimation procedure based on user-specified criteria (described in Section 7.1) evaluates where additional refinement is needed and grid generation procedures dynamically create or remove rectangular fine grid patches as resolution requirements change.

A good introduction to the style of AMR used here is in Lecture 1 of the Adaptive Mesh Refinement Short Course at https://ccse.lbl.gov/people/jbb/index.html

7.1 Tagging for Refinement

PeleLM determines what zones should be tagged for refinement at the next regridding step by using a set of built-in routines that test on quantities such as the density and pressure and determining whether the quantities themselves or their gradients pass a user-specified threshold. This may then be extended if `amr.n_error_buf` > 0 to a certain number of zones beyond these tagged zones. This section describes the process by which zones are tagged, and describes how to add customized tagging criteria.

The routines for tagging cells are located in the `Prob_nd.f90` file in the `Src_nd` directory. The main routines are `denerror`, `temperror`, `presserror`, `velerror`. They refine based on density, temperature, pressure, and velocity, respectively. The same approach is used for all of them. As an

example, we consider the density tagging routine. There are four parameters that control tagging. If the density in a zone is greater than the user-specified parameter `denerr`, then that zone will be tagged for refinement, but only if the current AMR level is less than the user-specified parameter `max_denerr_lev`. Similarly, if the absolute density gradient between a zone and any adjacent zone is greater than the user-specified parameter `dengrad`, that zone will be tagged for refinement, but only if we are currently on a level below `max_dengrad_lev`. Note that setting `denerr` alone will not do anything; you'll need to set `max_dengrad_lev` >= 1 for this to have any effect.

All four of these parameters are set in the `&tagging` namelist in your `probin` file. If left unmodified, they default to a value that means we will never tag. The complete set of parameters that can be controlled this way is the following:

- `denerr`
- `max_denerr_lev`
- `dengrad`
- `max_dengrad_lev`
- `temperr`
- `max_temperr_lev`
- `tempgrad`
- `max_tempgrad_lev`
- `velerr`
- `max_velerr_lev`
- `velgrad`
- `max_velgrad_lev`
- `presserr`
- `max_presserr_lev`
- `pressgrad`
- `max_pressgrad_lev`
- `raderr`
- `max_raderr_lev`
- `radgrad`
- `max_radgrad_lev`

Since there are multiple algorithms for determining whether a zone is tagged or not, it is worthwhile to specify in detail what is happening to a zone in the code during this step. We show this in the following pseudocode section. A zone is tagged if the variable `itag = SET`, and is not tagged if `itag = CLEAR` (these are mapped to 1 and 0, respectively).

```
itag = CLEAR

for errfunc[k] from k = 1 ... N
    // Three possibilities for itag: SET or CLEAR or remaining unchanged
    call errfunc[k](itag)
end for
```

In particular, notice that there is an order dependence of this operation; if `errfunc[2] CLEAR`s a zone and then `errfunc[3] SET`s that zone, the final operation will be to tag that zone (and vice versa). In practice by default this does not matter, because the built-in tagging routines never explicitly perform a `CLEAR`. However, it is possible to overwrite the `Tagging_nd.f90` file if you want to change how `ca_denerror, ca_temperror`, etc. operate. This is not recommended, and if you do so be aware that `CLEAR`ing a zone this way may not have the desired effect.

We provide also the ability for the user to define their own tagging criteria. This is done through the Fortran function `set_problem_tags` in the `problem_tagging_*d.f90` files. This function is provided the entire state (including density, temperature, velocity, etc.) and the array of tagging status for every zone. As an example of how to use this, suppose we have a 3D Cartesian simulation where we want to tag any zone that has a density gradient greater than 10, but we don't care about any regions outside a radius $r > 75$ from the problem origin; we leave them always unrefined. We also want to ensure that the region $r \leq 10$ is always refined. In our `probin` file we would set `denerr = 10` and `max_denerr_lev = 1` in the `&tagging` namelist. We would also make a copy of `problem_tagging_3d.f90` to our work directory and set it up as follows:

```
subroutine set_problem_tags(tag,tagl1,tagl2,tagl3,tagh1,tagh2,tagh3, &
                            state,state_l1,state_l2,state_l3,state_h1,state_h2,state_h3,&
                            set,clear,&
                            lo,hi,&
                            dx,problo,time,level)

  use bl_constants_module, only: ZERO, HALF
  use prob_params_module, only: center
  use meth_params_module, only: URHO, UMX, UMY, UMZ, UEDEN, NVAR

  implicit none

  integer         ,intent(in   ) :: lo(3),hi(3)
  integer         ,intent(in   ) :: state_l1,state_l2,state_l3, &
                                    state_h1,state_h2,state_h3
  integer         ,intent(in   ) :: tagl1,tagl2,tagl3,tagh1,tagh2,tagh3
  double precision,intent(in   ) :: state(state_l1:state_h1, &
                                          state_l2:state_h2, &
                                          state_l3:state_h3,NVAR)
  integer         ,intent(inout) :: tag(tagl1:tagh1,tagl2:tagh2,tagl3:tagh3)
  double precision,intent(in   ) :: problo(3),dx(3),time
  integer         ,intent(in   ) :: level,set,clear

  double precision :: x, y, z, r

  do k = lo(3), hi(3)
     z = problo(3) + (dble(k) + HALF) * dx(3) - center(3)
     do j = lo(2), hi(2)
        y = problo(2) + (dble(j) + HALF) * dx(2) - center(2)
        do i = lo(1), hi(1)
           x = problo(1) + (dble(i) + HALF) * dx(1) - center(2)

           r = (x**2 + y**2 + z**2)**(HALF)

           if (r > 75.0) then
             tag(i,j,k) = clear
           elseif (r <= 10.0) then
             tag(i,j,k) = set
           endif
        enddo
     enddo
  enddo

end subroutine set_problem_tags
```

CHAPTER 8

Visualization

8.1 Controlling What's in the PlotFile

There are a few options that can be set at runtime to control what variables appear in the plotfile.

amr.plot_vars =

and

amr.derive_plot_vars =

are used to control which variables are included in the plotfiles. The default for **amr.plot_vars** is all of the state variables. The default for **amr.derive_plot_vars** is none of the derived variables. So if you include neither of these lines then the plotfile will contain all of the state variables and none of the derived variables.

If you want all of the state variables plus pressure, for example, then set

amr.derive_plot_vars = pressure

If you just want density and pressure, for example, then set

amr.plot_vars = density

amr.derive_plot_vars = pressure

8.2 yt

yt is a free and open-source software that provides data analysis and publication-level visualization tools. It is geared more for astrophysical simulation results, but may be useful for your purposes. As yt is script-based, it's not as easy to use as VisIt, and certainly not as easy as amrvis, but the images can be worth it! Here we do not flesh out yt, but give an overview intended to get a person started. Full documentation and explanations from which this section was adapted can be found at http://yt-project.org/doc/index.html.

yt can be installed by the following commands:

$ wget http://hg.yt-project.org/yt/raw/stable/doc/install_script.sh

$ bash install_script.sh

This installs yt in your current directory. To update ytin the future, simply do

$ yt update

in your "yt-hg" folder.

8.2.1 AMReX Data

yt was originally created for simple analysis and visualization of data from the Enzo code. Since, it has grown to include support for a variety of codes, including Castro (which is a AMReX-based code for compressible astrophysics). However, yt will still sometimes make assumptions, especially about data field names, that favor Enzo and cause errors with AMReX data. These problems can usually be avoided by taking care to specify the data fields desired in visualization. For example, Enzo's density field is called "Density," and is the default for many plotting mechanisms when the user does not specify the field. However, PeleLM does not have a field called "Density"; instead, the density field is called "density." If a user does not specify a field while plotting with PeleLM data, chances are that yt will try (and fail) to find "Density" and return an error. As you will see in the examples, however, there is a way to create your own fields from existing ones. You can use these derived fields as you would use any other field.

There are also a few imperatives when it comes to reading in your AMReX simulation data and associated information. First and foremost is that the inputs file for the simulation **must** exist in the same directory as where the plotfile directory is located, and it **must** be named "**inputs**." yt reads information from the inputs file such as the number of levels in the simulation run, the number of cells, the domain dimensions, and the simulation time. yt will also optionally parse the probin file for pertinent information if it is similarly included with the name "**probin**" in the same directory as the plotfile of interest. When specifying a plotfile as the data source for plots, you may simply call it by its directory name, rather than using the Header file as in VisIt. As a final caveat, yt requires the existence of the job_info file within the plotfile directory.

The following examples for yt were taken from the Castro user guide, and so have a strong astrophysics bent to them, but is still useful in the context of combustion. The only subtlety is that Castro works in CGS units rather than MKS.

8.2.2 Interacting with yt: Command Line and Scripting

yt is written completely in python (if you don't have python, yt will install it for you) and there are a number of different ways to interact with it, including a web-based gui. Here we will cover command-line yt and scripts/the python interactive prompt, but other methods are outlined on the yt webpage at http://yt-project.org/doc/interacting/index.html.

The first step in starting up yt is to activate the yt environment:

$ source $YT_DEST/bin/activate

From the command line you can create simple plots, perform simple volume renderings, print the statistics of a field for your data set, and do a few other things. Try $ `yt` to see a list of commands, and $ `yt <command> --help` to see the details of a command. The command line is the easiest way to get quick, preliminary plots – but the simplicity comes at a price, as yt will make certain assumptions for you. We could plot a projection of density along the x-axis for the plotfile (yt calls it a parameter file) plt_def_00020 by doing the following:

$ yt plot -p -a 0 -f density plt_def_00020

Or a temperature-based volume rendering with 14 contours:

$ yt render -f Temp --contours 14 plt_def_00020

Any plots created from the command line will be saved into a subfolder called "frames." The command line is nice for fast visualization without immersing yourself too much in the data, but usually you'll want to specify and control more details about your plots. This can be done either through scripts or the python interactive prompt. You can combine the two by running scripts within the interactive prompt by the command

>>> execfile('script.py')

which will leave you in the interactive prompt, allowing you to explore the data objects you've created in your script and debug errors you may encounter. While in the yt environment, you can access the interactive prompt by $ *python* or the shortcut

$ pyyt

Once you're in the yt environment and in a .py script or the interactive prompt, there are a couple of points to know about the general layout of yt scripting. Usually there are five sections to a yt script:

1. Import modules
2. Load parameter files and saved objects
3. Define variables
4. Create and modify data objects, image arrays, plots, etc. → this is the meat of the script
5. Save images and objects

Note that neither saving nor loading objects is necessary, but can be useful when the creation of these objects is time-consuming, which is often the case during identification of clumps or contours.

8.2.3 yt Basics

The first thing you will always want to do is to import yt:

>>> from yt.mods import *

Under certain circumstances you will be required to import more, as we will see in some of the examples, but this covers most of it, including all of the primary functions and data objects provided by yt. Next, you'll need yt to access the plotfile you're interested in analyzing. Remember, you must have the "inputs" file in the same folder:

>>> pf = load('plt_def_00020')

When this line is executed, it will print out some key parameters from the simulation. However, in order to access information about all of the fluid quantities in the simulation, we must use the "hierarchy" object. It contains the geometry of the grid zones, their parent relationships, and the fluid states within each one. It is easily created:

>>> pf.h

Upon execution, yt may print out a number of lines saying it's adding unknown fields to the list of fields. This is because PeleLM has different names for fields than what yt expects. We can see what fields exist through the commands

>>> print pf.h.field_list

>>> print pf.h.derived_field_list

There may not be any derived fields for the PeleLM data. We can find out the number of grids and cells at each level, the simulation time, and information about the finest resolution cells:

>>> pf.h.print_stats()

You can also find the value and location of the maximum of a field in the domain:

>>> value, location = pf.h.find_max('density')

The list goes on. A full list of methods and attributes associated with the heirarchy object (and most any yt object or function) can be accessed by the help function:

>>> help(pf.h)

You can also use >>> *dir()* on an object or function to find out which names it defines. Check the yt documentation for help. Note that you may not always need to create the hierarchy object. For example, before calling functions like find_max; yt will construct it automatically if it does not already exist.

8.2.4 Data Containers and Selection

Sometimes, you'll want to select, analyze, or plot only portions of your simulation data. To that end, yt includes a way to create data "containers" that select data based on geometric bounds or

fluid quantity values. There are many, including rays, cylinders, and clumps (some in the examples, all described in the documentation), but the easiest to create is a sphere, centered on the location of the maximum density cell we found above:

>>> my_data_container = pf.h.sphere(location, 5.0e4/pf['km'])

Here, we put the radius in units of kilometers using a conversion. When specifying distances in yt, the default is to use the simulation-native unit named "1", which is probably identical to one of the other units, like "m". The pf.h.print_stats() command lists available units. We can access the data within the container:

>>> print my_data_container['density']

>>> print my_data_container.quantities['Extrema'](['density', 'pressure'])

When the creation of objects is time-consuming, it can be convenient to save objects so they can be used in another session. To save an object as part of the `.yt` file affiliated with the heirarchy:

>>> pf.h.save_object(my_data_container, 'sphere_to_analyze_later')

Once it has been saved, it can be easily loaded later:

>>> sphere_to_analyze = pf.h.load_object('sphere_to_analyze_later')

8.2.5 Grid Inspection

yt also allows for detailed grid inspection. The hierarchy object possesses an array of grids, from which we can select and examine specific ones:

>>> print pf.h.grids

>>> my_grid = pf.h.grids[4]

Each grid is a data object that carries information about its location, parent-child relationships (grids within which it resides, and grids that reside within it, at least in part), fluid quantities, and more. Here are some of the commands:

>>> print my_grid.Level

>>> print my_grid_ActiveDimensions

>>> print my_grid.LeftEdge

>>> print my_grid.RightEdge

>>> print my_grid.dds

(dds is the size of each cell within the grid).

>>> print my_grid.Parent

>>> print my_grid.Children[2].LeftEdge

>>> print my_grid['Density']

You can examine which cells within the grid have been refined with the child_mask attribute, a representative array set to zero everywhere there is finer resolution.To find the fraction of your grid that isn't further refined:

>>>print my_grid.child_mask.sum()/float(my_grid.ActiveDimensions.prod())

Rather than go into detail about the many possibilities for plotting in yt, we'll provide some examples.

8.2.6 Example Scripts

In these examples, we investigate 3-D simulation data of two stars orbiting in the center of the domain, which is a box of sides 10^{10} *cm*.

```
# Pressure Contours
from yt.mods import *

pf = load('plt00020')

field = 'pressure'

pf.h

# AMReX fields have no inherent units, so we add them in, in the form of a raw string
# with some LaTeX-style formatting.
pf.field_info[field]._units = r'\rm{Ba}'

# SlicePlot parameters include: parameter file, axis, field, window width (effectively the
# x and y zoom), and fontsize. We can also create projections with ProjectionPlot().
p = SlicePlot(pf, 'z', field, width=((5.0e9, 'cm'), (3.0e9, 'cm')),
              fontsize=13)

# Zlim is the range of the colorbar. In other words, the range of the data we want to display.
# Names for many colormaps can be found at wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps.
p.set_zlim(field, 2.85e13, 2.95e13)

p.set_cmap(field, 'jet')

# Here we add 5 density contour lines within certain limits on top of the image. We overlay
# our finest grids with a transparency of 0.2 (lower is more transparent). We add a quiver
# plot with arrows every 16 pixels with x_velocity in the x-direction and y_velocity in
# the y-direction. We also mark the center with an 'x' and label one of our stars.
p.annotate_contour('density', clim=(1.05e-4, 1.16e-4), ncont=5, label=False)

p.annotate_grids(alpha=0.2, min_level=2)

p.annotate_quiver('x_velocity', 'y_velocity', factor=16)

p.annotate_marker([5.0e9, 5.0e9], marker='x')

p.annotate_point([5.95e9, 5.1e9], 'Star!')

# This saves the plot to a file with the given prefix. We can alternatively specify
# the entire filename.
p.save('contours.press_den_')
```

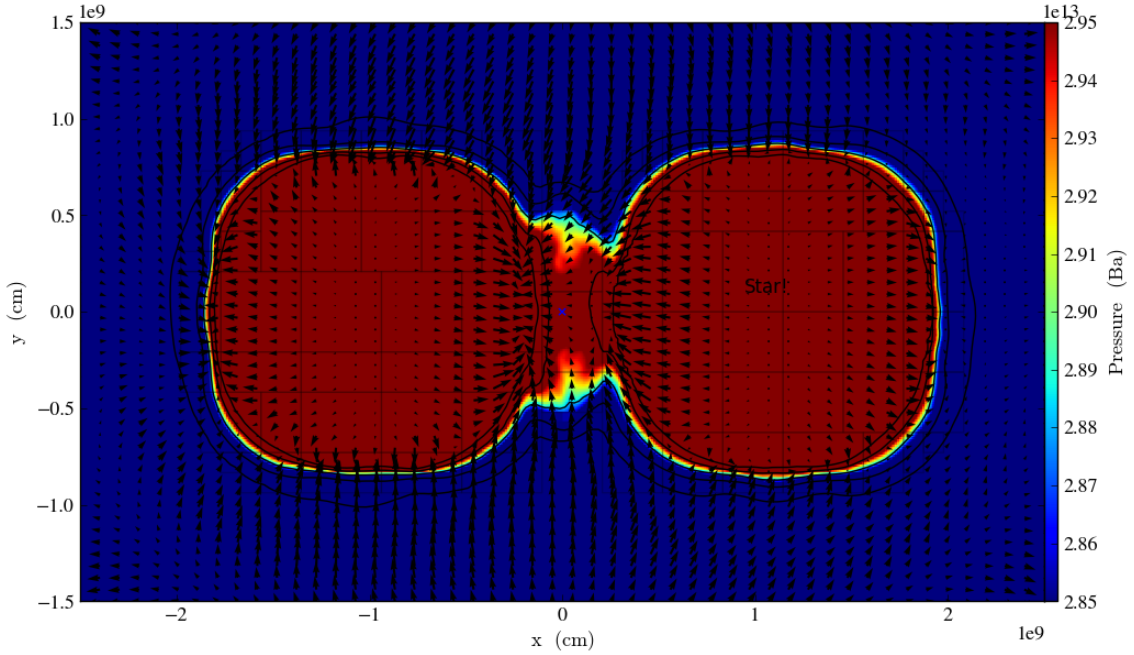

Figure 8.1: Pressure slice with annotations

```
#————————————

# Volume Rendering
from yt.mods import *

pf = load('plt00020')

field = 'pressure' dd = pf.h.all_data()

# We take the log of the extrema of the pressure field, as well as a couple other interesting
# value ranges we'd like to visualize.
h_mi, h_ma = dd.quantities['Extrema'](field)[0]

h_mi, h_ma = np.log10(h_mi), np.log10(h_ma)

s_mi, s_ma = np.log10(2.90e13), np.log10(3.10e13)

pf.h

# We deal in terms of logarithms here because we have such a large range of values.
# It can make things easier, but is not necessary.
pf.field_info[field].take_log=True

# This is what we use to visualize volumes. There are a couple of other, more complex
# ways. We set the range of values we're interested in and the number of bins in the
# function. Make sure to have a lot of bins if your data spans many orders of magnitude!
# Our raw data ranges from about 10^13 to 10^22.
tf = ColorTransferFunction((h_mi-1, h_ma+1), nbins=1.0e6)

# Here we add several layers to our function, either one at a time or in groups. We
# specify the value-center and width of the layer. We can manipulate the color by
```

```
# individually setting the colormaps and ranges to spread them over. We can also
# change the transparency, which will usually take some time to get perfect.
tf.sample_colormap(np.log10(2.0e21), 0.006, col_bounds=[h_mi,h_ma],
                   alpha=[27.0], colormap='RdBu_r')
tf.sample_colormap(np.log10(2.0e19), 0.001, col_bounds=[h_mi,h_ma],
                   alpha=[5.5], colormap='RdBu_r')
tf.add_layers(6, mi=np.log10(2.95e13), ma=s_ma,
              col_bounds=[s_mi,s_ma],
              alpha=19*na.ones(6,dtype='float64'), colormap='RdBu_r')
tf.sample_colormap(np.log10(2.95e13), 0.000005, col_bounds=[s_mi,s_ma],
                   alpha=[13.0], colormap='RdBu_r')
tf.sample_colormap(np.log10(2.90e13), 0.000007, col_bounds=[s_mi,s_ma],
                   alpha=[11.5], colormap='RdBu_r')
tf.sample_colormap(np.log10(2.85e13), 0.000008, col_bounds=[s_mi,s_ma],
                   alpha=[9.5], colormap='RdBu_r')

# By default each color channel is only opaque to itself. If we set grey_opacity=True,
# this is no longer the case. This is good to use if we want to obscure the inner
# portions of our rendering. Here it only makes a minor change, as we must set our
# alpha values for the outer layers higher to see a strong effect.
tf.grey_opacity=True

# Volume rendering uses a camera object which centers the view at the coordinates we've
# called 'c.' 'L' is the normal vector (automatically normalized) between the camera
# position and 'c,' and 'W' determines the width of the image—again, like a zoom.
# 'Nvec' is the number of pixels in the x and y directions, so it determines the actual
# size of the image.
c = [5.0e9, 5.0e9, 5.0e9]
L = [0.15, 1.0, 0.40]
W = (pf.domain_right_edge - pf.domain_left_edge)*0.5
Nvec = 768

# 'no_ghost' is an optimization option that can speed up calculations greatly, but can
# also create artifacts at grid edges and affect smoothness. For our data, there is no
# speed difference, so we opt for a better-looking image.
cam = pf.h.camera(c, L, W, (Nvec,Nvec), transfer_function = tf,
                  fields=[field], pf=pf, no_ghost=False)

# Obtain an image! However, we'll want to annotate it with some other things before
# saving it.
im = cam.snapshot()
```

 # Here we draw a box around our stars, and visualize the gridding of the top two levels.
*# Note that draw_grids returns a new image while draw_box does not. Also, add_*
 # background_color in front of draw_box is necessary to make the box appear over
 # blank space (draw_grids calls this internally). For draw_box we specify the left
 # (lower) and right(upper) bounds as well its color and transparency.
im.add_background_color('black', inline=True)

cam.draw_box(im, np.array([3.0e9, 4.0e9, 4.0e9]),

np.array([7.0e9, 6.0e9, 6.0e9]), np.array([1.0, 1.0, 1.0, 0.14]))

im = cam.draw_grids(im, alpha=0.12, min_level=2)

im = cam.draw_grids(im, alpha=0.03, min_level=1, max_level=1)

 # 'im' is an image array rather than a plot object, so we save it using a different
 # function. There are others, such as 'write_bitmap.'
im.write_png('pressure_shell_volume.png')

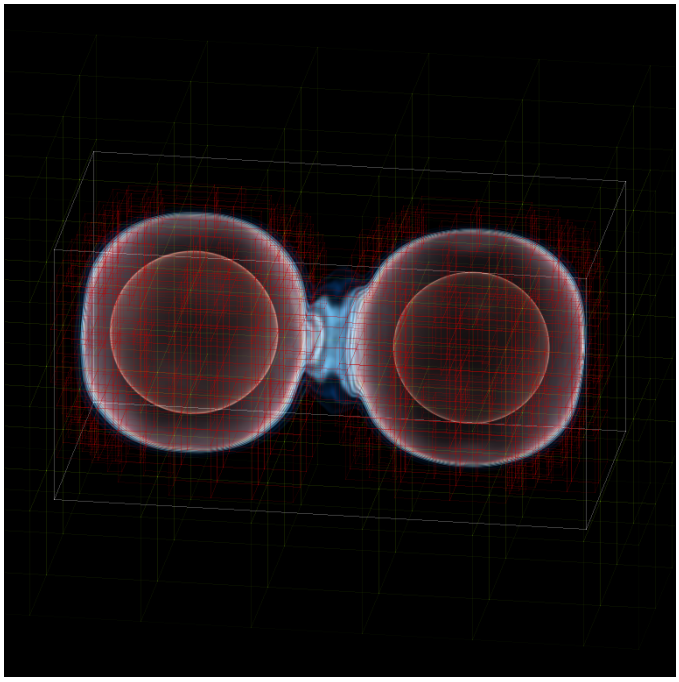

Figure 8.2: Volume rendering

#————————————

 # Isocontour Rendering
 # Here we extract isocontours using some extra modules and plot them using matplotlib.
from mpl_toolkits.mplot3d import Axes3D

from mpl_toolkits.mplot3d.art3d import Poly3DCollection

import matplotlib.pyplot as plt

from yt.mods import *

```
pf = load('plt00020')

field = 'pressure'

field_weight = 'magvel'

contour_value = 2.83e13

domain = pf.h.all_data()

# This object identifies isocontours at a given value for a given field. It returns
# the vertices of the triangles in that isocontour. It requires a data source, which
# can be an object—but here we just give it all of our data. Here we find a pressure
# isocontour and color it the magnitude of velocity over the same contour.
surface = pf.h.surface(domain, field, contour_value)

colors = apply_colormap(np.log10(surface[field_weight]), cmap_name='RdBu')

fig = plt.figure()

ax = fig.gca(projection='3d')

p3dc = Poly3DCollection(surface.triangles, linewidth=0.0)

p3dc.set_facecolors(colors[0,:,:]/255.)

ax.add_collection(p3dc)

# By setting the scaling on the plot to be the same in all directions (using the x scale),
# we ensure that no warping or stretching of the data occurs.
ax.auto_scale_xyz(surface.vertices[0,:], surface.vertices[0,:],

                  surface.vertices[0,:])

ax.set_aspect(1.0)

plt.savefig('pres_magvel_isocontours.png')

#————————————

#1-D and 2-D Profiles
# Line plots and phase plots can be useful for analyzing data in detail.
from yt.mods import *

pf = load('plt00020')

pf.h

# Just like with the pressure_contours script, we can set the units for fields that
# have none.
pf.field_info['magvel']._units = r'\rm{cm}/\rm{s}'

pf.field_info['kineng']._units = r'\rm{ergs}'

# We can create new fields from existing ones. ytassumes all units are in cgs, and
# does not do any unit conversions on its own (but we can make it). Creating new fields
# requires us to define a function that acts on our data and returns the new data,
# then call add_field while supplying the field name, the function the data comes from,
# and the units. Here, we create new fields simply to rename our data to make the plot
```

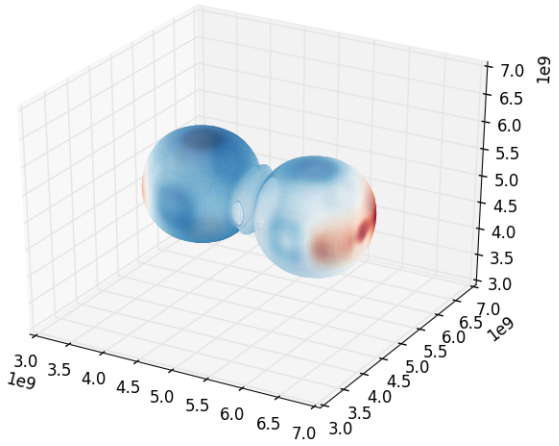

Figure 8.3: Pressure isocontour rendering colored with velocity magnitude

```
# look prettier.
def _newT(field, data):
    return data['t']
add_field('X', function=_newT, units=r'\rm{domain} \rm{fraction}')
def _newDen(field, data):
    return data['density']
add_field('Density', function=_newDen, units=r'\rm{g}/\rm{cm}^{3}')

# PlotCollections are one of the most commonly used tools in yt, alongside SlicePlots and
# ProjectionPlots. They are useful when we want to create multiple plots from the same
# parameter file, linked by common characteristics such as the colormap, its bounds, and
# the image width. It is easy to create 1-D line plots and 2-D phase plots through a
# PlotCollection, but we can also create thin projections and so on. When we create a
# PlotCollection, it is empty, and only requires the parameter file and the 'center' that
# will be supplied to plots like slices and sphere plots.
pc = PlotCollection(pf, 'c')

# Now we add a ray—a sample of our data field along a line between two points we define
# in the function call.
ray = pc.add_ray([0.0, 5.0e9, 5.0e9],[1.e10, 5.0e9, 5.0e9], 'magvel')

# This is where our derived fields come in handy. Our ray is drawn along the x-axis
# through the center of the domain, but by default the fraction of the ray we have gone
# along is called 't.' We now have the same data in another field we called 'X,' whose
# name makes more sense, so we'll reassign the ray's first field to be that. If we wanted,
(# we could also reassign names to 'magvel' and 'kineng.'
ray.fields = ['X', 'magvel']
```

```
# Next, we'll create a phase plot. The function requires a data source, and we can't
# just hand it our parameter file, but as a substitute we can quickly create an object
# that spans our entire domain (or use the method in the isocontour example). The
# specifications of the region (a box) are the center, left bound, and right bound.
region = pf.h.region([5.0e9, 5.0e9, 5.0e9], [0.0, 0.0, 0.0],
                     [1.0e10, 1.0e10, 1.0e10])

# The phase object accepts a data source, fields, a weight, a number of bins along both
# axes, and several other things, including its own colormap, logarithm options,
# normalization options, and an accumulation option. The first field is binned onto
# the x-axis, the second field is binned onto the y-axis, and the third field is
# binned with the colormap onto the other two. Subsequent fields go into an underlying
# profile and do not appear on the image.
phase = pc.add_phase_object(region, ['Density', 'magvel','kineng'], weight=None,
                            x_bins=288, y_bins=288)

pc.save('profile')

#————————————

#Off-Axis Projection
# If we don't want to take a projection (this can be done for a slice as well) along
# one of the coordinate axes, we can take one from any direction using an
# OffAxisProjectionPlot. To accomplish the task of setting the view up, the plot
# requires some of the same parameters as the camera object: a normal vector, center,
# width, and field, and optionally we can set no_ghost (default is False). The normal
# vector is automatically normalized as in the case of the camera. The plot also
# requires a depth—that is, how much data we want to sample along the line of sight,
# centered around the center. In this case 'c' is a shortcut for the domain center.
pf = load('plt00020')

field = 'density'

L = [0.25, 0.9, 0.40]

plot = OffAxisProjectionPlot(pf, L, field, center='c',
                             width=(5.0e9, 4.0e9), depth=3.0e9)

# Here we customize our newly created plot, dictating the font, colormap, and title.
# Logarithmic data is used by default for this plot, so we turn it off.
plot.set_font({'family':'Bitstream Vera Sans', 'style':'italic',
               'weight':'normal', 'size':14, 'color':'red'})

plot.set_log(field, False)

plot.set_cmap(field, 'jet')

plot.annotate_title('Off-Axis Density Projection')

# The actual size of the image can also be set. Note that the units are in inches.
plot.set_window_size(8.0)
```

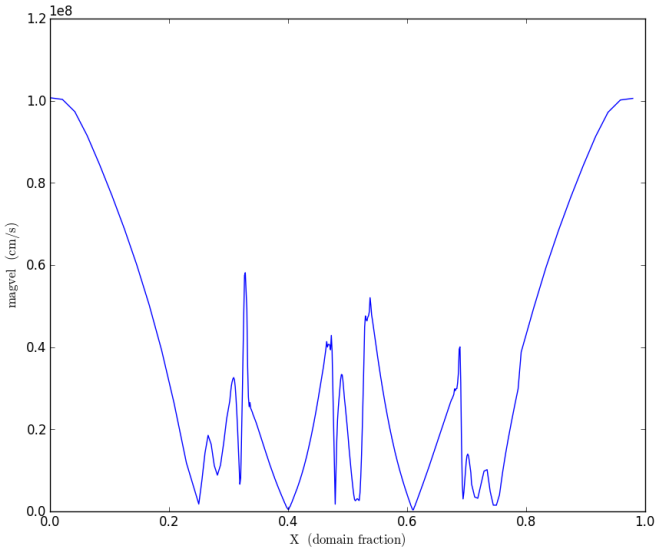

Figure 8.4: 1-D velocity magnitude profile

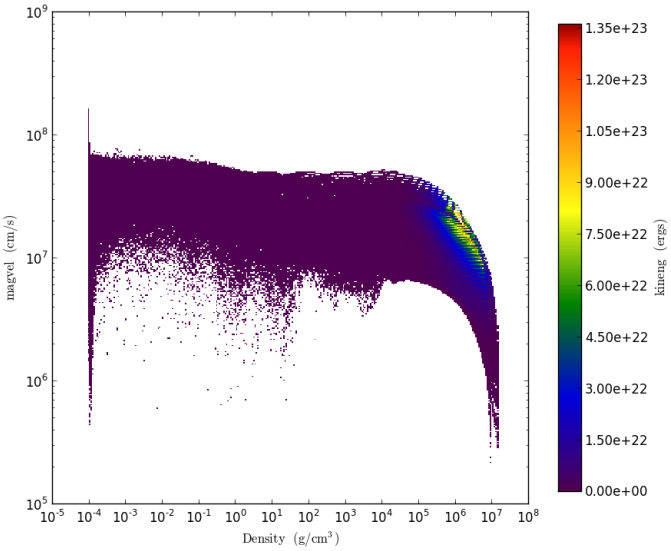

Figure 8.5: Density/velocity magnitude/kinetic energy phase plot

plot.save(‘off_axis_density’)

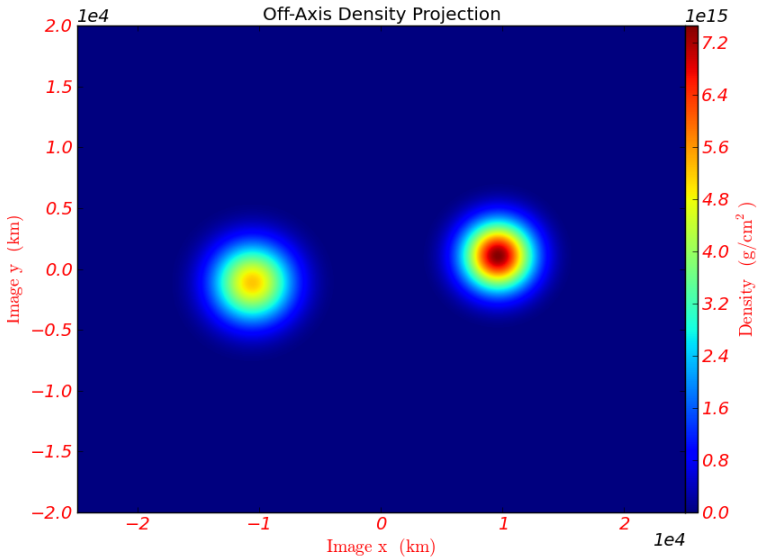

Figure 8.6: Off-axis density projection

References

[1] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. 142:1–46, 1998.

[2] A. S. Almgren, J. B. Bell, and W. Y. Crutchfield. Approximate projection methods: Part I. Inviscid analysis. *SIAM J. Sci. Comput.*, 22(4):1139–59, 2000.

[3] A. S. Almgren, J. B. Bell, and W. G. Szymczak. A numerical method for the incompressible Navier-Stokes equations based on an approximate projection. *SIAM J. Sci. Comput.*, 17(2):358–369, March 1996.

[4] Andrew Aspden, Nikos Nikiforakis, Stuart Dalziel, and John Bell. Analysis of implicit les methods. *Communications in Applied Mathematics and Computational Science*, 3(1):103–126, 2008.

[5] J. B. Bell, P. Colella, and L. H. Howell. An efficient second-order projection method for viscous incompressible flow. In *Proceedings of the Tenth AIAA Computational Fluid Dynamics Conference*, pages 360–367. AIAA, June 1991.

[6] J. B. Bell and D. L. Marcus. A second-order projection method for variable-density flows. 101:334–348, 1992.

[7] Marsha Berger and Isidore Rigoutsos. An algorithm for point clustering and grid generation. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(5):1278–1286, 1991.

[8] M. S. Day and J. B. Bell. Numerical simulation of laminar reacting flows with complex chemistry. *Combust. Theory Modelling*, 4(4):535–556, 2000.

[9] A Ern and V Giovangigli. Eglib: A general-purpose fortran library for multicomponent transport property evaluation. *Manual of EGlib version*, 3:12, 2004.

[10] Evatt R Hawkes, Ramanan Sankaran, James C Sutherland, and Jacqueline H Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. In *Journal of Physics: Conference Series*, volume 16, page 65. IOP Publishing, 2005.

[11] Randall J LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[12] A. Nonaka, J. B. Bell, M. S. Day, C. Gilet, A. S. Almgren, and M. L. Minion. A deferred correction coupling strategy for low Mach number flow with complex chemistry. *Combust. Theory Modelling*, 16(6):1053–1088, 2012.

[13] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. An adaptive projection method for unsteady low-Mach number combustion. *Comb. Sci. Tech.*, 140:123–168, 1998.

[14] Xiaoqing You, Fokion N Egolfopoulos, and Hai Wang. Detailed and simplified kinetic models of n-dodecane oxidation: The role of fuel cracking in aliphatic hydrocarbon combustion. *Proceedings of the Combustion Institute*, 32(1):403–410, 2009.