PeleLM Documentation

Release 2018.10

J.B. Bell, M.S. Day, E. Motheau, D. Graves, M. Henry de Frahan, R.

Sep 08, 2021

Documentation contents:

PeleLM is an adaptive-mesh low Mach number hydrodynamics code for reacting flows. *PeleLM* has an official project homepage, and can be obtained via GitHub. If you need help or have questions, please join the users forum. The documentation pages appearing here are distributed with the code in the `Docs` folder as "restructured text" files. The html is built automatically with certain pushes to the *PeleLM* GibHub repository and are maintained online by ReadTheDocs. A local version can also be built as follows

```
cd ${PELELM_DIR}/Docs
make html
```

where `PELELM_DIR` is the location of your clone of the *PeleLM* repository. To view the local pages, point your web browser at the file `${PELELM_DIR}/Docs/build/html/index.html`.

Current docs build status on ReadTheDocs:

CHAPTER 1

PeleLM Quickstart

PeleLM was created in 2017 by renaming *LMC*, the low Mach code from CCSE, and is built on the *AMReX* library, the *AMReX-Hydro* set of advection schemes, the *IAMR* code and the *PelePhysics* chemistry and thermodynamics library. For the impatient, the following summarizes how to obtain *PeleLM* and all the supporting software required, and how to build and run a simple case in order to obtain a first set of results. A thorough discussion of the model equations, and time stepping algorithms in *PeleLM* is given in *The PeleLM Model*. More details about the make system are given in *Building with GNU Make*. Parameters provided for runtime control of *PeleLM* are discussed in *PeleLM control*. Visualization of the results from *PeleLM* is discussed in *Visualization*.

1.1 Obtaining *PeleLM*

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

Then, there are two options to obtain *PeleLM* and its dependencies:

1.1.1 1. PeleProduction

***PeleProduction* enables the user to obtain a consistent version of *PeleLM* and all its dependencies** with a single git clone (from the user). This is the prefered option when one wants to use *PeleLM*

but do not intend to make development into the code. More information on *PeleProduction* can be found on the GitHub page.

a. Download the *PeleProduction* repository and :

```
git clone https://github.com/AMReX-Combustion/PeleProduction.git

cd PeleProduction
```

b. The first time you do this, you will need to tell git that there are submodules. Git will look at the `.gitmodules` file in this branch and use that :

```
cd Submodules
git submodule init
git submodule update
```

c. Finally, get into the FlameSheet folder of the *PeleLM* submodule:

```
cd PeleLM/Exec/RegTests/FlameSheet
```

1.1.2 2. Individual repositories

Alternatively, all the individual dependencies of *PeleLM* can be obtained independently. The user then needs to provide environment variables for each of *AMReX*, *IAMR*, *AMReX-Hydro*, *PelePhysics* and *PeleLM* installation path. This method is intended for users wanting to modify the *PeleLM* source code and who are more comfortable with maintaining up-to-date the four repositories.

a. 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*:

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

b. 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`. Then switch to the `development` branch of IAMR:

```
cd IAMR
git checkout -b development origin/development
```

c. Download the *AMReX-Hydro* repository by typing:

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

This will create a folder called `AMReX-Hydro/` on your machine. Set the environment variable, `AMREX_HYDRO_HOME`.

d. Clone the *PeleLM* and *PelePhysics* repositories:

```
git clone git@github.com:AMReX-Combustion/PeleLM.git
git clone git@github.com:AMReX-Combustion/PelePhysics.git
```

This will create folders called `PeleLM` and `PelePhysics` on your machine. Set the environment variables, `PELELM_HOME` and `PELE_PHYSICS_HOME`, respectively to where you put these.

d. Periodically update each of these repositories by typing `git pull` within each repository.

e. Finally, get into the `FlameSheet` folder of *PeleLM* :

```
cd PeleLM/Exec/RegTests/FlameSheet
```

1.2 Building *PeleLM*

In *PeleLM* each different problem setup is stored in its own sub-folder under `$(PELELM_HOME)/Exec/`, and a local version of the *PeleLM* executable is built directly in that folder (object libraries are not used to manage *AMReX* and the application code). In the following, we step through building a representative *PeleLM* executable.

1. Regardless of which path you decided to choose in order to get the *PeleLM* code and its dependencies, you should be now be in the `FlameSheet` folder. If you have chosen Option 2 to get the *PeleLM* sources, you have already set the environement variable necessary to compile the executable. If you have chosen the first option, you now have to modify the `GNUmakefile` to ensure that the variable `TOP` define on the first line points to the `Submodules` folder of *PeleProduction* :

```
TOP = /path/to/PeleProduction/Submodules
```

such that the following lines provide path to *PeleLM* and its dependencies. Note that an absolute path in needed.

2. Edit the `GNUmakefile` to ensure that the following are set:

```
DIM = 2
COMP = gnu (or your favorite C++/F90 compiler suite)
DEBUG = FALSE
USE_MPI = FALSE
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. Note that for centers managing their enviroments with "modules", the programming environment determining your available compiler should agree with your choice of `COMP` in the `GNUmakefile` (e.g., `PrgEnv-gnu` module requires `COMP=gnu`).

3. Start by building the Sundials Third Party Library used to integrate the chemistry:

```
make TPL
```

   and finally build *PeleLM* executable:

```
make
```

If successful, the resulting executable name will look something like `PeleLM2d.gnu.ex`. Depending on your compilation option the actual name of the executable might vary (including `MPI`, or `DEBUG`, ...).

1.3 Running *PeleLM*

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

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

2. While running, *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 collection 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 the AMReX-compatible readers context to the field data.

1.4 Visualization of the results

There are several options for visualizing the data. The popular packages *Vis-It* and *Paraview* support 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, however there are limited visualization tools available in *Amrvis*, so most users make use of multiple tools depending on their needs.

For more information on how to use *Amrvis* and *VisIt*, refer to the *AMReX* User's guide in the *AMReX* git repository for download/build/usage instructions.

CHAPTER 2

The *PeleLM* Model

In this section, we present the actual model that is evolved numerically by *PeleLM*, and the numerical algorithms to do it. There are many control parameters to customize the solution strategy and process, and in order to actually set up and run specific problems with *PeleLM*, the user must specific the chemical model, and provide routines that implement initial and boundary data and refinement criteria for the adaptive mesh refinement. We discuss setup and control of *PeleLM* in later sections.

2.1 Overview of *PeleLM*

PeleLM evolves chemically reacting low Mach number flows with block-structured adaptive mesh refinement (AMR). The code depends upon the AMReX library to provide the underlying data structures, and tools to manage and operate on them across massively parallel computing architectures. *PeleLM* also borrows heavily from the source code and algorithmic infrastructure of the IAMR. *IAMR* implements an AMR integration for the variable-density incompressible Navier-Stokes equations. *PeleLM* extends *IAMR* to include complex coupled models for generalized thermodynamic relationships, multi-species transport and chemical reactions. The core algorithms in *PeleLM* (and *IAMR*) are described in the following 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, *Combust. Theor. Model.*, **22** (1) 156-184 (2018)
- *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 Model*, **16** (6) 1053-1088 (2012)
- *Numerical Simulation of Laminar Reacting Flows with Complex Chemistry*, M. S. Day and J. B. Bell, *Combust. Theory Model* **4** (4) 535-556 (2000)
- *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** 123-168 (1998)
- *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** 1-46 (1998)

2.2 The low Mach number flow equations

PeleLM solves the reacting Navier-Stokes flow equations in the *low Mach number* regime, where the characteristic fluid velocity is small compared to the sound speed, and the effect of acoustic wave propagation is unimportant to the overall dynamics of the system. Accordingly, acoustic wave propagation can be mathematically removed from the equations of motion, allowing for a numerical time step based on an advective CFL condition, and this leads to an increase in the allowable time step of order $1/M$ over an explicit, fully compressible method (M is the Mach number). In this mathematical framework, the total pressure is decomposed into the sum of a spatially constant (ambient) thermodynamic pressure P_0 and a perturbational pressure, $\pi(\vec{x})$ that drives the flow. Under suitable conditions, $\pi/P_0 = \mathcal{O}(M^2)$.

The set of conservation equations specialized to the low Mach number regime is a system of PDEs with advection, diffusion and reaction (ADR) processes that are constrained to evolve on the manifold of a spatially constant P_0:

$$\frac{\partial(\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u} + \tau) = -\nabla\pi + \rho \boldsymbol{F},$$
$$\frac{\partial(\rho Y_m)}{\partial t} + \nabla \cdot (\rho Y_m \boldsymbol{u} + \boldsymbol{\mathcal{F}}_m) = \rho\dot{\omega}_m,$$
$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho h \boldsymbol{u} + \boldsymbol{\mathcal{Q}}) = 0,$$

where ρ is the density, $\boldsymbol{u}$ is the velocity, h is the mass-weighted enthalpy, T is temperature and Y_m is the mass fraction of species m. $\dot{\omega}_m$ is the molar production rate for species m, the modeling of which will be described later in this section. τ is the stress tensor, $\boldsymbol{\mathcal{Q}}$ is the heat flux and $\boldsymbol{\mathcal{F}}_m$ are the species diffusion fluxes. These transport fluxes require the evaluation of transport coefficients (e.g., the viscosity μ, the conductivity λ and the diffusivity matrix D) which are computed using the library EGLIB, as will be described in more depth in the diffusion section. The momentum source, $\boldsymbol{F}$, is an external forcing term. For example, we have used $\boldsymbol{F}$ to implement a long-wavelength time-dependent force to establish and maintain quasi-stationary turbulence.

These evolution equations are supplemented by an equation of state for the thermodynamic pressure. For example, the ideal gas law,

$$P_0(\rho, Y_m, T) = \frac{\rho \mathcal{R} T}{W} = \rho \mathcal{R} T \sum_m \frac{Y_m}{W_m}$$

can be used, although *PeleLM* will soon support other more general expressions, such as Soave-Redlich-Kwong. In the above, W_m and W are the species m, and mean molecular weights, respectively. To close the system we also require a relationship between enthalpy, species and temperature. We adopt the definition used in the CHEMKIN standard,

$$h = \sum_m Y_m h_m(T)$$

where h_m is the species m enthalpy. Note that expressions for $h_m(T)$ see <section on thermo properties> incorporate the heat of formation for each species.

Neither species diffusion nor reactions redistribute the total mass, hence we have $\sum_m \boldsymbol{\mathcal{F}}_m = 0$ and $\sum_m \dot{\omega}_m = 0$. Thus, summing the species equations and using the definition $\sum_m Y_m = 1$ we obtain the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \boldsymbol{u} = 0$$

This, together with the conservation equations form a differential-algebraic equation (DAE) system that describes an evolution subject to a constraint. A standard approach to attacking such a system computationally is to differentiate the constraint until it can be recast as an initial value problem. Following this procedure, we set the thermodynamic pressure constant in the frame of the fluid,

$$\frac{DP_0}{Dt} = 0$$

and observe that if the initial conditions satisfy the constraint, an evolution satisfying the above will continue to satisfy the constraint over all time. Expanding this expression via the chain rule and continuity:

$$\nabla \cdot \boldsymbol{u} = \frac{1}{T}\frac{DT}{Dt} + W\sum_m \frac{1}{W_m}\frac{DY_m}{Dt} = S$$

The constraint here take the form of a condition on the divergence of the flow. Note that the actual expressions to use here will depend upon the chosen models for evaluating the transport fluxes.

2.2.1 Transport Fluxes

Expressions for the transport fluxes appearing above can be approximated in the Enskog-Chapman expansion as:

$$\boldsymbol{\mathcal{F}}_m = \rho Y_m \boldsymbol{V_m}$$

$$\tau_{i,j} = -\Big(\kappa - \frac{2}{3}\mu\Big)\delta_{i,j}\frac{\partial u_k}{\partial x_k} - \mu\Big(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\Big)$$

$$\boldsymbol{\mathcal{Q}} = \sum_m h_m \boldsymbol{\mathcal{F}}_m - \lambda' \nabla T - P_0 \sum_m \theta_m \boldsymbol{d_m}$$

where μ is the shear viscosity, κ is the bulk viscosity, and λ' is the partial thermal conductivity. In the *full matrix diffusion model*, the vector of m species diffusion velocities, $\boldsymbol{V_m}$, is given by:

$$\boldsymbol{V_m} = -\sum_j D_{m,j}\boldsymbol{d_j} - \theta_m \nabla ln(T)$$

where $D_{m,j}$ is the diffusion matrix, and $\boldsymbol{\theta}$ are thermal diffusion coefficients associated with the Soret (mass concentration flux due to an energy gradient) and Dufour (the energy flux due to a mass concentration gradient) effects. The m species transport driving force due to composition gradients, $\boldsymbol{d_m}$, is given by:

$$\boldsymbol{d_m} = \nabla X_m + (X_m - Y_m)\frac{\nabla P_0}{P_0}$$

Alternatively (as in the transport library, EGLIB) the thermal diffusion *ratios* $\boldsymbol{\chi}$ may be preferred and the diffusion velocities and energy flux recast as:

$$\boldsymbol{V_m} = -\sum_j D_{m,j}(\boldsymbol{d_j} + \chi_j \nabla ln(T))$$

$$\boldsymbol{\mathcal{Q}} = \sum_m h_m \boldsymbol{\mathcal{F}}_m - \lambda \nabla T + P_0 \sum_m \chi_m \boldsymbol{V_m}$$

where $D\boldsymbol{\chi} = \boldsymbol{\theta}$.

As can be seen, the expression for these fluxes relies upon several transport coefficients that need to be evaluated. However, in the present framework several effects are neglected, thus simplifying the fluxes evaluation.

2.2.2 The *PeleLM* Equation Set

The full diffusion model couples together the advance of all thermodynamics fields, including a dense matrix transport operator that is cumbersome to deal with computationally, while also being generally viewed as an overkill for most practical combustion applications – particularly those involving turbulent fluid dynamics. For *PeleLM*, we make the following simplifying assumptions:

1. The bulk viscosity, κ, is usually negligible, compared to the shear viscosity,
2. The low Mach limit implies that there are no spatial gradients in the thermodynamic pressure,
3. The *mixture-averaged* diffusion model is assumed,
4. Dufour and Soret effects are negligible

With these assumptions, the conservation equations take the following form:

$$\frac{\partial(\rho \boldsymbol{u})}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u} + \tau) = -\nabla \pi + \rho \boldsymbol{F},$$

$$\frac{\partial(\rho Y_m)}{\partial t} + \nabla \cdot (\rho Y_m \boldsymbol{u} + \boldsymbol{\mathcal{F}}_m) = \rho \dot{\omega}_m$$

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho h \boldsymbol{u} + \boldsymbol{\mathcal{Q}}) = 0,$$

with

$$\boldsymbol{\mathcal{F}}_m = \rho Y_m \boldsymbol{V_m} = -\rho \sum_k \widetilde{D_{m,k}} \nabla X_m$$

$$\tau_{i,j} = \frac{2}{3}\mu \delta_{i,j} \frac{\partial u_k}{\partial x_k} - \mu\Big(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\Big)$$

$$\boldsymbol{\mathcal{Q}} = \sum_m h_m \boldsymbol{\mathcal{F}}_m - \lambda \nabla T$$

where $\boldsymbol{d_m} = \nabla X_m$ and $\widetilde{D_{m,k}} = Y_m D_{m,k}$. Using these expressions, we can write an equation for T that is needed in order to evaluate the right-hand side of the divergence constraint:

$$\rho C_p \frac{DT}{Dt} = \nabla \cdot \lambda \nabla T + \sum_m \Big(h_m \nabla \cdot \boldsymbol{\mathcal{F}}_m - \nabla \cdot h_m \boldsymbol{\mathcal{F}}_m - h_m \rho \dot{\omega}_m \Big)$$

where $C_p = \partial h / \partial T$ is the specific heat of the mixture at constant pressure. For an ideal gas, the constraint then becomes:

$$\nabla \cdot \boldsymbol{u} =$$

$$\frac{1}{\rho C_p T}\Big[\nabla \cdot \lambda \nabla T + \sum_m \Big(h_m \nabla \cdot \boldsymbol{\mathcal{F}}_m - \nabla \cdot h_m \boldsymbol{\mathcal{F}}_m \Big)\Big]$$

$$-\frac{W}{\rho} \sum_m \frac{1}{W_m} \nabla \cdot \boldsymbol{\mathcal{F}}_m + \sum_m \Big(\frac{W}{W_m} - \frac{h_m(T)}{c_p T} \Big) \dot{\omega}_m$$

The mixture-averaged transport coefficients discussed above (μ, λ and $D_{m,mix}$) can be evaluated from transport properties of the pure species. We follow the treatment used in the EGLib library, based on the theory/approximations developed by Ern and Givangigli (however, *PeleLM* uses a recoded version of these routines that are thread safe and vectorize well on suitable processors).

The following choices are currently implemented in *PeleLM*

- The viscosity, μ, is estimated based on one step of the conjugate gradient method, using temperature dependent ratios of collisions integrals (EGZE3).
- The conductivity, λ, is based on an empirical mixture formula (EGZL1):

$$\lambda = \mathcal{A}_{0.25}$$

with

$$\mathcal{A}_\alpha = \Big(\sum_m X_m (\lambda_m)^\alpha \Big)^{1/\alpha}$$

- The flux diffusion matrix is approximated using the diagonal of the flux diffusion vector $\rho\widetilde{\Upsilon}$, where:

$$\rho\widetilde{\Upsilon}_m = \rho \frac{W_m}{W} D_{m,mix}, \quad \text{where} \quad D_{m,mix} = \frac{1 - Y_m}{\sum_{j \neq m} X_j / \mathcal{D}_{m,j}}$$

and the $\mathcal{D}_{m,j}$ are the binary diffusion coefficients of the pair (m,j). This leads to a mixture-averaged approximation that is similar to that of Hirschfelder-Curtiss (EGZVR1):

$$\rho Y_m \boldsymbol{V_m} = -\rho D_{m,mix} \frac{W_m}{W} \nabla X_m$$

Note that with these definitions, there is no guarantee that $\sum \boldsymbol{\mathcal{F}}_m = 0$, as required for mass conservation. An arbitrary *correction flux,* consistent with the mixture-averaged diffusion approximation, is added in *PeleLM* to enforce conservation.

The pure species and mixture transport properties are evaluated with (thread-safe, vectorized) EGLib functions, which require as input polynomial fits of the logarithm of each quantity versus the logarithm of the temperature.

$$ln(q_m) = \sum_{n=1}^{4} a_{q,m,n} ln(T)^{(n-1)}$$

q_m represents η_m, λ_m or $D_{m,j}$. These fits are generated as part of a preprocessing step managed by the tool *FUEGO* based on the formula (and input data) discussed above. The role of *FUEGO* is to preprocess the model parameters for transport as well as chemical kinetics and thermodynamics.

2.2.3 Chemical kinetics and the reaction source term

Chemistry in combustion systems involves the N_s species interacting through a set of M_r elementary reaction steps, expressed as

$$\sum_{m=1}^{N_s} \nu'_{m,j} [X_m] \rightleftharpoons \sum_{m=1}^{N_s} \nu''_{m,j} [X_m], \quad for \quad j \in [1, M_r]$$

where $[X_m]$ is the molar concentration of species m, and $\nu'_{m,j}$, $\nu''_{m,j}$ are the stoichiometric coefficients on the reactant and product sides of reaction j, associated with m. For such a system, the rate of reaction j (R_j) can be expressed in terms of the the forward ($k_{f,j}$) and backward ($k_{r,j}$) rate coefficients,

$$R_j = k_{f,j} \prod_{m=1}^{N_s} [X_m]^{\nu'_{m,j}} - k_{r,j} \prod_{m=1}^{N_s} [X_m]^{\nu''_{m,j}}$$

The net molar production rate, :math:‘ dot{omega}_m‘ of species m is obtained by collating the rate of creation and destruction over reactions:

$$\dot{\omega}_m = \sum_{j=1}^{M_r} \nu_{m,j} R_j$$

where $\nu_{m,j} = \nu''_{m,j} - \nu'_{m,j}$. Expressions for the reaction rates coefficients $k_{(f,r),j}$ depend on the type of reaction considered. We use the CHEMKIN modified Arrhenius reaction format:

$$k_f = AT^{\beta} exp\left(\frac{-E_a}{RT}\right)$$

where A is the pre-exponential (frequency) factor, β is the temperature exponent and E_a is the activation energy. The CHEMKIN format additionally allows for a number of specializations of this format to represent pressure dependencies and third-body enhancements – see the CHEMKIN Manual or Cantera website for additional information.

Most fundamental Arrhenius reactions are bidirectional, and typically only the forward rates are specified. In this case, the balance of forward and reverse rates are dictacted by equilibrium thermodynamics, via the equilibrium constant, $K_{c,j}$. In a low Mach system, $K_{c,j}$ is a function only of temperature and the thermodynamic properties of the reactants and products of reaction j,

$$k_{r,j} = \frac{k_{f,j}}{K_{c,j}(T)} \quad \text{where} \quad K_{c,j} = K_{p,j}\left(\frac{P_0}{RT}\right)^{\sum_{k=1}^{N_s}\nu_{k,j}}$$

$$\text{and} \quad K_{p,j} = \exp\left(\frac{\Delta {S_j}^0}{R} - \frac{\Delta {H_j}^0}{RT}\right)$$

ΔH_j and ΔS_j are the change in enthalpy and entropy of the reaction j, and P_0 is the ambient thermodynamic pressure.

Species production rates are evaluated via functions that are generated as part of a preprocessing step managed by the tool *FUEGO*.

2.2.4 Thermodynamic properties

Currently, expressions for the thermodynamic properties in *PeleLM* follow those of CHEMKIN, which assume a mixture of ideal gases. Species enthalpies and entropies are thus functions of only temperature (for perfect gases, they are independent of pressure) and are given in terms of polynomial fits to the species molar heat capacities ($C_{p,\cdot}$),

$$\frac{C_{p,m}(T)}{\mathcal{R}} = \sum_{k=1}^{N_s} a_{k,m} T^{k-1}$$

where, in the standard CHEMKIN framework (the 7-coefficients NASA format), $N = 5$ and

$$\frac{C_{p,m}(T)}{\mathcal{R}} = a_{1,m} + a_{2,m}T + a_{3,m}T^2 + a_{4,m}T^3 + a_{5,m}T^4$$

Accordingly, the standard-state molar enthalpy of species m is given by:

$$\frac{H_m(T)}{\mathcal{R}T} = a_{1,m} + \frac{a_{2,m}}{2}T + \frac{a_{3,m}}{3}T^2 + \frac{a_{4,m}}{4}T^3 + \frac{a_{5,m}}{5}T^4 + a_{6,m}/T$$

Note that the standard specifies that the heat of formation for the molecule is included in this expression. Similarly, the standard-state molar entropy is written as:

$$\frac{S_m(T)}{\mathcal{R}} = a_{1,m}ln(T) + a_{2,m}T + \frac{a_{3,m}}{2}T^2 + \frac{a_{4,m}}{3}T^3 + \frac{a_{5,m}}{4}T^4 + a_{7,m}$$

For each species m, in the model the user must specify the 7 k coefficients $a_{k,m}$. All other required thermodynamic properties are then determined (see, e.g., the CHEMKIN manual for additional details). Thermodynamic properties of the species, and those of the mixture, are evaluated via functions that are generated as part of a preprocessing step managed by the tool *FUEGO*.

2.2.5 *FUEGO* chemistry preprocessing

A typical model for *PeleLM* contains all the information associated with the CHEMKIN parameterization of the Arrhenius reaction set, as well as fitting coefficients for the thermodynamic relationships, and the specification of the species including data required to compute pure-species transport properties. In the combustion community, this information is communicated for each complete model –or *mechanism*, through multiple text files that conform to the CHEMKIN standards. The CHEMKIN driver code (or equivalent) can then be used to ingest the large number of parameters contained in these files and provide a set of functions for evaluating all the properties and rates required. Earlier versions of *PeleLM* linked to the CHEMKIN codes directly (and thereby assumed that all problems consisted of a mixture of ideal gases). However, evaluations were not very efficient because the functions stepped through generic expressions that included a large number of conditional statements and unused generality. Direct evaluation of these complex expressions allows for a much more efficient code that optimizes well with modern compilers. This is important because an appreciable fraction of *PeleLM* runtime is spent in these functions. Performance issues notwithstanding, customized evaluators will be necessary to extend *PeleLM* to a larger class of (*real*) gas models outside the CHEMKIN standard, such as SRK, that are already part of the *PeleC* code capabilities (*PeleC* shares use of *PelePhysics* for combustion model specification).

For these reasons, *PeleLM* no longer uses CHEMKIN functions directly, but instead relies on a preprocessing tool, *FUEGO*, to generate highly efficient C code implementations of the necessary thermodynamic, transport and kinetics evaluations. The source code generated from *FUEGO* is linked into the *PeleLM* executable, customizing each executable for a specific model at compile time. The implementation source code files can also be linked conveniently to post-processing analysis tools. The *FUEGO* processing tool, and the functions necessary to interface the generated functions to *PeleLM* are distributed in the auxiliary code package, *PelePhysics*. Included in the *PelePhysics* distribution is a broad set of models for the combustion of hydrogen, carbon-monoxide, methane, heptane, n-dodecane, dimethyl ether, and others, as well as instructions for users to extend this set using *FUEGO*, based on their own CHEMKIN-compliant inputs. *PelePhysics* also provides support for simpler *gama-law* equations-of-state, and simple/constant transport properties.

2.3 The *PeleLM* temporal integration

The temporal discretization in *PeleLM* combines a modified spectral deferred correction (SDC) coupling of chemistry and transport with a density-weighted approximate projection method for low Mach number flow. The projection method enforces a constrained evolution of the velocity field, and is implemented iteratively in such a way as to ensure 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 deferred corrections strategy. The integration algorithm, discussed in the following sections, is second-order accurate in space and time, and is implemented in the context of a subcycled approach for a nested hierarchy of mesh levels, where each level consists of logically rectangular patches of rectangular cells. All cells at a level have the same size in all coordinates.

Due to the complexity of the *PeleLM* algorithm, it is best presented in a number of passes. Focusing first on the single-level advance, we begin with a general discussion of the SDC-based time step iteration, which is designed to couple together the various physics processes. We then describe the projection steps used to enforce the constraint in the context of this iterative update. Next, we dive a little deeper into precisely how the advance of the thermodynamic components of the state is sequenced. There are a few crucial nuances to the formulation/sequencing of the energy advection, energy diffusion, conservative corrections to the species diffusion fluxes, and of the projection that can then be discussed in the context of overall single-level time step. Finally, with all these aspects defined, we give an overview of the modifications necessary to support the AMR subcycling strategy.

2.3.1 SDC preliminaries

The basic idea of SDC is to write the solution of an ODE

$$\phi_t = F(t, \phi(t)), \qquad t \in [t^n, t^{n+1}];$$

$$\phi(t^n) = \phi^n,$$

as an integral,

$$\phi(t) = \phi^n + \int_{t^n}^{t} F(\phi)\, d\tau,$$

where we suppress explicit dependence of F and ϕ on t for notational simplicity. Given an approximation $\phi^{(k)}(t)$ to $\phi(t)$, one can then define a residual,

$$E(t, \phi^{(k)}) = \phi^n + \int_{t^n}^{t} F(\phi^{(k)})\, d\tau - \phi^{(k)}(t).$$

Defining the error as $\delta^{(k)}(t) = \phi(t) - \phi^{(k)}(t)$, one can then show that

$$\delta^{(k)}(t) = \int_{t^n}^{t} \left[F(\phi^{(k)} + \delta^{(k)}) - F(\phi^{(k)}) \right] d\tau + E(t, \phi^{(k)}).$$

In SDC algorithms, the integral in the above equation is evaluated with a higher-order quadrature rule. By using a low-order discretization of the integral one can construct an iterative scheme that improves the overall order of accuracy of the approximation by one per iteration, up to the order of accuracy of the underlying quadrature rule used to evaluate the integral. Specifically, if we let $\phi^{(k)}$ represent the current approximation and define $\phi^{(k+1)} = \phi^{(k)} + \delta^{(k)}$ to be the iterative update, then arrive at the update equation,

$$\phi^{(k+1)}(t) = \phi^n + \int_{t^n}^{t} \left[F(\phi^{(k+1)}) - F(\phi^{(k)}) \right] d\tau + \int_{t^n}^{t} F(\phi^{(k)})\, d\tau,$$

where a low-order discretization (e.g., forward or backward Euler) is used for the first integral and a higher-order quadrature is used to evaluate the second integral. For our reacting flow model, the underlying projection methodology for the time-advancement of velocity is second-order, so we require the use of second-order (or higher) numerical quadrature for the second integral.

2.3.2 MISDC Correction Equations

We based the time advance here on a variant of SDC, referred to as MISDC, in which F is decomposed into distinct processes, each treated separately with methods appropriate to its own time scale. Here, we write

$$\phi_t = F \equiv A(\phi) + D(\phi) + R(\phi),$$

to refer to advection, diffusion, and reaction processes. For this construction we assume that we are given an approximate solution $\phi^{(k)}$ that we want to improve. A series of correction equations is developed to update $\phi^{(k)}$ that uses relatively simple second-order discretizations of $A(\phi)$ and $D(\phi)$ but a high-accuracy treatment of $R(\phi)$. In our approach, $A(\phi^{(k)})$ is piecewise-constant over each time step, and is evaluated using a second-order Godunov procedure. The Godunov procedure computes a time-centered advection term at $t^{n+1/2}$, and incorporates an explicit diffusion source term and an iteratively lagged reaction source term, i.e.,

$$A(\phi^{(k)}) \equiv A^{n+1/2,(k)} = A\left(\phi^n, D(\phi^n), I_R^{(k-1)}\right),$$

where $I_R^{(k-1)}$ is the effective contribution due to reactions from the previous iteration, i.e.,

$$I_R^{(k-1)} = \frac{1}{\Delta t^n} \int_{t^n}^{t^{n+1}} R(\phi^{(k-1)})\, d\tau.$$

where $\Delta t^n = t^{n+1} - t^n$. Here $I_R^{(k-1)}$ is computed from a high-accuracy integration of the reaction kinetics equations, augmented with piecewise constant-in-time representation of advection and diffusion. Details of this procedure are given below.

We also represent $D(\phi^{(k)})$ as piecewise constant over the time step, using a mid-point rule:

$$D(\phi^k) = \frac{1}{2}(D(\phi^n) + D(\phi^{(n+1,k)}))$$

In the spirit of MISDC, we solve correction equations for the individual processes sequentially. We begin by discretizing the update equation, but only including the advection and diffusion terms in the correction integral,

$$\phi_{\rm AD}^{(k+1)}(t) = \phi^n + \int_{t^n}^{t} \left[A^{(k+1)} - A^{(k)} + D^{(k+1)} - D^{(k)}\right] d\tau + \int_{t^n}^{t} F^{(k)}\, d\tau.$$

Thus, $\phi_{\rm AD}^{(k+1)}(t)$ represents an updated approximation of the solution after correcting the advection and diffusion terms only. For the first integral, we use an explicit update for the advection term and a backward Euler discretization for the diffusion term. For the second integral, we represent F in terms of A, D, and R and use the definition of $A^{(k)}$, $D^{(k)}$, and $I_R^{(k-1)}$ to obtain a discrete update for $\phi_{\rm AD}^{n+1,(k+1)}$:

$$\phi_{\rm AD}^{n+1,(k+1)} = \phi^n + \Delta t \left[A^{n+1/2,(k+1)} - A^{n+1/2,(k)} + D_{\rm AD}^{n+1,(k+1)} - D^{n+1,(k)}\right] + \Delta t \left[A^{n+1/2,(k)} + \frac{1}{2}\left(D^n + D^{n+1,(k)}\right) + I_R^{(k)}\right],$$

This equation simplifies to the following backward Euler type linear system, with the right-hand-side consisting of known quantities:

$$\phi_{\rm AD}^{n+1,(k+1)} - \Delta t D_{\rm AD}^{n+1,(k+1)} = \phi^n + \Delta t \left[A^{n+1/2,(k+1)} + \frac{1}{2}\left(D^n - D^{n+1,(k)}\right) + I_R^{(k)}\right], \tag{2.1}$$

After computing $\phi_{\rm AD}^{n+1,(k+1)}$, we complete the update by solving a correction equation for the reaction term. Standard MISDC approaches would formulate the reaction correction equation as

$$\phi^{(k+1)}(t) = \phi^n + \int_{t^n}^{t} \left[A^{(k+1)} - A^{(k)} + D_{\rm AD}^{(k+1)} - D^{(k)}\right] d\tau + \int_{t^n}^{t} \left[R^{(k+1)} - R^{(k)}\right] d\tau + \int_{t^n}^{t} F^{(k)}\, d\tau,$$

and use a backward Euler type discretization for the integral of the reaction terms. Here, to address stiffness issues with detailed chemical kinetics, we will instead formulate the correction equation for the reaction as an ODE, which

is treated separately with an ODE integrator package. In particular, by differentiating the SDC update we obtain

$$\begin{aligned}
\phi_t^{(k+1)} = \\
\left[A^{n+1/2,(k+1)} - A^{n+1/2,(k)} + D_{\mathrm{AD}}^{n+1,(k+1)} - D^{n+1,(k)}\right] \\
+ \left[R^{(k+1)} - R^{(k)}\right] + \left[A^{n+1/2,(k)} + \frac{1}{2}\left(D^n + D^{n+1,(k)}\right) + R^{(k)}\right] \\
= \\
R^{(k+1)} + \underbrace{A^{n+1/2,(k+1)} + D_{\mathrm{AD}}^{n+1,(k+1)} + \frac{1}{2}\left[D^n - D^{n+1,(k)}\right]}_{F_{\mathrm{AD}}^{n+1,(k+1)}},
\end{aligned}$$

which we then advance with the ODE integrator over Δt to obtain $\phi^{n+1,(k+1)}$. After the integration, we can evaluate $I_R^{(k+1)}$, which is required for the next iteration

$$I_R^{(k+1)} = \frac{\phi^{n+1,(k+1)} - \phi^n}{\Delta t} - F_{\mathrm{AD}}^{n+1,(k+1)}.$$

Summarizing, the variant of SDC used in the single-level time-step of *PeleLM* integrates the A, D and R components of the discretization scheme in an iterative fashion, and each process incorporates a source term that is constructed using a lagged approximation of the other processes. In the case of the implicit diffusion, an additional source term arises from the SDC formulation. If the SDC iterations were allowed to fully converge, all the process advanced implicitly would be implicitly coupled to all others. Moreover, each process is discretized using methods that are tailored specifically to the needs of that operator. In the next section, we give more details for each of the components, including how and where the *velocity projections* play a role.

2.3.3 Data centering, A-D-R, and the projections

PeleLM implements a finite-volume, Cartesian grid discretization approach with constant grid spacing, where U, ρ, ρY_m, ρh, and T represent cell averages, and the pressure field, π, is defined on the nodes of the grid, and is temporally constant on the intervals over the time step. There are three major steps in the algorithm:

Step 1: (*Compute advection velocities*) Use a second-order Godunov procedure to predict a time-centered velocity, $U^{\mathrm{ADV},*}$, on cell faces using the cell-centered data (plus sources due to any auxiliary forcing) at t^n, and the lagged pressure gradient from the previous time interval, which we denote as $\nabla\pi^{n-1/2}$. The provisional field, $U^{\mathrm{ADV},*}$, fails to satisfy the divergence constraint. We apply a discrete projection by solving the elliptic equation with a time-centered source term:

$$D^{\mathrm{FC}\to\mathrm{CC}}\frac{1}{\rho^n}G^{\mathrm{CC}\to\mathrm{FC}}\phi = D^{\mathrm{FC}\to\mathrm{CC}}U^{\mathrm{ADV},*} - \left(\widehat{S}^n + \frac{\Delta t^n}{2}\frac{\widehat{S}^n - \widehat{S}^{n-1}}{\Delta t^{n-1}}\right),$$

for ϕ at cell-centers, where $D^{\mathrm{FC}\to\mathrm{CC}}$ represents a cell-centered divergence of face-centered data, and $G^{\mathrm{CC}\to\mathrm{FC}}$ represents a face-centered gradient of cell-centered data, and ρ^n is computed on cell faces using arithmetic averaging from neighboring cell centers. Also, $\widehat{S}$ refers to the RHS of the constraint equation, with adjustments to be discussed in the next section – these adjustments are computed to ensure that the final update satisfied the equation of state. The solution, ϕ, is then used to define

$$U^{\mathrm{ADV}} = U^{\mathrm{ADV},*} - \frac{1}{\rho^n}G^{\mathrm{CC}\to\mathrm{FC}}\phi,$$

After the *MAC*-projection, U^{ADV} is a second-order accurate, staggered grid vector field at $t^{n+1/2}$ that discretely satisfies the constraint. This field is the advection velocity used for computing the time-explicit advective fluxes for U, ρh, and ρY_m.

Step 2: (*Advance thermodynamic variables*) Integrate $(\rho Y_m, \rho h)$ over the full time step. The details of this are presented in the next subsection.

Step 3: (*Advance the velocity*) Compute an intermediate cell-centered velocity field, $U^{n+1,*}$ using the lagged pressure gradient, by solving

$$\rho^{n+1/2}\frac{U^{n+1,*}-U^n}{\Delta t}+\rho^{n+1/2}\left(U^{\mathrm{ADV}}\cdot\nabla U\right)^{n+1/2}=$$
$$\frac{1}{2}\left(\nabla\cdot\tau^n+\nabla\cdot\tau^{n+1,*}\right)-\nabla\pi^{n-1/2}+\frac{1}{2}(F^n+F^{n+1}),$$

where $\tau^{n+1,*}=\mu^{n+1}[\nabla U^{n+1,*}+(\nabla U^{n+1,*})^T-2\mathcal{I}\widehat{S}^{n+1}/3]$ and $\rho^{n+1/2}=(\rho^n+\rho^{n+1})/2$, and F is the velocity forcing. This is a semi-implicit discretization for U, requiring a linear solve that couples together all velocity components. The time-centered velocity in the advective derivative, $U^{n+1/2}$, is computed in the same way as $U^{\mathrm{ADV},*}$, but also includes the viscous stress tensor evaluated at t^n as a source term in the Godunov integrator. At this point, the intermediate velocity field $U^{n+1,*}$ does not satisfy the constraint. Hence, we apply an approximate projection to update the pressure and to project $U^{n+1,*}$ onto the constraint surface. In particular, we compute $\widehat{S}^{n+1}$ from the new-time thermodynamic variables and an estimate of $\dot{\omega}_m^{n+1}$, which is evaluated directly from the new-time thermodynamic variables. We project the new-time velocity by solving the elliptic equation,

$$L^{\mathrm{N}\to\mathrm{N}}\phi=D^{\mathrm{CC}\to\mathrm{N}}\left(U^{n+1,*}+\frac{\Delta t}{\rho^{n+1/2}}G^{\mathrm{N}\to\mathrm{CC}}\pi^{n-1/2}\right)-\widehat{S}^{n+1}$$

for nodal values of ϕ. Here, $L^{\mathrm{N}\to\mathrm{N}}$ represents a nodal Laplacian of nodal data, computed using the standard bilinear finite-element approximation to $\nabla\cdot(1/\rho^{n+1/2})\nabla$. Also, $D^{\mathrm{CC}\to\mathrm{N}}$ is a discrete second-order operator that approximates the divergence at nodes from cell-centered data and $G^{\mathrm{N}\to\mathrm{CC}}$ approximates a cell-centered gradient from nodal data. Nodal values for $\widehat{S}^{n+1}$ required for this equation are obtained by interpolating the cell-centered values. Finally, we determine the new-time cell-centered velocity field using

$$U^{n+1}=U^{n+1,*}-\frac{\Delta t}{\rho^{n+1/2}}G^{\mathrm{N}\to\mathrm{CC}}(\phi-\pi^{n-1/2}),$$

and the new time-centered pressure using $\pi^{n+1/2}=\phi$.

Thus, there are three different types of linear solves required to advance the velocity field. The first is the *MAC* solve in order to obtain *face-centered* velocities used to compute advective fluxes. The second is the multi-component *cell-centered* solver is used to obtain the provisional new-time velocities. Finally, a *nodal* solver is used to project the provisional new-time velocities so that they satisfy the constraint.

2.3.4 Thermodynamic Advance

Here we describe the details of **Step 2** above, in which we iteratively advance $(\rho Y_m, \rho h)$ over the full time step. We begin by computing the diffusion operators at t^n that will be needed throughout the iteration. Specifically, we evaluate the transport coefficients $(\lambda, C_p, \mathcal{D}_m, h_m)^n$ from $(Y_m, T)^n$, and the provisional diffusion fluxes, $\widetilde{\boldsymbol{\mathcal{F}}}_m^n$. These fluxes are conservatively corrected (i.e., adjusted to sum to zero by adding a mass-weighted "correction velocity") to obtain $\boldsymbol{\mathcal{F}}_m^n$ such that $\sum\boldsymbol{\mathcal{F}}_m^n=0$. Finally, we copy the transport coefficients, diffusion fluxes and the thermodynamic state from t^n as starting values for t^{n+1}, and initialize the reaction terms, I_R from the values used in the previous step. The following sequence is then repeated for each iteration, $k<k_{max}$

Step 2-I: Use a second-order Godunov integrator to predict species time-centered edge states, $(\rho Y_m)^{n+1/2,(k+1)}$ and their advection terms at $t^{n+1/2}$, $(A_m^{n+1/2,(k+1)})$. Source terms for this prediction include explicit diffusion forcing, D^n, and an iteration-lagged reaction term, $I_R^{(k)}$. Since the remaining steps of the algorithm for this iteration (including diffusion and chemistry advances) will not affect the new-time density for this iteration, we can already compute $\rho^{n+1,(k+1)}$. This will be needed in the trapezoidal-in-time diffusion solves.

$$\frac{\rho^{n+1,(k+1)}-\rho^n}{\Delta t}=A_\rho^{n+1/2,(k+1)}=\sum A_m^{n+1/2,(k+1)}=-\sum_m\nabla\cdot\left(U^{\mathrm{ADV}}\rho Y_m\right)^{n+1/2,(k+1)}.$$

In addition to predicting ρ and ρY_m to the faces to compute advective fluxes, we also need ρh there. We could use a Godunov scheme as well, however, because h contains the heat of formation scaled to an arbitrary reference state, it is not generally monotonic through flames. Also, because the equation of state is generally nonlinear, this will often lead to numerically-generated n on-mononoticity in the temperature field. An analytically equivalent approach, based on the fact that temperature should be smoother and monotonic through the flame, is to instead predict temperature with the Godunov scheme to the cell faces directly. Then, using T, $\rho = \sum(\rho Y_m)$ and $Y_m = (\rho Y_m)/\rho$ on the cell faces directly, we can evaluate h instead of extrapolating. We can then evaluate the enthalpy advective flux divergence, $A_h^{n+1/2,(k+1)}$, for ρh.

Step 2-II: Update the transport coefficients (if necessary) with the most current cell-centered thermodynamic state, then interpolate those values to the cell faces. We now compute provisional, time-advanced species mass fractions, $\widetilde{Y}_{m,\mathrm{AD}}^{n+1,(k+1)}$, by solving a backward Euler type correction equation for the Crank-Nicolson update, using Eq. (2.1). Note that the provisional species diffusion fluxes reads $\widetilde{\boldsymbol{\mathcal{F}}}_{m,\mathrm{AD}}^{(0)} = -\rho^n D_{m,mix}^n \nabla \widetilde{X}_{m,\mathrm{AD}}^{(0)}$. This expression couples together all of the species mass fractions (Y_m) in the update of each, even for the mixture-averaged model. Computationally, it is much more tractable to write this as a diagonal matrix update with a lagged correction by noting that $X_m = (W/W_m)Y_m$. Using the chain rule, $\widetilde{\boldsymbol{\mathcal{F}}}_{m,\mathrm{AD}}^{(0)}$ then has components proportional to ∇Y_m and ∇W. The latter is lagged in the iterations, and is typically very small. In the limit of sufficient iterations, diffusion is driven by the true form of the the driving force, d_m, but in this form, each iteration involves decoupled diagonal solves (following the SDC formalism used above):

$$\frac{\rho^{n+1,(k+1)}\widetilde{Y}_{m,\mathrm{AD}}^{n+1,(k+1)} - (\rho Y_m)^n}{\Delta t} = A_m^{n+1/2,(k+1)} + \widetilde{D}_{m,AD}^{n+1,(k+1)} + \frac{1}{2}(D_m^n - D_m^{n+1,(k)}) + I_{R,m}^{(k)}$$

where

$$D_m^n = -\nabla \cdot \boldsymbol{\mathcal{F}}_m^n$$
$$D_m^{n+1,(k)} = -\nabla \cdot \boldsymbol{\mathcal{F}}_m^{n+1,(k)}$$
$$\widetilde{D}_{m,AD}^{n+1,(k+1)} = -\nabla \cdot \widetilde{\boldsymbol{\mathcal{F}}}_{m,AD}^{n+1,(k+1)}$$
$$\widetilde{D}_{m,AD}^{n+1,(k+1)} = \nabla \cdot \left[\rho^{n+1,(k+1)} D_{m,mix}^{n+1,(k)} \frac{W^{n+1,(k)}}{W_m} \nabla \widetilde{Y}_{m,\mathrm{AD}}^{n+1,(k+1)} \; + \; \rho^{n+1,(k+1)} D_{m,mix}^{n+1,(k)} \frac{Y_m^{n+1,(k)}}{W_m} \nabla W^{n+1,(k)}\right]$$

By iteratively lagging the ∇W term (and $D_{m,mix}$), this equation is a scalar, time-implicit, parabolic and linear for the updated $\widetilde{Y}_{m,\mathrm{AD}}^{n+1,(k+1)}$ (and requires a linear solve). The form of this solve, from a software perspective, is identical to that of the *MAC* projection discussed above.

Once all the species equations are updated, we compute $\boldsymbol{\mathcal{F}}_{m,\mathrm{AD}}^{n+1,(k+1)}$, which are conservatively corrected versions of $\widetilde{\boldsymbol{\mathcal{F}}}_{m,\mathrm{AD}}^{n+1,(k+1)}$, and then the species mass fractions are updated too, using

$$\frac{\rho^{n+1,(k+1)} Y_{m,\mathrm{AD}}^{n+1,(k+1)} - (\rho Y_m)^n}{\Delta t} = A_m^{n+1/2,(k+1)} + D_{m,AD}^{n+1,(k+1)} + \frac{1}{2}(D_m^n - D_m^{n+1,(k)}) + I_{R,m}^{n+1,(k)} \tag{2.2}$$

where

$$D_{m,AD}^{n+1,(k+1)} = -\nabla \cdot \boldsymbol{\mathcal{F}}_{m,\mathrm{AD}}^{n+1,(k+1)}$$

Next, we compute the time-advanced enthalpy, $h_{\mathrm{AD}}^{n+1,(k+1)}$. Much like for the diffusion of the species densities, ρY_m, where a ∇X_m driving force leads to a nonlinear, coupled Crank-Nicolson update; the enthalpy diffuses with a ∇T driving force. We define an alternative linearized strategy. We begin by following the same SDC-correction formalism used for the species, and write the nonlinear update for ρh (noting that there is no reaction source term here):

$$\begin{aligned}\frac{\rho^{n+1,(k+1)} h_{\mathrm{AD}}^{n+1,(k+1)} - (\rho h)^n}{\Delta t} &= A_h^{n+1/2,(k+1)} + D_{T,AD}^{n+1,(k+1)} + H_{AD}^{n+1,(k+1)} \\ &\quad + \frac{1}{2}\Big(D_T^n - D_T^{n+1,(k)} + H^n - H^{n+1,(k)}\Big)\end{aligned} \tag{2.3}$$

where

$$D_T^n = \nabla \cdot \lambda^n \nabla T^n,$$
$$H^n = -\nabla \cdot \sum h_m(T^n)\, \boldsymbol{\mathcal{F}}_m^n$$
$$D_T^{n+1,(k)} = \nabla \cdot \lambda^{n+1,(k)} \nabla T^k,$$
$$H^{n+1,(k)} = -\nabla \cdot \sum h_m(T^{n+1,(k)})\, \boldsymbol{\mathcal{F}}_m^{n+1,(k)}$$
$$D_{T,AD}^{n+1,(k+1)} = \nabla \cdot \lambda_{AD}^{n+1,(k+1)} \nabla T_{AD}^{n+1,(k+1)},$$
$$H_{AD}^{n+1,(k+1)} = -\nabla \cdot \sum h_m(T_{AD}^{n+1,(k+1)})\, \boldsymbol{\mathcal{F}}_{m,AD}^{n+1,(k+1)}$$

However, since we cannot compute $h_{\rm AD}^{n+1,(k+1)}$ directly, we solve this iteratively based on the approximation $h_{\rm AD}^{(k+1),\ell+1} \approx h_{\rm AD}^{(k+1),\ell} + C_p^{(k+1),\ell}\delta T^{(k+1),\ell+1}$, with $\delta T^{(k+1),\ell+1} = T_{\rm AD}^{(k+1),\ell+1} - T_{\rm AD}^{(k+1),\ell}$, and iteration index, ℓ = 1: ℓ_{MAX}. The enthalpy update equation is thus recast into a linear equation for $\delta T^{(k+1);\ell+1}$

$$\rho^{n+1,(k+1)} C_p^{(k+1),\ell}\delta T^{(k+1),\ell+1} - \Delta t\, \nabla \cdot \lambda^{(k)} \nabla(\delta T^{(k+1),\ell+1}) =$$
$$\rho^n h^n - \rho^{n+1,(k+1)} h_{AD}^{(k+1),\ell} + \Delta t\Big(A_h^{n+1/2,(k+1)} + D_{T,AD}^{(k+1),\ell} + H_{AD}^{(k+1),\ell}\Big)$$
$$+ \frac{\Delta t}{2}\Big(D_T^n - D_T^{n+1,(k)} + H^n - H^{n+1,(k)}\Big)$$

where $H_{AD}^{(k+1),\ell} = -\nabla\cdot\sum h_m(T_{AD}^{(k+1),\ell})\, \boldsymbol{\mathcal{F}}_{m,AD}^{n+1,(k+1)}$ and $D_{T,AD}^{(k+1),\ell} = \nabla\cdot\lambda^{(k)}\, \nabla T_{AD}^{(k+1),\ell}$. Note that again the solve for this Crank-Nicolson update has a form that is identical to that of the *MAC* projection discussed above. After each iteration, update $T_{\rm AD}^{(k+1),\ell+1} = T_{\rm AD}^{(k+1),\ell} + \delta T^{(k+1),\ell+1}$ and re-evaluate $(C_p, h_m)^{(k+1),\ell+1}$ using $(T_{\rm AD}^{(k+1),\ell+1}, Y_{m,{\rm AD}}^{n+1,(k+1)})$.

Step 2-III: Based on the updates above, we define an effective contribution of advection and diffusion to the update of ρY_m and ρh:

$$Q_m^{n+1,(k+1)} = A_m^{n+1/2,(k+1)} + D_{m,AD}^{(n+1,k+1)} + \frac{1}{2}(D_m^n - D_m^{n+1,(k)})$$

$$Q_h^{n+1,(k+1)} = A_h^{n+1/2,(k+1)} + D_{T,AD}^{n+1,(k+1)} + H_{AD}^{n+1,(k+1)} + \frac{1}{2}(D_T^n - D_T^{n+1,(k)} + H^n - H^{n+1,(k)})$$

that we treat as piecewise-constant source terms to advance $(\rho Y_m, \rho h)^n$ to $(\rho Y_m, \rho h)^{n+1,(k+1)}$. The ODE system for the reaction part over Δt^n then takes the following form:

$$\frac{\partial(\rho Y_m)}{\partial t} = Q_m^{n+1,(k+1)} + \rho\dot{\omega}_m(Y_m, T),$$
$$\frac{\partial(\rho h)}{\partial t} = Q_h^{n+1,(k+1)}.$$

After the integration is complete, we make one final call to the equation of state to compute $T^{n+1,(k+1)}$ from $(Y_m, h)^{n+1,(k+1)}$. We also can compute the effect of reactions in the evolution of ρY_m using,

$$I_{R,m}^{(k+1)} = \frac{(\rho Y_m)^{n+1,(k+1)} - (\rho Y_m)^n}{\Delta t} - Q_m^{n+1,(k+1)}.$$

If $k < k_{\max} - 1$, set $k = k + 1$ and return to **Step 2-I**. Otherwise, the time-advancement of the thermodynamic variables is complete, and set $(\rho Y_m, \rho h)^{n+1} = (\rho Y_m, \rho h)^{n+1,(k+1)}$. If $k + 1 = k_{max}$, **Step 2** of our algorithm is complete.

2.3.5 Modifications for AMR

The framework to manage adaptive mesh refinement (AMR) used in *PeleLM* borrows heavily from the *AMReX* library, and the *IAMR* code; the reader is referred to documentation of both of these components in order to understand the distributed, logically rectangular data structures used, and the recursive time-stepping strategy for advancing a hierarchy of nested grid levels.

Summarizing, there is a bulk-synchronous advance of each level over its respective time step, dt, followed recursively by a number of (sub-)steps of the next-finer AMR level. Each fine level advanced is over an interval $(1/R)dt$, if the fine cells are a factor of R smaller, and in this scenario, the coarser level provides Dirichlet boundary condition data for the fine-level advances. Note that the levels are properly nested so that the finer level is fully contained within the coarser level, except perhaps at physical boundaries, where their edges can be coincident - thus, the fine level has sufficient boundary data for a well-posed advance.

After two adjacent levels in the hierarchy reach the same physical time, a *synchronization* operation is performed to ensure that the coarse data is consistent with the volume integral of the fine data that covers it, and the fluxes across of the coarse-fine interface are those of the fine solution. The latter of these two operations can be quite complex, as it must correct coarse-grid errors committed by each of the operators used to perform the original advance. It may also be non-local, in that cells far away from the coarse-fine interface may need to incorporate flux increments due to the mismatched coarse and fine solutions. Formally, the synchronzation is a bilevel correction that should be computed as a sequence of two-level solves. However, this would lead to the same amound of work that was required to create the original (pre-sync) data. We assume that the corrections computed for the synchronization are smooth enough to be well represented by an increment on the coaser of the two-levels, and interpolated to the finer grid. Note that the transport coefficients are not updated to account for the state changes during the synchronization.

Generically, the synchronization procedure in *PeleLM* follows that described for the *IAMR* code, but with modifications to explicitly enforce that the sum of the species diffusion correction fluxes is zero, that the nonlinear enthalpy update is solved (similar to described above for the single-level advance), and the corection for the advection velocity is adjusted iteratively so that the final synchronized state satisfies the EOS.

There are several components in *PeleLM* that contribute to the flux mismatch at the coarse-fine interface. The first component arise from the face-centered velocity, $U^{ADV,\ell}$, used to advect the scalars at each AMR level ℓ, since the field satifies a divergence constraint on the coarse and fine levels separately. We compute a velocity mismatch

$$\delta U^{ADV,\ell} = -U^{ADV,\ell,n+1/2} + \frac{1}{R^{d-1}} \sum_{k=0}^{R-1} \sum_{edges} U^{ADV,\ell+1,n+k+1/2}$$

(where d is the number of spatial dimensions) along the coarse-fine boundary. We then solve the elliptic projection equation

$$D^{MAC} \frac{1}{\rho} \delta e^{\ell} = D^{MAC} \delta U^{ADV,\ell} + \delta_{\chi}^{\ell}$$

where δ_{χ}^{ℓ} is incremented iteratively to enfoce the final state to satisfy the EOS and compute the correction velocity

$$U^{ADV,\ell,corr} = -\frac{1}{\rho} G^{MAC} \delta e^{\ell}$$

which is the increment of velocity required to carry advection fluxes needed to correct the errors made by advancing the coarse state with the wrong velocities.

The second part of the mismatch arises because the advective and diffusive fluxes on the coarse grid were computed without explicitly accounting for the fine grid, while on the fine grid the fluxes were computed using coarse-grid

Dirichlet boundary data. We define the flux discrepancies on the coarser level ℓ of the pair of levels considered:

$$\delta\boldsymbol{\mathcal{F}}^{\boldsymbol{\ell}} = \Delta t^{\ell}\Big(- \boldsymbol{\mathcal{F}}^{\ell,n+1/2} + \frac{1}{R^{d-1}}\sum_{k=0}^{R-1}\sum_{edges}\boldsymbol{\mathcal{F}}^{\ell+1,n+k+1/2}\Big)$$

where $\boldsymbol{\mathcal{F}}$ is the total (advective + diffusive) flux through a face on the coarse-fine interface prior the synchronization operations. Since all operations are performed on the coarse level we will drop the ℓ in the following.

Since mass is conserved, corrections to density, $\delta\rho^{sync}$ on the coarse grid associated with mismatched advection fluxes may be computed explicitly

$$\delta\rho^{sync} = -D^{MAC}\Big(\sum_m U^{ADV,corr}\rho Y_m\Big)^{n+1/2} + \sum_m \nabla\cdot\delta\boldsymbol{\mathcal{F}}_m$$

We can compute the post-sync new-time value of density, $\rho^{n+1} = \rho^{n+1,p} + \delta\rho^{sync}$, where ^{p} denotes *pre*-sync quantities. The synchronization correction of a state variable $\delta(\rho\phi)^{sync}$ (where $\phi \in (Y_m, h)$) is obtained by subtracting the pre-sync state value $(\rho\phi)^{n+1,p}$ from the corrected one $(\rho\phi)^{n+1}$, both of which expressed from an SDC iteration update (see Eq. (2.2) and (2.3)) but with the divergence of the correction velocity fluxes ($U^{ADV,corr}$) and fluxes mismatch ($\delta\boldsymbol{\mathcal{F}}$) included in the advection and diffusion corrected operators.

Given the corrected density we can decompose the sync corrections $\delta(\rho\phi)^{sync} = \phi^{n+1,p}\delta\rho^{sync} + \rho^{n+1}\delta\phi^{sync}$ and obtain the linear system for $\delta\phi^{sync}$ since the fluxes mismatch contain implicit diffusion fluxes from the Crank-Nicolson discretization. For species m the implicit system reads:

$$\rho^{n+1}\widetilde{\delta Y_m^{sync}} - \Delta t\nabla\cdot\widetilde{\mathcal{F}_{\Updownarrow}}(\widetilde{\delta Y_m^{sync}}) = -D^{MAC}(U^{ADV,corr}\rho Y_m)^{n+1/2} + \nabla\cdot\delta\boldsymbol{\mathcal{F}}_m - Y_m^{n+1,p}\delta\rho^{sync} \tag{2.4}$$

where $\widetilde{\mathcal{F}_{\Updownarrow}}$ is the species correction flux due to the sync correction, $\widetilde{\delta Y_m^{sync}}$. However, as in the single-level algorithm, the species fluxes must be corrected to sum to zero. These adjusted fluxes are then used to recompute a δY_m^{sync}, which is then used via the expression above to compute $\delta(\rho Y_m)^{sync}$, the increment to the species mass densities.

In order to get the equation for the enthalpy sync correction, we operate as for species mass fractions. We will present the details of the method. The SDC advection-diffusion udpate for *pre*-sync enthalpy is (2.3) (now including the superscript p) and its corrected counterpart reads:

$$\begin{aligned}\frac{\rho^{n+1}h_{\mathrm{AD}}^{n+1} - (\rho h)^n}{dt} &= A_h^{n+1/2,(k+1),*} + D_T^{n+1,(k+1),*} + H^{n+1,(k+1),*} \\ &\quad + \frac{1}{2}\Big(D_T^{n,*} - D_T^{n+1,(k),*} + H^{n,*} - H^{n+1,(k),*}\Big)\end{aligned} \tag{2.5}$$

where

$$A_h^{n+1/2,(k+1),*} = -\nabla\cdot(\rho h(U^{ADV}+U^{ADV,corr})^{n+1/2,(k+1)} + \delta\boldsymbol{\mathcal{F}}_{Adv,h}) = A_h^{n+1/2,(k+1),p} - \nabla\cdot(\rho h(U^{ADV,corr})^{n+1/2,(k+1)} + \delta\boldsymbol{\mathcal{F}}_{Adv,h})$$

$$D_T^{n,*} = \nabla\cdot(\lambda^n\nabla T^n + \delta\boldsymbol{\mathcal{F}}_{DT,h}^n) = D_T^{n,p} + \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DT,h}^n)$$

$$D_T^{n+1,(k),*} = \nabla\cdot(\lambda^{n+1,(k)}\nabla T^{n+1,(k)} + \delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k)}) = D_T^{n+1,(k),p} + \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k)})$$

$$D_T^{n+1,(k+1),*} = \nabla\cdot(\lambda^{n+1,(k+1),p}\nabla(T^{n+1,(k+1),p}+\delta T^{sync}) + \delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k+1)}) = D_T^{n+1,(k+1),p} + \nabla\cdot(\lambda^{n+1,(k+1),p}\nabla\delta T^{sync} + \delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k+1)})$$

$$H^{n,*} = -\nabla\cdot(\sum_m h_m(T^n)\,\boldsymbol{\mathcal{F}}_m^n + \delta\boldsymbol{\mathcal{F}}_{DH,h}^n) = H^{n,p} - \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DH,h}^n)$$

$$H^{n+1,(k),*} = -\nabla\cdot(\sum_m h_m(T^{n+1,(k)})\,\boldsymbol{\mathcal{F}}_m^{n+1,(k)} + \delta\boldsymbol{\mathcal{F}}_{DH,h}^{n+1,(k)}) = H^{n+1,(k),p} - \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DH,h}^{n+1,(k)})$$

$$H^{n+1,(k+1),*} = -\nabla\cdot(\sum_m h_m(T^{n+1,(k+1),p}+\delta T^{sync})\,\boldsymbol{\mathcal{F}}_m^{n+1,(k+1)} + \delta\boldsymbol{\mathcal{F}}_{DH,h}^{n+1,(k+1)})$$

and with $\delta T^{sync} = T^{n+1,(k+1)} - T^{n+1,(k+1),p}$. Subtracting the *pre*-sync eq. (2.3) (with the superscript p) from the above equation (2.5) and gathering the flux mismatches and correction velocity fluxes in S_h^{sync} we obtain:

$$\frac{\delta(\rho h)^{sync}}{\Delta t} = S_h^{sync} + D_T^{n+1,(k+1)} - D_T^{n+1,(k+1),p} + H^{n+1,(k+1)} - H^{n+1,(k+1),p} \tag{2.6}$$

where

$$S_h^{sync} = -\nabla\cdot(\rho h(U^{ADV,corr})^{n+1/2,(k+1)} + \delta\boldsymbol{\mathcal{F}}_{Adv,h}) + \frac{1}{2}\nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DT,h}^n - \delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k)}) - \frac{1}{2}\nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DH,h}^n - \delta\boldsymbol{\mathcal{F}}_{DH,h}^{n+1,(k)}) + \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DT,h}^{n+1,(k+1)}) - \nabla\cdot(\delta\boldsymbol{\mathcal{F}}_{DH,h}^{n+1,(k+1)})$$

and the updated $(n+1)$ fluxes (without the $*$) only now contain the implicit contribution, i.e:

$$D_T^{n+1,(k+1)} = D_T^{n+1,(k+1),p} + \nabla\cdot(\lambda^{n+1,(k+1),p}\nabla\delta T^{sync})$$
$$H^{n+1,(k+1)} = -\nabla\cdot(\sum_m h_m(T^{n+1,(k+1),p}+\delta T^{sync})\,\boldsymbol{\mathcal{F}}_m^{n+1,(k+1)}$$

To go further we note that:

$$D_T^{n+1,(k+1)} - D_T^{n+1,(k+1),p} = \nabla\cdot(\lambda^{n+1,(k+1),p}\nabla\delta T^{sync})$$
$$H^{n+1,(k+1)} - H^{n+1,(k+1),p} = -\nabla\cdot\sum_m(h_m(T^{n+1,(k+1),p}+\delta T^{sync})\,\delta\boldsymbol{\mathcal{F}}_m^{sync} + \delta h_m^{sync}\,\boldsymbol{\mathcal{F}}_m^{n+1,(k+1),p})$$

where $\delta h_m^{sync} = h_m(T^{n+1,(k+1)}) - h_m(T^{n+1,(k+1),p})$ and $\delta \boldsymbol{\mathcal{F}}_m^{sync}$ is the species flux increment due to the species sync correction appearing on the LHS of eq. (2.4). Eq. (2.6) is the equation for the sync correction. At this point, we can drop the SDC iteration index $k+1$ for simplicity (all k related quantities are contained in S_h^{sync}). Note that the evaluation of the transport properties is relatively expensive, such that we don't want to update the conductivity in D_T^{n+1} since a lagged (pre-sync) version is sufficient for second-order accuracy. However we do want to use an updated version of h_m.

Just as in the level advance, we cannot compute h^{n+1} directly, so we solve this iteratively based on the approximation $h^{n+1,\eta+1} \approx h^{n+1,\eta} + C_p^{n+1,\eta}\Delta T^{\eta+1}$, with $\Delta T^{\eta+1} = T^{n+1,\eta+1} - T^{n+1,\eta}$, and iteration index, η = 1: η_{MAX}. The sync equation is thus recast into a linear equation for $\Delta T^{\eta+1}$, and we lag the H terms in iteration η,

$$\begin{aligned}
&\rho^{n+1}C_p^{n+1,\eta}\Delta T^{\eta+1} - dt\,\nabla\cdot\lambda^{(n+1,p)}\nabla(\Delta T^{\eta+1}) \\
&= \rho^{(n+1,p)}h^{(n+1,p)} - \rho^{n+1}h^{n+1,\eta} + dt\Big(S_h^{sync} + \nabla\cdot\lambda^{(n+1,p)}\nabla(\delta T^{sync,\eta}) \\
&\qquad -\nabla\cdot\sum_m\Big(h_m^{n+1}\delta\boldsymbol{\mathcal{F}}_m^{sync} + \delta h_m^{sync}\boldsymbol{\mathcal{F}}_m^{(m+1)}\Big)\Big)
\end{aligned}$$

After each :math:eta iteration, update $T^{n+1,\eta+1} = T^{n+1,\eta} + \Delta T^{\eta+1}$, $\delta T^{sync,\eta+1} = T^{n+1,\eta+1} - T^{(n+1,p)}$, and re-evaluate $(C_p, h_m)^{n+1,\eta+1}$ using $(T^{n+1,\eta+1}, Y_m^{n+1})$. Iterations are continued until the norm of $\Delta T^{\eta+1}$ drops below a tolerance threshold. Then set $T^{n+1} = T^{(n+1,p)} + \delta T^{sync,\eta_{MAX}}$, and compute $h^{n+1} = h(T^{n+1}, Y_m^{n+1})$.

CHAPTER 3

Setting up a new *PeleLM* Case

In order to set up and run a new case in *PeleLM*, the user must provide problem-specific code for two main tasks

- Initial conditions
- Boundary conditions

These functions are typically collected into a single subfolder in `${PELELM_HOME}/Exec`, such as `FlameSheet`. The user can organize these tasks in any way that is convenient - the examples distributed with `PeleLM` represent a certain style for managing this with some level of flexibility, but the basic requirement is simply that source be linked into the build for the functions `pelelm_initdata` for initial conditions and `bcnormal` for boundary conditions.

3.1 Initial Conditions

At the beginning of a *PeleLM* run, for each level, after grids are generated, the cell-centered values of the state must be initialized. In the code, this is done in an `MFIter` loop over grids, and a call to the user's initialization function, `pelelm_initdata`, that must be provided:

```
for (MFIter mfi(S_new,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    const Box& box = mfi.validbox();
    auto sfab = S_new.array(mfi);

    amrex::ParallelFor(box,
    [=] AMREX_GPU_DEVICE (int i, int j, int k) noexcept
    {
      pelelm_initdata(i, j, k, sfab, geomdata, *lprobparm, lpmfdata);
    });
}
```

where `(i,j,k)` are the cell indices, `sfab` is a light data pointer to the initial state MultiFab and `geomdata`, `lprobparm` and `lpmfdata` are container for the geometry, user-define input and PMF data. The associated user function (in pelelm_prob.H) will provide a value for each entry of the state (velocity, density, mass fraction, ...) :

```
AMREX_GPU_DEVICE
AMREX_FORCE_INLINE
void
pelelm_initdata (int i, int j, int k,
                 amrex::Array4<amrex::Real> const& state,
                 amrex::GeometryData const& geomdata,
                 ProbParm const& prob_parm,
                 PmfData const *pmf_data)
{

   const amrex::Real* prob_lo = geomdata.ProbLo();
   const amrex::Real* prob_hi = geomdata.ProbHi();
   const amrex::Real* dx      = geomdata.CellSize();

   const amrex::Real z = prob_lo[2] + (k+0.5)*dx[2];
   const amrex::Real y = prob_lo[1] + (j+0.5)*dx[1];
   const amrex::Real x = prob_lo[0] + (i+0.5)*dx[0];

   constexpr amrex::Real Pi = 3.14159265358979323846264338327950288;
   const amrex::Real L_x = prob_hi[0] - prob_lo[0];
   const amrex::Real L_y = prob_hi[1] - prob_lo[1];


   ...


   state(i,j,k,DEF_Temp) = prob_parm.Temp;

   for (int n = 0; n < NUM_SPECIES; n++){
     massfrac[n] = 1.0/NUM_SPECIES;
   }


   state(i,j,k,Xvel) = 0.0;
   state(i,j,k,Yvel) =  prob_parm.Vel;
#elif ( AMREX_SPACEDIM == 3 )
   state(i,j,k,Zvel) = 0.0;
#endif

   amrex::Real rho_cgs, P_cgs;
   P_cgs = prob_parm.P_mean * 10.0;

   auto eos = pele::physics::PhysicsType::eos();
   eos.PYT2R(P_cgs, massfrac, state(i,j,k,DEF_Temp), rho_cgs);
   state(i,j,k,Density) = rho_cgs * 1.0e3;            // CGS -> MKS conversion

   eos.TY2H(state(i,j,k,DEF_Temp), massfrac, state(i,j,k,DEF_RhoH));
   state(i,j,k,DEF_RhoH) = state(i,j,k,DEF_RhoH) * 1.0e-4 * state(i,j,k,Density);   //
↪ CGS -> MKS conversion

   for (int n = 0; n < NUM_SPECIES; n++) {
     state(i,j,k,DEF_first_spec+n) = massfrac[n] * state(i,j,k,Density);
   }
}
```

Note home the geometry data are retrived from the `geomdata` object to obtain the coordinate of each cell center. The state data indices (`Xvel`, `Yvel`, `Density`, ...) are prescribed in the `$(PELELM_HOME)/Source/IndexDefines.H` file. Note that the conserved states are stored for species and enthalpy (i.e., ρY_i and ρh); these

are the variables that the user must fill in the initial and boundary condition routines. Typically, however, the primitive state (i.e., Y_i and T) is known directly. If that is the case, the user can make use of the compiled-in model-specific equation-of-state routines (`eos.`) to translate primitive to conserved state values. Consult the example setups provided to see how to call these routines, and how to load the final values required for initial data.

The runtime option (such as initial temperature, inlet velocity, ...) are gathered in the `ProbParm` C++ structure defined in `pelelm_prob_parm.H` and filled from the input file in `pelelm_prob.cpp` using *AMReX* parser. This structure can be modified by the user to hold any data necessary for initial or boundary conditions.

3.2 Boundary Conditions

In *PeleLM*, a single function is used to fill all the state component at physical boundaries. The function `bcnormal` is in the `pelelm_prob.H` file. The main objective of this function is to fill the `s_ext` array fill boundary state data. The function `bcnormal` is called on each side (*lo* or *hi*) for each spatial dimension and will be used to fill the ghost cells of the state variables for which the *PeleLM* internal boundary condition is `EXT_DIR` (external Dirichlet) on that face. For example, specifying a *PeleLM* `Inflow` boundary condition on the lower face in the y direction in the input file leads to an `EXT_DIR` for species mass fraction, which then need to be provided in `bcnormal`. An example of the `bcnormal` of the FlameSheet is presented here:

```
AMREX_GPU_DEVICE
AMREX_FORCE_INLINE
void
bcnormal(
  const amrex::Real x[AMREX_SPACEDIM],
  amrex::Real s_ext[DEF_NUM_STATE],
  const int idir,
  const int sgn,
  const amrex::Real time,
  amrex::GeometryData const& geomdata,
  ProbParm const& prob_parm,
  ACParm const& ac_parm,
  PmfData const *pmf_data)
{
  const amrex::Real* prob_lo = geomdata.ProbLo();
  const amrex::Real* prob_hi = geomdata.ProbHi();
  amrex::GpuArray<amrex::Real, NUM_SPECIES + 4> pmf_vals = {0.0};
  amrex::Real molefrac[NUM_SPECIES] = {0.0};
  amrex::Real massfrac[NUM_SPECIES] = {0.0};

  if (sgn == 1) {
    PMF::pmf(pmf_data,prob_lo[idir], prob_lo[idir], pmf_vals);

    s_ext[Xvel] = 0.0;
#if ( AMREX_SPACEDIM == 2 )
    s_ext[Yvel] = pmf_vals[1]*1e-2;
#elif (AMREX_SPACEDIM == 3)
    s_ext[Yvel] = 0.0;
    s_ext[Zvel] = pmf_vals[1]*1e-2;
#endif

    s_ext[DEF_Temp] = pmf_vals[0];

    for (int n = 0; n < NUM_SPECIES; n++){
      molefrac[n] = pmf_vals[3 + n];
    }
```

(continues on next page)

(continued from previous page)

```
    auto eos = pele::physics::PhysicsType::eos();
    eos.X2Y(molefrac, massfrac);

    amrex::Real rho_cgs, P_cgs, RhoH_temp;
    P_cgs = prob_parm.P_mean * 10.0;

    eos.PYT2R(P_cgs, massfrac, s_ext[DEF_Temp], rho_cgs);
    s_ext[Density] = rho_cgs * 1.0e3;

    eos.TY2H(s_ext[DEF_Temp], massfrac, RhoH_temp);
    s_ext[DEF_RhoH] = RhoH_temp * 1.0e-4 * s_ext[Density];   // CGS -> MKS conversion

    for (int n = 0; n < NUM_SPECIES; n++) {
      s_ext[DEF_first_spec+n] = massfrac[n] * s_ext[Density];
    }
  }
}
```

The `sgn` input takes a value of 1 on the low face and -1 on the high face, while `ìdir` provide the spatial direction (0, 1 or 2 corresponding to X, Y or Z, respectively). This allow to differentiate between the various boundary conditions when more than 1 `ÈXT_DIR` is needed. In this example, the boundary conditions are extracted from a pre-computed premixed flame which data are stored in the `pmf_data` structure.

Here, we've made use of a local convenience function, `bcnormal` endowed with the knowledge of all boundary values, and extract the appropriate quantity from the results of that call. This was done to localize all boundary condition calculations to a single routine in the code, and helps to preserve consistency. This is only one style though, and as long as appropriate Dirichlet values are set for this state, it makes no difference how the work is organized. For example, data may be provided by interpolating "live data" being actively generated by a co-running separate code, by interpolating data files, evaluating functional forms, etc.

Note that although the array structure to be filled contains valid cell-centered state data where it overlaps the valid domain, the values set in the grow cells of the container will be applied on the boundary face of the corresponding cells. Internally, all *PeleLM* code understands to apply Dirichlet conditions on the boundary faces.

3.3 Refinement Criteria

The dynamic creation and destruction of grid levels is a fundamental part of *PeleLM*'s capabilities. The process for this is described in some detail in the *AMReX* documentation, but we summarize the key points here.

At regular intervals (set by the user), each Amr level that is not the finest allowed for the run will invoke a "regrid" operation. When invoked, a list of error tagging functions is traversed. For each, a field specific to that function is derived from the state over the level, and passed through a kernel that "set"'s or "clear"'s a flag on each cell. The field and function for each error tagging quantity is identified in the setup phase of the code where the state descriptors are defined (i.e., in *PeleLM_setup.cpp*). Each function in the list adds or removes to the list of cells tagged for refinement. This final list of tagged cells is sent to a grid generation routine, which uses the Berger-Rigoutsos algorithm to create rectangular grids which will define a new finer level (or set of levels). State data is filled over these new grids, copying where possible, and interpolating from coarser level when no fine data is available. Once this process is complete, the existing Amr level(s) is removed, the new one is inserted into the hierarchy, and the time integration continues.

The traditional *AMReX* approach to setting up and controlling the regrid process involves explicitly creating ("hard coding") a number of functions directly into *PeleLM*'s setup code. (Consult the source code and *AMReX* documentation for precisely how this is done). *PeleLM* provides a limited capability to augment the standard set of error functions that is based entirely on runtime data specified in the inputs (ParmParse) data. The following example portion of a ParmParse'd input file demonstrates the usage of this feature:

```
amr.refinement_indicators = flame_tracer lo_temp gradT

amr.flame_tracer.max_level = 3
amr.flame_tracer.value_greater = 1.e-6
amr.flame_tracer.field_name = Y(H)

amr.lo_temp.max_level = 1
amr.lo_temp.value_less = 450
amr.lo_temp.field_name = temp

amr.gradT.max_level = 2
amr.gradT.adjacent_difference_greater = 20
amr.gradT.field_name = temp
amr.gradT.start_time = 0.001
amr.gradT.end_name = 0.002
```

Here, we have added three new custom-named criteria – `flame_tracer`: cells with the mass fraction of H greater than 1 ppm; `lo_temp`: cells with T less than 450K, and `gradT`: cells having a temperature difference of 20K from that of their immediate neighbor. The first will trigger up to Amr level 3, the second only to level 1, and the third to level 2. The third will be active only when the problem time is between 0.001 and 0.002 seconds.

Note that these additional user-created criteria operate in addition to those defined as defaults. Also note that these can be modified between restarts of the code. By default, the new criteria will take effect at the next scheduled regrid operation. Alternatively, the user may restart with `amr.regrid_on_restart = 1` in order to do a full (all-levels) regrid after reading the checkpoint data and before advancing any cells.

CHAPTER 4

Building with GNU Make

The build of *PeleLM* is managed with GNUmake. For a specific case setup and run configuration, you write your own make input files that define a number of variables and build rules, and then invoke `make` to initiate the build process. This will result in an executable upon successful completion. Temporary files generated in the building process (such as object files) are stored in a directory named `tmp_build_dir` in such a way that allows multiple build configurations to co-exist.

4.1 Dissecting a Simple Make File

An example input file for GNU Make can be found in any of the example setup, such as `$(PELELM_HOME)/Exec/RegTests/FlameSheet`. Table 4.1 below shows a list of important variables.

4.1: Important make variables

Variable	Value	Default
AMREX_HOME	Path to amrex	environment
IAMR_HOME	Path to IAMR	environment
PELELM_HOME	Path to PeleLM	environment
PELE_PHYSICS_HOME	Path to PelePhysics	environment
COMP	gnu, cray, ibm, intel, llvm, or pgi	none
DEBUG	TRUE or FALSE	TRUE
DIM	2 or 3	none
USE_MPI	TRUE or FALSE	FALSE
USE_OMP	TRUE or FALSE	FALSE
USE_CUDA	TRUE or FALSE	FALSE
USE_HIP	TRUE or FALSE	FALSE

At the beginning of `$(PELELM_HOME)/Exec/FlameSheet/GNUmakefile`, the make variable `AMREX_HOME` is set to the path to the top directory of AMReX. Note that in the example `?=` is a conditional variable assignment operator that only has an effect if `AMREX_HOME` has not been defined (including in the environment). The make variable can also take it value from the corresponding environment variable, `AMREX_HOME`, if it exists. For example in bash, one can set

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

prior to running `make`. alternatively, in tcsh one can set

```
setenv AMREX_HOME /path/to/amrex
```

Path to *IAMR* (`IAMR_HOME`), *PelePhysics* (`PELE_PHYSICS_HOME`) and *PeleLM* (`PELELM_HOME`) should also be provided in the same manner.

One must set the `COMP` variable to choose a compiler suite (for C, C++, f90). Currently the list of supported compiler suites includes gnu, cray, ibm, intel, llvm, and pgi. One must also set the `DIM` variable to either 1, 2, or 3, depending on the dimensionality of the problem.

Variables `DEBUG`, `USE_MPI`, `USE_OMP`, `USE_CUDA` and `USE_HIP` are optional with default set to TRUE, FALSE, FALSE, FALSE and FALSE, respectively. Note that the last three entries are mutually exclusive. The meaning of these variables should be obvious. When `DEBUG=TRUE`, aggressive compiler optimization flags are turned off and assertions in source code are turned on. For production runs, `DEBUG` should be set to FALSE.

After defining these make variables, an application code may also wish to include its own Make.package file (e.g., `./Make.package`) or otherwise directly append source files to the build system, using operator `+=`. Variables for various source file types are shown below.

CEXE_sources C++ source files. Note that C++ source files are assumed to have a .cpp extension.

CEXE_headers C++ headers with .h or .H extension.

cEXE_sources C source files with .c extension.

cEXE_headers C headers with .h extension.

f90EXE_sources Free format Fortran source with .f90 extension.

F90EXE_sources Free format Fortran source with .F90 extension. Note that these Fortran files will go through preprocessing.

In the `FlameSheet` example, the extra source file, `drm19Soln_seed_0.50.f` is in a directory that is already in the build system's search path. Additional files, that are local to this setup, such as `pele_prob.cpp` need to be added to the appropriate file list explicitly as well. If this case included files in a separate folder (e.g., `mysrcdir`), you will then need to add the following:

```
VPATH_LOCATIONS += mysrcdir
INCLUDE_LOCATIONS += mysrcdir
```

Here `VPATH_LOCATIONS` and `INCLUDE_LOCATIONS` are the search path for source and header files, respectively.

Finally, *PeleLM* requires a number of defines and setup for every case that must be processed into final filelists for building, and various defines for complilation – these are managed in the make include file `$(PELELM_HOME)/Tools/Make/Make.PeleLM`. In particular, this file contains macros to find the chemistry mechanism/model files associated with the string value of the `Chemistry_Model` variable. Look in `$(PELELM_HOME)/Tools/Make/Make.PeleLM` for a list of currently recognized models, and to see which folder that the string maps to in `$(PELE_PHYSICS_HOME)/Support/Fuego/Mechanism/Models` folder. That folder will contain a `Make.package` that appends the model-specific source files to the build list (typically a C-source file generated by *FUEGO* from a CHEMKIN-compatible set of specification files – see the file `$(PELE_PHYSICS_HOME)/README.rst` for more information on model generation.

4.2 Tweaking the Make System

The GNU Make build system is located in the *AMReX* source code distribution in `$(AMREX_HOME)/Tools/GNUMake`. You can read `README.md` and the make files there for more information. Here we will give a brief overview.

Besides building executable, other common make commands include:

`make clean` This removes the executable, .o files, and the temporarily generated files. Note that one can add additional targets to this rule using the double colon (::)

`make realclean` This removes all files generated by make.

`make help` This shows the rules for compilation.

`make print-xxx` This shows the value of variable xxx. This is very useful for debugging and tweaking the make system.

Compiler flags are set in `$(AMREX_HOME)/Tools/GNUMake/comps/`. Note that variables like `CC` and `CFLAGS` are reset in that directory and their values in environment variables are disregarded. Site-specific setups (e.g., the MPI installation) are in `$(AMREX_HOME)/Tools/GNUMake/sites/`, which includes a generic setup in `Make.unknown`. You can override the setup by having your own `sites/Make.$(host_name)` file, where variable `host_name` is your host name in the make system and can be found via `make print-host_name`. You can also have an `$(AMREX_HOME)/Tools/GNUMake/Make.local` file to override various variables. See `$(AMREX_HOME)/Tools/GNUMake/Make.local.template` for an example.

4.3 Specifying your own compiler / GCC on macOS

The `$(AMREX_HOME)/Tools/GNUMake/Make.local` file can also be used to specify your own compile commands by setting the valiables `CXX`, `CC`, `FC`, and `F90`. This might be neccarry if your systems contains non-standard names for compiler commands.

For example, mac OSX Xcode ships with its own (woefully outdated) version of GCC (4.2.1). It is therefore recommended to install GCC using the homebrew package manager. Running `brew install gcc` installs gcc with names reflecting the version number. If GCC 8.2 is installed, homebrew installs it as `gcc-8`. AMReX can be built using `gcc-8` without MPI by using the following `$(AMREX_HOME)/Tools/GNUMake/Make.local`:

```
ifeq ($(USE_MPI),TRUE)
  CXX = mpicxx
  CC  = mpicc
  FC  = mpif90
  F90 = mpif90
else
  CXX = g++-8
  CC  = gcc-8
  FC  = gfortran-8
  F90 = gfortran-8
endif
```

For building with MPI, we assume `mpicxx`, `mpif90`, etc. provide access to the correct underlying compilers.

Note that if you are building *PeleLM* using homebrew's gcc, it is recommended that you use homebrew's mpich. Normally is it fine to simply install its binaries: `brew install mpich`. But if you are experiencing problems, we suggest building mpich usinging homebrew's gcc: `brew install mpich --cc=gcc-8`.

CHAPTER 5

PeleLM control

5.1 Physical Units

PeleLM currently supports only MKS units. All inputs and problem initialization should be specified in MKS; output is in MKS unless otherwise specified.

5.2 Control parameters

The *PeleLM* executable primarily uses a single inputs files at runtime to set and alter the behavior of the algorithm and initial conditions.

The 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` in a parameter listed as `amr.max_grid`). Parameters set here are read using the `ParmParse` class in *AMReX*. The namespaces are typically used to group control parameters by source code class or overall functionality. There are, for example, a large set of parameters that control the generation of the solution-adaptive meshes during the run, as well as the location and content of output files and logging information. There are also a set of parameters that control the details of the `PeleLM` time-stepping strategy, such as the number of SDC iterations taken per time step, solver types and tolerances, and algorithmic variations. These latter control parameters are detailed separately, in *PeleLM algorithm controls*.

5.2.1 Working with `inputs` files

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.

5.2.2 Problem Geometry

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

1. `geometry.prob_lo`: physical location of low corner of the domain (type: `Real`; must be set. A number is needed for each dimension in the problem

2. `geometry.prob_hi`: physical location of high corner of the domain (type: `Real`; must be set. A number is needed for each dimension in the problem

3. `geometry.coord_sys`: coordinate system, 0 = Cartesian, 1 = rz (2D only), 2 = spherical (1D only); must be set.

4. `geometry.is_periodic`: is the domain periodic in this direction? `0` if false, `1` if true (default: `0 0 0`). An integer is needed for each dimension in the problem

As an example, the following:

```
geometry.prob_lo = -0.1 -0.1 0.0
geometry.prob_hi = +0.1 +0.1 0.2
geometry.coord_sys = 0
geometry.is_periodic = 0 1 0
```

defines the domain to span the region from (-10,-10,0) cm at the lower left to (10, 10, 20) cm at the upper right in physical coordinates, specifies a Cartesian geometry, and makes the domain periodic in the y-direction only.

5.2.3 Domain Boundary Conditions

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

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

The valid boundary types are:

```
Interior
Inflow
Outflow
Symmetry
SlipWallAdiab
NoSlipWallAdiab
SlipWallIsotherm
NoSlipWallIsotherm
```

Note: `peleLM.lo_bc` and `peleLM.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 `Interior` for that direction.

As an example, the following:

```
peleLM.lo_bc = Inflow SlipWallAdiab Interior
peleLM.hi_bc = Outflow SlipWallAdiab Interior

geometry.is_periodic = 0 0 1
```

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

5.2.4 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 less than or equal to `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). Ratio values must be either or 2 or 4.

```
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` is less than 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`.

5.2.5 Regridding

The details of the regridding strategy are described elsewhere; here we cover how the input parameters can control the gridding. The user defines functions which tag individual cells at a given level if they need refinement (this is discussed in *Refinement Criteria*). This list of tagged cells is sent to a grid generation routine, which uses the Berger-Rigoutsos algorithm 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_grids`, 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 (2D), 32 (3D))

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

- `amr.blocking_factor`: all generated grid dimensions will 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 the number of processors is greater than the number 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`, here 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 each of 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.

5.2.6 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. For example, having grids that are large enough to coarsen multiple levels in a V-cycle is essential for good multigrid performance. 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.

5.2.7 Simulation Time

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

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

5.2.8 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 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 numerical 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.

Occasionally, the user will want to set the timestep explicitly, using

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

If `ns.init_shrink` not equal 1 then the first time step will in fact be `ns.init_shrink` * `ns.fixed_dt`.

5.2.9 Restart

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 $geq 1$; 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
```

5.2.10 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)

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. Also, 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.

5.2.11 User runtime problem data

As mentioned in *Setting up a new PeleLM Case*, the user can specify problem specific data, provided that the appropriate variable has been added to the ProbParm structure defined in pelelm_prob_parm.H and the required ParmParse functions are called in pelelm_prob.cpp. It is customary to prepend the problem specific data with prob as done for example in the FlameSheet case:

```
#----------------------- PROBLEM PARAMETERS----------------------
prob.P_mean = 101325.0
prob.standoff = -.022
prob.pertmag = 0.0004
prob.pmf_datafile = "drm19_pmf.dat"
```

5.2.12 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)
- `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.

5.2.13 *PeleLM* algorithm controls

The following parameters affect detailed aspects of the PeleLM integration algorithm:

- `ns.do_diffuse_sync`: Debugging flag, do or skip diffusion of the mac_sync (int; default: 1)
- `ns.do_reflux_visc`: Debugging flag, do or skip the viscous reflux step (int; default: 1)
- `ns.do_active_control`: Turn on active control of the inflow velocity (int; default: 0)
- `ns.do_active_control_temp`: Turn on active control of the temperature (int; default: 0)

- ns.temp_control: The control temperature, used in ns.do_active_control_temp=1 (Real; default: -1)
- ns.v: Overall timestepping verbosity (int; default: 1)
- ns.divu_ceiling: DEPRECATED (int; default:)
- ns.divu_dt_factor: Safety factor on the estimated divu_dt (Real; default: 1)
- ns.min_rho_divu_ceiling: Minmimum density for computing the divu_dt (Real; default: 0.1)
- ns.htt_tempmin: Minimum allowable temperature during Newtons solves to compute T from RhoH and composition (Real; default: 250)
- ns.htt_tempmax: Maximum allowable temperature during Newtons solves to compute T from RhoH and composition (Real; default: 3000)
- ns.floor_species: Flag, should the species be floored to zero throughout the time-stepping algorithm (int; default: 0)
- ns.do_set_rho_to_species_sum: Flag, show the density be replaced by the sum of the species density throughout the time-stepping algorithm (int; default: 1)
- ns.num_divu_iters: Number of passes during initialization that the dt is adjusted for the purposes of computing divu prior to init_iters (int; default: 3)
- ns.do_not_use_funccount: Flag, do not use work estimate to rebalance workloads during chemistry advance (int; default: 0)
- ns.unity_Le: Deprecated (int; default:)
- ns.sdc_iterMAX: Maximum number of SDC iterations in the level advance (int; default: 1)
- ns.num_mac_sync_iter: Maximum number of iterations taken during the mac_sync operation for the correction velocity (int; default: 1)
- ns.thickening_factor: A multiplier that is applied to both the transport and reaction rates to artificially thicken a computed flame while preserving its propagation speed (int; default: 1)
- ns.hack_nochem: Debug flag to shut off chemical reactions in the level advance (int; default: 0)
- ns.hack_nospecdiff: Debug flag to shut off species transport in the level advance (int; default: 0)
- ns.hack_noavgdivu: Flag, do not average down divu, and thus replace the velocity divergence computed on covered coarse cells (int; default: 1)
- ns.do_check_divudt: Flag, check after the fact if the divu dt condition was violated, now that we have the mac velocities (int; default: 1)
- ns.avg_down_chem: Flag, rather than doing chemical advance on covered coarse cells, average down the reaction source from fine cells of the previous time step (an attempt to avoid computing chemistry with averaged down states) (int; default: 0)
- ns.reset_typical_vals_int: Interval (in coarse time steps) between resetting the typical values of all states via scanning the solution (int; default: -1 [do no reset])
- ns.do_OT_radiation: Flag, add optically-thin radiative energy loss (phenomenalogical expressions bassed on specific molecules present in the run) (int; default: 0)
- ns.do_heat_sink: Flag, add user-specific term to inject/remove energy locally (int; default: 0)
- ns.use_tranlib: Deprecated (int; default:)
- ns.turbFile: Deprecated (int; default:)

- ns.zeroBndryVisc: Flag, call user function to modify transport coefficients on cell faces at the physical domain (in order to effectively change a local boundary condition from Dirichlet to Neumann) (int; default: 1)
- ns.scal_diff_coefs: Deprecated (int; default:)
- amr.probin_file: Name of text file to search for Fortran namelists used to set problem-specific setup/helper variables (int; default: probin)
- ShowMF_Sets: Debugging tool, write all ShowMF MultiFabs tagged with one of the strings listed here (list of string; default: "")
- ShowMF_Dir: Debugging tool, Folder where the ShowMF sets are written (int; default:)
- ShowMF_Verbose: Debugging tool, write to stdio whenever ShowMF sets are written (int; default: 0)
- ShowMF_Check_Nans: Debugging tool, flag to check for NaNs in the ShowMF sets begin written (int; default: 0)
- ShowMF_Fab_Format: Debugging tool, format of ShowMF set files (string; default:)
- peleLM.num_forkjoin_tasks: Number of fork-join tasks that the species implicit diffusion solves are split into (int; default: 1)
- peleLM.forkjoin_verbose: Flag, write to stdio some info while forking diffusion work (int; default:)
- peleLM.num_deltaT_iters_MAX: Maximum number of iterations taken to iterative advance the enthalpy equation via temperature solves(int; default:)
- peleLM.deltaT_norm_max: Tolerance of iterative solve for iterative enthalpy solve (Real; default: 1.e-12)
- peleLM.deltaT_verbose: Flag, write to stdio some info during ierative enthalpy solve (int; default: 0)
- ht.chem_box_chop_threshold: Parameter used when refining box layout for chemistry solves (int; default:)
- ht.plot_reactions: Flag, add reactions to plotfiles (int; default: 0)
- ht.plot_consumption: Flag, add rate of consumption to plotfiles (int; default: 0)
- ht.plot_auxDiags: Flag, compute auxiliary diagnostics when doing reactions (int; default: 0)
- ht.plot_heat_release: Flag, add heat release to plotfiles (int; default: 0)
- ht.new_T_threshold: DEPRECATED (int; default:)
- ht.do_curvature_sample: Flag, add curvature of the temperature field to plotfiles (int; default: 0)
- ht.typValY_NAME: Override the typical value used for the chemical species, NAME (int; default: -1 (do not override))
- ht.typValY_Temp: Override the typical value used for the temperature (int; default: -1 (do not override))
- ht.typValY_RhoH: Override the typical value used for the RhoH (int; default: -1 (do not override))
- ht.typValY_Vel: Override the typical value used for the velocity (int; default: -1 (do not override))
- ht.pltfile: Name of pltfile to use for initializing data based on previous calculation (string; default: <blank>)
- ht.velocity_plotfile: Name of a plotfile to use for initializing the velocity field based on a previous calculation (stribng; default: <blank>)
- ht.plot_rhoydot: Flag, add rhoY of all chemical species to plotfiels (int; default: 0)
- ns.fuelName: Name of species to associate with the fuel (string; default: <blank>)
- ns.consumptionName: Name(s) of species to plot the consumption of if plot_consumption = 1 (int; default: <blank>)

- `ns.oxidizerName`: Name of species to associate with the oxidizer (string; default: <blank>)
- `ns.productName`: Name of species to associate with the product (string; default: <blank>)
- `ns.flameTracName`: Name of species to associate with a flame tracer (string; default: <blank>)
- `ns.do_group_bndry_fills`: DEPRECATED (int; default:)
- `ns.speciesScaleFile`: Name of a file containing species scales (string; default: <blank>)
- `ns.verbose_vode`: Flag, write to stdout information associated with the chemistry solve (int; default:)

CHAPTER 6

Visualization

There are several visualization tools that can be used for AMReX plotfiles. 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. Plotfiles can also be viewed using the VisIt, ParaView, and yt packages. Particle data can be viewed using ParaView.

6.1 Amrvis

Our favorite visualization tool is Amrvis. We heartily encourage you to build the `amrvis1d`, `amrvis2d`, and `amrvis3d` executables, and to try using them to visualize your data. A very useful feature is View/Dataset, which allows you to actually view the numbers in a spreadsheet that is nested to reflect the AMR hierarchy – this can be handy for debugging. You can modify how many levels of data you want to see, whether you want to see the grid boxes or not, what palette you use, etc. Below are some instructions and tips for using Amrvis; you can find additional information in Amrvis/Docs/Amrvis.tex (which you can build into a pdf using pdflatex).

1. Download and build :

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

   Then cd into Amrvis/, edit the GNUmakefile by setting `COMP` to the compiler suite you have.

   Type `make DIM=1`, `make DIM=2`, or `make DIM=3` to build, resulting in an executable that looks like amrvis2d. . . ex.

   If you want to build amrvis with `DIM=3`, you must first download and build `volpack`:

   ```
   git clone https://ccse.lbl.gov/pub/Downloads/volpack.git
   ```

   Then cd into volpack/ and type `make`.

   Note: Amrvis requires the OSF/Motif libraries and headers. If you don't have these you will need to install the development version of motif through your package manager. `lesstif` gives some functionality and will allow you to build the amrvis executable, but Amrvis may exhibit subtle anomalies.

On most Linux distributions, the motif library is provided by the `openmotif` package, and its header files (like Xm.h) are provided by `openmotif-devel`. If those packages are not installed, then use the OS-specific package management tool to install them.

You may then want to create an alias to amrvis2d, for example

```
alias amrvis2d /tmp/Amrvis/amrvis2d...ex
```

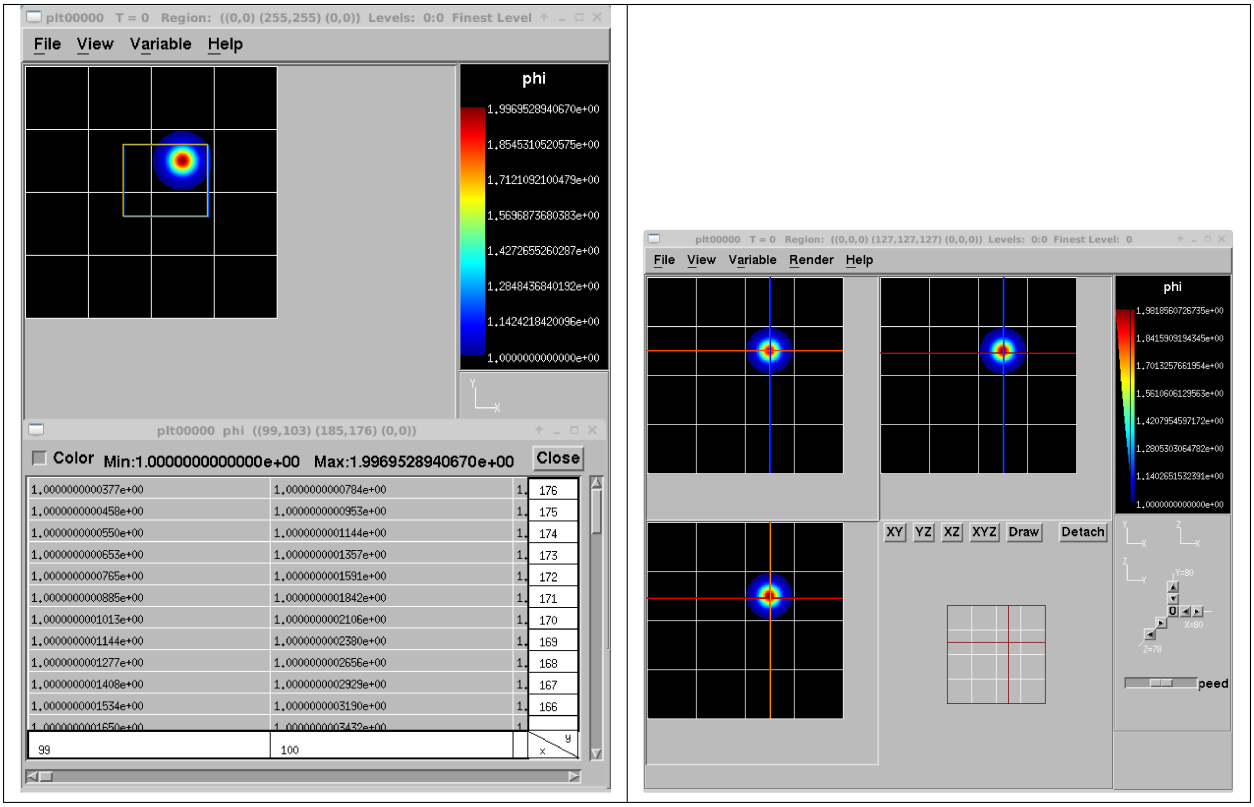
2. Run the command `cp Amrvis/amrvis.defaults ~/.amrvis.defaults`. Then, in your copy, edit the line containing "palette" line to point to, e.g., "palette /home/username/Amrvis/Palette". The other lines control options such as the initial field to display, the number format, widow size, etc. If there are multiple instances of the same option, the last option takes precedence.

3. Generally the plotfiles have the form pltXXXXX (the plt prefix can be changed), where XXXXX is a number corresponding to the timestep the file was output. `amrvis2d <filename>` or `amrvis3d <filename>` to see a single plotfile, or for 2D data sets, `amrvis2d -a plt*`, which will animate the sequence of plotfiles. FArrayBoxes and MultiFabs can also be viewed with the `-fab` and `-mf` options. When opening MultiFabs, use the name of the MultiFab's header file `amrvis2d -mf MyMultiFab_H`.

   You can use the "Variable" menu to change the variable. You can left-click drag a box around a region and click "View" → "Dataset" in order to look at the actual numerical values (see 6.1). Or you can simply left click on a point to obtain the numerical value. You can also export the pictures in several different formats under "File/Export". In 2D you can right and center click to get line-out plots. In 3D you can right and center click to change the planes, and the hold shift+(right or center) click to get line-out plots.

   We have created a number of routines to convert AMReX plotfile data other formats (such as matlab), but in order to properly interpret the hierarchical AMR data, each tends to have its own idiosyncrasies. If you would like to display the data in another format, please contact Marc Day (MSDay@lbl.gov) and we will point you to whatever we have that can help.

6.1: 2D and 3D images generated using Amrvis



6.1.1 Building Amrvis on macOS

As previously outlined at the end of section *Building with GNU Make*, it is recommended to build using the homebrew package manager to install gcc. Furthermore, you will also need x11 and openmotif. These can be installed using homebrew also:

1. `brew cask install xquartz`
2. `brew install openmotif`

Note that when the `GNUmakefile` detects a macOS install, it assumes that dependencies are installed in the locations that Homebrew uses. Namely the `/usr/local/` tree for regular dependencies and the `/opt/` tree for X11.

6.2 VisIt

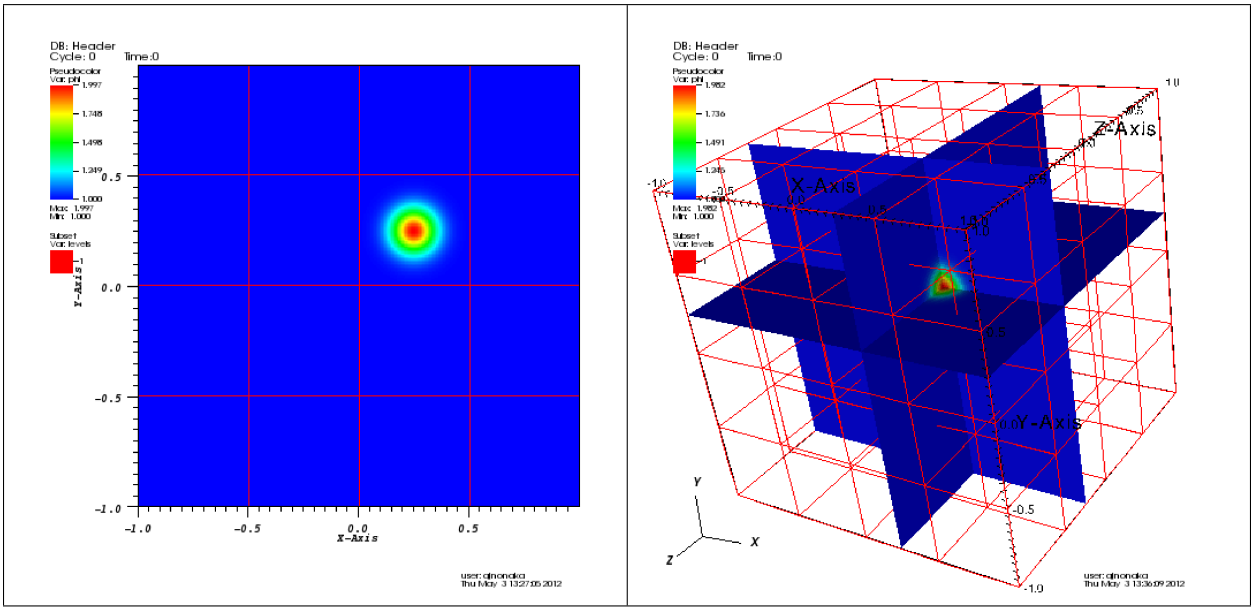
AMReX data can also be visualized by VisIt, an open source visualization and analysis software. To follow along with this example, first build and run the first heat equation tutorial code (see the section on XXX).

Next, download and install VisIt from https://wci.llnl.gov/simulation/computer-codes/visit. To open a single plotfile, run VisIt, then select "File" → "Open file . . . ", then select the Header file associated the the plotfile of interest (e.g., plt00000/Header). Assuming you ran the simulation in 2D, here are instructions for making a simple plot:

- To view the data, select "Add" → "Pseudocolor" → "phi", and then select "Draw".
- To view the grid structure (not particularly interesting yet, but when we add AMR it will be), select " → "subset" → "levels". Then double-click the text "Subset - levels", enable the "Wireframe" option, select "Apply", select "Dismiss", and then select "Draw".
- To save the image, select "File" → "Set save options", then customize the image format to your liking, then click "Save".

Your image should look similar to the left side of 6.2.

6.2: : 2D (left) and 3D (right) images generated using VisIt.



In 3D, you must apply the "Operators" → "Slicing" → "ThreeSlice", with the "ThreeSlice operator attribute" set to `x=0.25`, `y=0.25`, and `z=0.25`. You can left-click and drag over the image to rotate the image to generate something similar to right side of 6.2.

To make a movie, you must first create a text file named `movie.visit` with a list of the Header files for the individual frames. This can most easily be done using the command:

```
~/amrex/Tutorials/Basic/HeatEquation_EX1_C> ls -1 plt*/Header | tee movie.visit
plt00000/Header
plt01000/Header
plt02000/Header
plt03000/Header
plt04000/Header
plt05000/Header
plt06000/Header
plt07000/Header
plt08000/Header
plt09000/Header
plt10000/Header
```

The next step is to run VisIt, select "File" → "Open file . . . ", then select movie.visit. Create an image to your liking and press the "play" button on the VCR-like control panel to preview all the frames. To save the movie, choose "File" → "Save movie . . . ", and follow the on-screen instructions.

6.3 ParaView

The open source visualization package ParaView v5.5 can be used to view 3D plotfiles, and particle data. Download the package at https://www.paraview.org/.

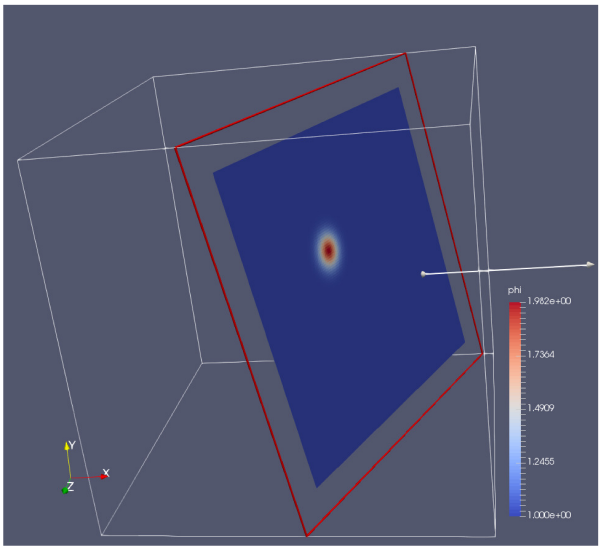
To open a single plotfile (for example, you could run the `HeatEquation_EX1_C` in 3D):

1. Run ParaView v5.5, then select "File" → "Open".
2. Navigate **into** the plotfile directory, and **manually** type in "Header". ParaView will ask you about the file type – choose "Boxlib 3D Files"
3. Under the "Cell Arrays" field, select a variable (e.g., "phi") and click "Apply".
4. Under "Representation" select "Surface".
5. Under "Coloring" select the variable you chose above.
6. To add planes, near the top left you will see a cube icon with a green plane slicing through it. If you hover your mouse over it, it will say "Slice". Click that button.
7. You can play with the Plane Parameters to define a plane of data to view, as shown in 6.1.

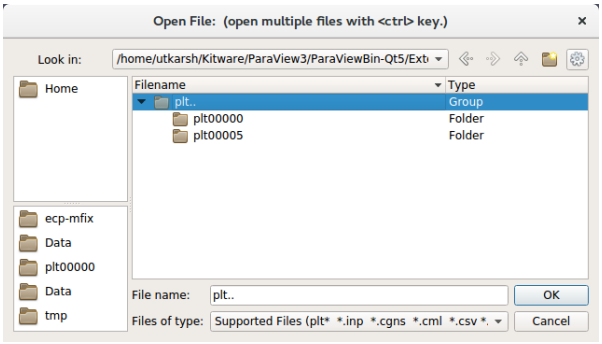
To visualize particle data within plofile directories (for example, you could run the `ShortRangeParticles` example):

1. Run ParaView v5.5, and select then "File" → "Open". You will see a combined "plt.." group. Click on "+" to expand the group, if you want inspect the files in the group. You can select an individual plotfile directory or select a group of directories to read them a time series, as shown in 6.2, and click OK.

1. The "Properties" panel in ParaView allows you to specify the "Particle Type", which defaults to "particles". Using the "Properties" panel, you can also choose which point arrays to read.

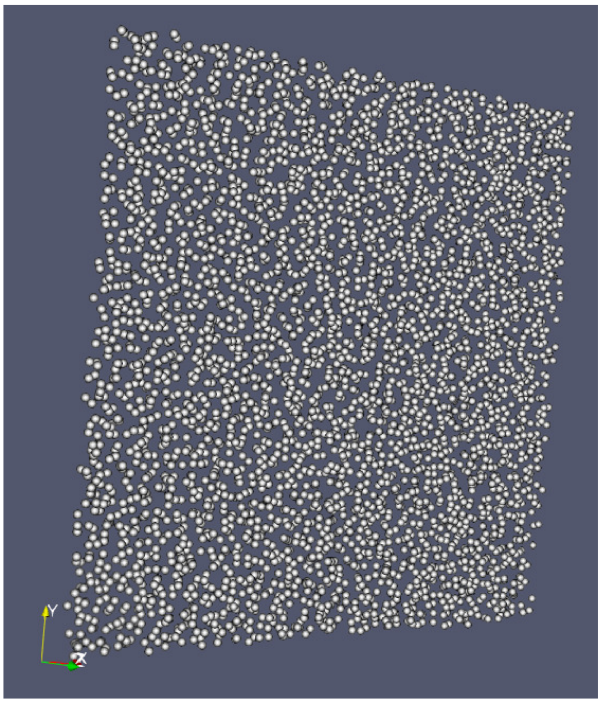


6.1: : Plotfile image generated with ParaView



6.2: : File dialog in ParaView showing a group of plotfile directories selected

2. Click “Apply” and under “Representation” select “Point Gaussian”.
3. Change the Gaussian Radius if you like. You can scroll through the frames with the VCR-like controls at the top, as shown in 6.3.



6.3: : Particle image generated with ParaView

6.4 yt

yt, an open source Python package available at http://yt-project.org/, can be used for analyzing and visualizing mesh and particle data generated by AMReX codes. Some of the AMReX developers are also yt project members. Below we describe how to use on both a local workstation, as well as at the NERSC HPC facility for high-throughput visualization of large data sets.

Note - AMReX datasets require yt version 3.4 or greater.

6.4.1 Using on a local workstation

Running yt on a local system generally provides good interactivity, but limited performance. Consequently, this configuration is best when doing exploratory visualization (e.g., experimenting with camera angles, lighting, and color schemes) of small data sets.

To use yt on an AMReX plot file, first start a Jupyter notebook or an IPython kernel, and import the `yt` module:

```
In [1]: import yt

In [2]: print(yt.__version__)
3.4-dev
```

Next, load a plot file; in this example we use a plot file from the Nyx cosmology application:

```
In [3]: ds = yt.load("plt00401")
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: current_time              = 0.
↪00605694344696544
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_dimensions         = [128␣
↪128 128]
yt : [INFO     ] 2017-05-23 10:03:56,182 Parameters: domain_left_edge          = [ 0.␣
↪ 0.  0.]
yt : [INFO     ] 2017-05-23 10:03:56,183 Parameters: domain_right_edge         = [ 14.
↪24501  14.24501  14.24501]

In [4]: ds.field_list
Out[4]:
[('DM', 'particle_mass'),
 ('DM', 'particle_position_x'),
 ('DM', 'particle_position_y'),
 ('DM', 'particle_position_z'),
 ('DM', 'particle_velocity_x'),
 ('DM', 'particle_velocity_y'),
 ('DM', 'particle_velocity_z'),
 ('all', 'particle_mass'),
 ('all', 'particle_position_x'),
 ('all', 'particle_position_y'),
 ('all', 'particle_position_z'),
 ('all', 'particle_velocity_x'),
 ('all', 'particle_velocity_y'),
 ('all', 'particle_velocity_z'),
 ('boxlib', 'density'),
 ('boxlib', 'particle_mass_density')]
```

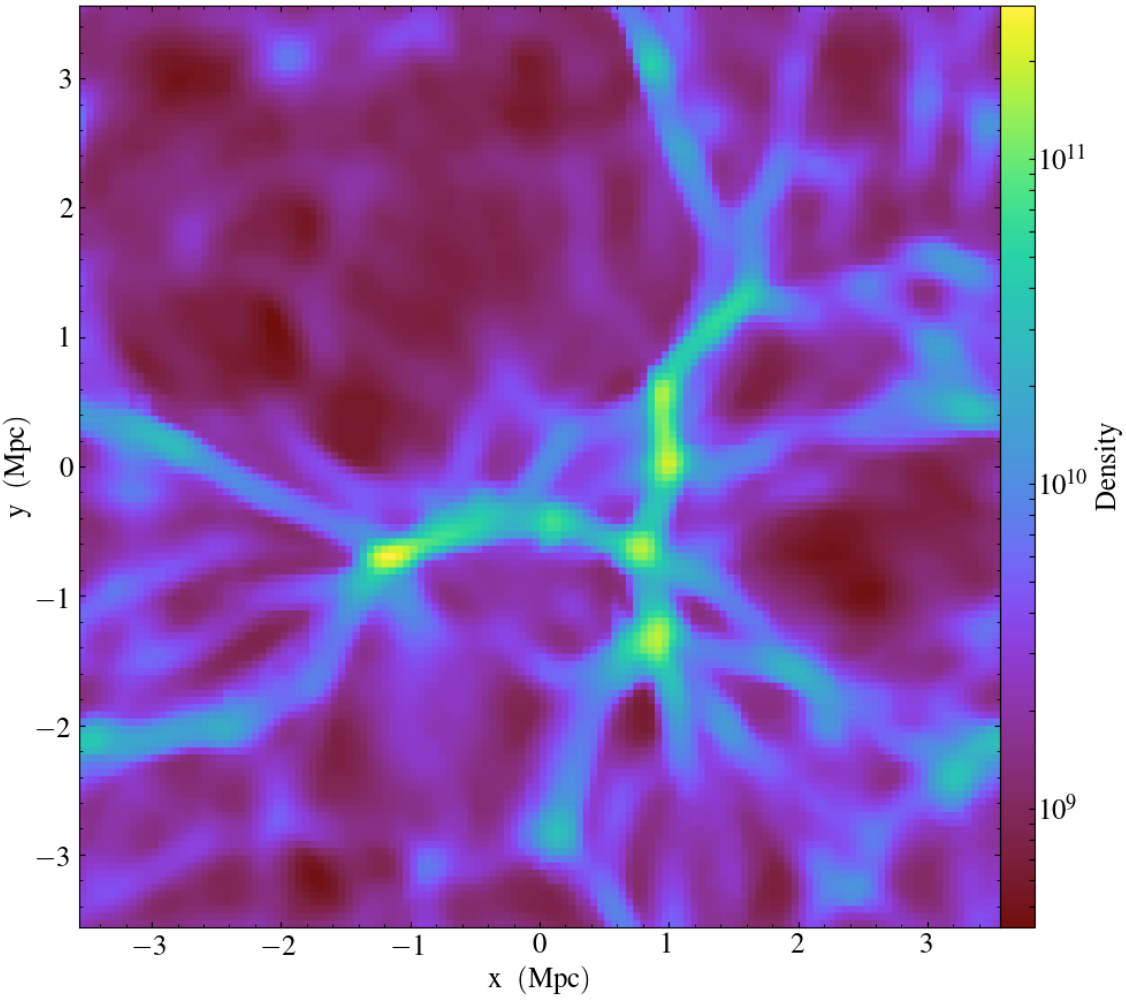
From here one can make slice plots, 3-D volume renderings, etc. An example of the slice plot feature is shown below:

```
In [9]: slc = yt.SlicePlot(ds, "z", "density")
yt : [INFO     ] 2017-05-23 10:08:25,358 xlim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,358 ylim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,359 xlim = 0.000000 14.245010
yt : [INFO     ] 2017-05-23 10:08:25,359 ylim = 0.000000 14.245010

In [10]: slc.show()

In [11]: slc.save()
yt : [INFO     ] 2017-05-23 10:08:34,021 Saving plot plt00401_Slice_z_density.png
Out[11]: ['plt00401_Slice_z_density.png']
```

The resulting image is 6.4. One can also make volume renderings with ; an example is show below:

```
In [12]: sc = yt.create_scene(ds, field="density", lens_type="perspective")

In [13]: source = sc[0]

In [14]: source.tfh.set_bounds((1e8, 1e15))

In [15]: source.tfh.set_log(True)

In [16]: source.tfh.grey_opacity = True

In [17]: sc.show()
```

(continues on next page)



6.4: : Slice plot of 128^3 Nyx simulation using yt.

(continued from previous page)

```
<Scene Object>:
Sources:
    source_00: <Volume Source>:YTRegion (plt00401): , center=[  1.09888770e+25   1.
↪09888770e+25   1.09888770e+25] cm, left_edge=[ 0.  0.  0.] cm, right_edge=[  2.
↪19777540e+25   2.19777540e+25   2.19777540e+25] cm transfer_function:None
Camera:
    <Camera Object>:
    position:[ 14.24501  14.24501  14.24501] code_length
    focus:[ 7.122505  7.122505  7.122505] code_length
    north_vector:[ 0.81649658 -0.40824829 -0.40824829]
    width:[ 21.367515  21.367515  21.367515] code_length
    light:None
    resolution:(512, 512)
Lens: <Lens Object>:
    lens_type:perspective
    viewpoint:[ 0.95423473  0.95423473  0.95423473] code_length

In [19]: sc.save()
yt : [INFO     ] 2017-05-23 10:15:07,825 Rendering scene (Can take a while).
yt : [INFO     ] 2017-05-23 10:15:07,825 Creating volume
yt : [INFO     ] 2017-05-23 10:15:07,996 Creating transfer function
yt : [INFO     ] 2017-05-23 10:15:07,997 Calculating data bounds. This may take a␣
↪while.
Set the TransferFunctionHelper.bounds to avoid this.
yt : [INFO     ] 2017-05-23 10:15:16,471 Saving render plt00401_Render_density.png
```

The output of this is 6.5.

6.4.2 Using yt at NERSC (*under development*)

Because yt is Python-based, it is portable and can be used in many software environments. Here we focus on yt's capabilities at NERSC, which provides resources for performing both interactive and batch queue-based visualization and analysis of AMReX data. Coupled with yt's MPI and OpenMP parallelization capabilities, this can enable high-throughput visualization and analysis workflows.

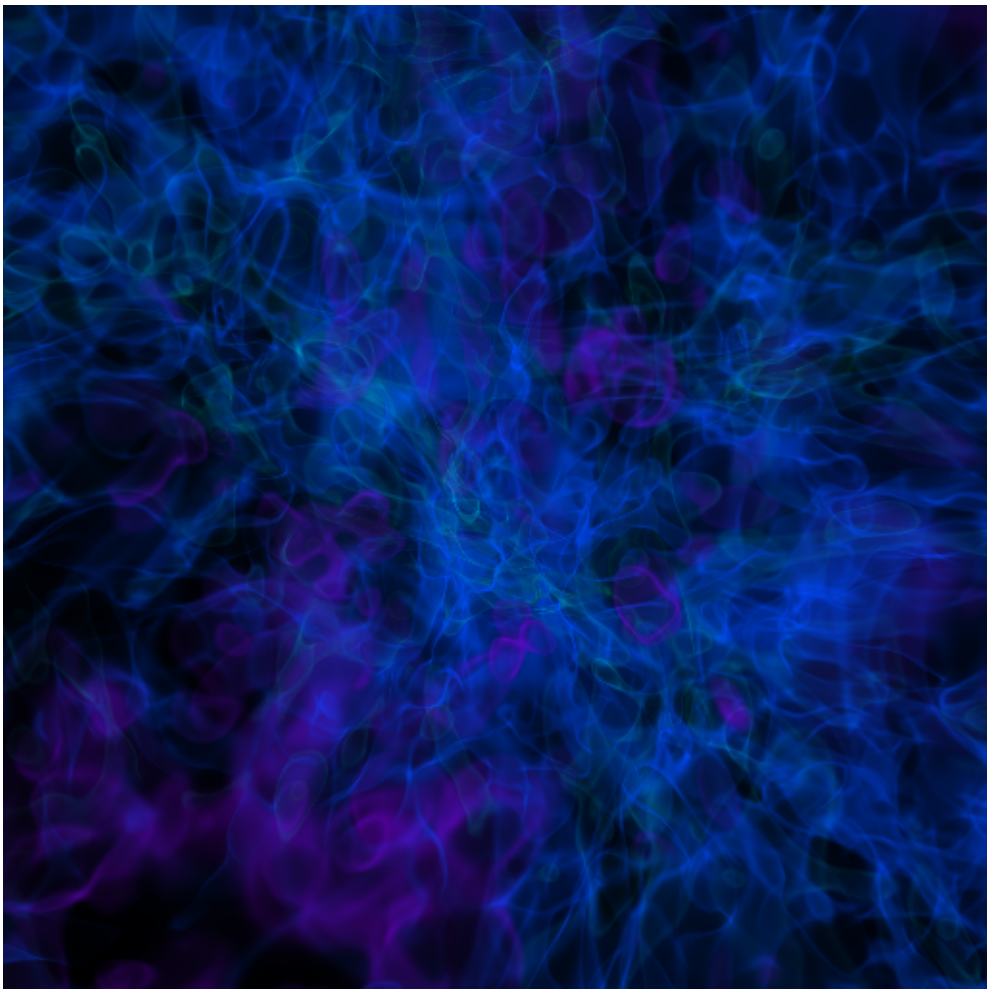
6.4.3 Interactive yt with Jupyter notebooks

Unlike VisIt (see the section on *VisIt*), yt has no client-server interface. Such an interface is often crucial when one has large data sets generated on a remote system, but wishes to visualize the data on a local workstation. Both copying the data between the two systems, as well as visualizing the data itself on a workstation, can be prohibitively slow.

Fortunately, NERSC has implemented several resources which allow one to interact with yt remotely, emulating a client-server model. In particular, NERSC now hosts Jupyter notebooks which run IPython kernels on the Cori system; this provides users access to the `$HOME`, `/project`, and `$SCRATCH` file systems from a web browser-based Jupyter notebook. ***Please note that Jupyter hosting at NERSC is still under development, and the environment may change without notice.***

NERSC also provides Anaconda Python, which allows users to create their own customizable Python environments. It is recommended to install yt in such an environment. One can do so with the following example:

```
user@cori10:~> module load python/3.5-anaconda
user@cori10:~> conda create -p $HOME/yt-conda numpy
user@cori10:~> source activate $HOME/yt-conda
(/global/homes/u/user/yt-conda/) user@cori10:~> pip install yt
```



6.5: Volume rendering of 128^3 Nyx simulation using yt. This corresponds to the same plot file used to generate the slice plot in 6.4.

More information about Anaconda Python at NERSC is here: http://www.nersc.gov/users/data-analytics/data-analytics/python/anaconda-python/.

One can then configure this Anaconda environment to run in a Jupyter notebook hosted on the Cori system. Currently this is available in two places: on https://ipython.nersc.gov, and on https://jupyter-dev.nersc.gov. The latter likely reflects what the stable, production environment for Jupyter notebooks will look like at NERSC, but it is still under development and subject to change. To load this custom Python kernel in a Jupyter notebook, follow the instructions at this URL under the "Custom Kernels" heading: http://www.nersc.gov/users/data-analytics/data-analytics/web-applications-for-data-analytics. After writing the appropriate `kernel.json` file, the custom kernel will appear as an available Jupyter notebook. Then one can interactively visualize AMReX plot files in the web browser.[1]

6.4.4 Parallel

Besides the benefit of no longer needing to move data back and forth between NERSC and one's local workstation to do visualization and analysis, an additional feature of yt which takes advantage of the computational resources at NERSC is its parallelization capabilities. yt supports both MPI- and OpenMP-based parallelization of various tasks, which are discussed here: http://yt-project.org/doc/analyzing/parallel_computation.html.

Configuring yt for MPI parallelization at NERSC is a more complex task than discussed in the official yt documentation; the command `pip install mpi4py` is not sufficient. Rather, one must compile `mpi4py` from source using the Cray compiler wrappers `cc`, `CC`, and `ftn` on Cori. Instructions for compiling `mpi4py` at NERSC are provided here: http://www.nersc.gov/users/data-analytics/data-analytics/python/anaconda-python/#toc-anchor-3. After `mpi4py` has been compiled, one can use the regular Python interpreter in the Anaconda environment as normal; when executing yt operations which support MPI parallelization, the multiple MPI processes will spawn automatically.

Although several components of yt support MPI parallelization, a few are particularly useful:

- **Time series analysis.** Often one runs a simulation for many time steps and periodically writes plot files to disk for visualization and post-processing. yt supports parallelization over time series data via the `DatasetSeries` object. yt can iterate over a `DatasetSeries` in parallel, with different MPI processes operating on different elements of the series. This page provides more documentation: http://yt-project.org/doc/analyzing/time_series_analysis.html#time-series-analysis.
- **Volume rendering**. yt implements spatial decomposition among MPI processes for volume rendering procedures, which can be computationally expensive. Note that yt also implements OpenMP parallelization in volume rendering, and so one can execute volume rendering with a hybrid MPI+OpenMP approach. See this URL for more detail: http://yt-project.org/doc/visualizing/volume_rendering.html?highlight=openmp#openmp-parallelization.
- **Generic parallelization over multiple objects.** Sometimes one wishes to loop over a series which is not a `DatasetSeries`, e.g., performing translational or rotational operations on a camera to make a volume rendering in which the field of view moves through the simulation. In this case, one is applying a set of operations on a single object (a single plot file), rather than over a time series of data. For this workflow, yt provides the `parallel_objects()` function. See this URL for more details: http://yt-project.org/doc/analyzing/parallel_computation.html#parallelizing-over-multiple-objects.

  An example of MPI parallelization in yt is shown below, where one animates a time series of plot files from an IAMR simulation while revolving the camera such that it completes two full revolutions over the span of the animation:

```
import yt
import glob
import numpy as np
```

(continues on next page)

[1] It is convenient to use the magic command `%matplotlib inline` in order to render matplotlib figures in the same browser window as the notebook, as opposed to displaying it as a new window.

(continued from previous page)

```
yt.enable_parallelism()

base_dir1 = '/global/cscratch1/sd/user/Nyx_run_p1'
base_dir2 = '/global/cscratch1/sd/user/Nyx_run_p2'
base_dir3 = '/global/cscratch1/sd/user/Nyx_run_p3'

glob1 = glob.glob(base_dir1 + '/plt*')
glob2 = glob.glob(base_dir2 + '/plt*')
glob3 = glob.glob(base_dir3 + '/plt*')

files = sorted(glob1 + glob2 + glob3)

ts = yt.DatasetSeries(files, parallel=True)

frame = 0
num_frames = len(ts)
num_revol = 2

slices = np.arange(len(ts))

for i in yt.parallel_objects(slices):
    sc = yt.create_scene(ts[i], lens_type='perspective', field='z_velocity
↪')

    source = sc[0]
    source.tfh.set_bounds((1e-2, 9e+0))
    source.tfh.set_log(False)
    source.tfh.grey_opacity = False

    cam = sc.camera

    cam.rotate(num_revol*(2.0*np.pi)*(i/num_frames),
               rot_center=np.array([0.0, 0.0, 0.0]))

    sc.save(sigma_clip=5.0)
```

When executed on 4 CPUs on a Haswell node of Cori, the output looks like the following:

```
user@nid00009:~/yt_vis/> srun -n 4 -c 2 --cpu_bind=cores python make_yt_
↪movie.py
yt : [INFO     ] 2017-05-23 16:51:33,565 Global parallel computation␣
↪enabled: 0 / 4
yt : [INFO     ] 2017-05-23 16:51:33,565 Global parallel computation␣
↪enabled: 2 / 4
yt : [INFO     ] 2017-05-23 16:51:33,566 Global parallel computation␣
↪enabled: 1 / 4
yt : [INFO     ] 2017-05-23 16:51:33,566 Global parallel computation␣
↪enabled: 3 / 4
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: current_time    ␣
↪          = 0.103169376949795
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: domain_
↪dimensions          = [128 128 128]
P003 yt : [INFO     ] 2017-05-23 16:51:33,957 Parameters: domain_left_
↪edge           = [ 0.  0.  0.]
P003 yt : [INFO     ] 2017-05-23 16:51:33,958 Parameters: domain_right_
↪edge          = [ 6.28318531  6.28318531  6.28318531]
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: current_time    ␣
↪          = 0.0
```

(continues on next page)

(continued from previous page)

```
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_
↪dimensions          = [128 128 128]
P002 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: current_time
↪          = 0.0687808060674485
P000 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_left_
↪edge          = [ 0.  0.  0.]
P002 yt : [INFO     ] 2017-05-23 16:51:33,969 Parameters: domain_
↪dimensions          = [128 128 128]
P000 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_right_
↪edge          = [ 6.28318531  6.28318531  6.28318531]
P002 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_left_
↪edge          = [ 0.  0.  0.]
P002 yt : [INFO     ] 2017-05-23 16:51:33,970 Parameters: domain_right_
↪edge          = [ 6.28318531  6.28318531  6.28318531]
P001 yt : [INFO     ] 2017-05-23 16:51:33,973 Parameters: current_time
↪          = 0.0343922351851018
P001 yt : [INFO     ] 2017-05-23 16:51:33,973 Parameters: domain_
↪dimensions          = [128 128 128]
P001 yt : [INFO     ] 2017-05-23 16:51:33,974 Parameters: domain_left_
↪edge          = [ 0.  0.  0.]
P001 yt : [INFO     ] 2017-05-23 16:51:33,974 Parameters: domain_right_
↪edge          = [ 6.28318531  6.28318531  6.28318531]
P000 yt : [INFO     ] 2017-05-23 16:51:34,589 Rendering scene (Can take a
↪while).
P000 yt : [INFO     ] 2017-05-23 16:51:34,590 Creating volume
P003 yt : [INFO     ] 2017-05-23 16:51:34,592 Rendering scene (Can take a
↪while).
P002 yt : [INFO     ] 2017-05-23 16:51:34,592 Rendering scene (Can take a
↪while).
P003 yt : [INFO     ] 2017-05-23 16:51:34,593 Creating volume
P002 yt : [INFO     ] 2017-05-23 16:51:34,593 Creating volume
P001 yt : [INFO     ] 2017-05-23 16:51:34,606 Rendering scene (Can take a
↪while).
P001 yt : [INFO     ] 2017-05-23 16:51:34,607 Creating volume
```

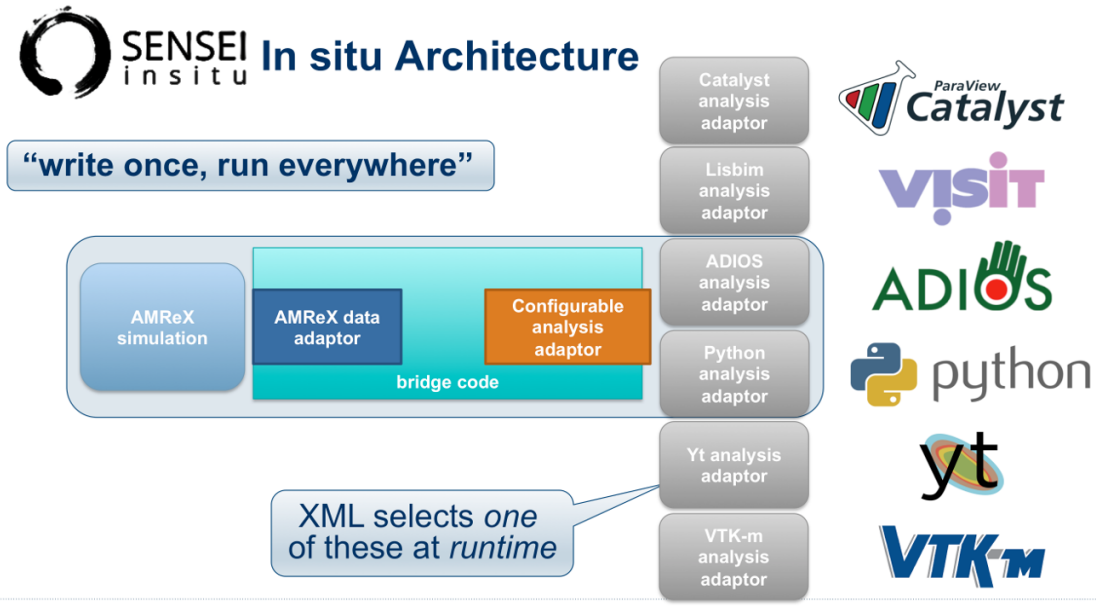
Because the `parallel_objects()` function transforms the loop into a data-parallel problem, this procedure strong scales nearly perfectly to an arbitrarily large number of MPI processes, allowing for rapid rendering of large time series of data.

6.5 SENSEI

SENSEI is a light weight framework for in situ data analysis. SENSEI's data model and API provide uniform access to and run time selection of a diverse set of visualization and analysis back ends including VisIt Libsim, ParaView Catalyst, VTK-m, Ascent, ADIOS, Yt, and Python.

6.5.1 System Architecture

The three major architectural components in SENSEI are *data adaptors* which present simulation data in SENSEI's data model, *analysis adaptors* which present the back end data consumers to the simulation, and *bridge code* from which the simulation manages adaptors and periodically pushes data through the system. SENSEI comes equipped with a number of analysis adaptors enabling use of popular analysis and visualization libraries such as VisIt Libsim, ParaView Catalyst, Python, and ADIOS to name a few. AMReX contains SENSEI data adaptors and bridge code making it easy to use in AMReX based simulation codes.



6.6: SENSEI's in situ architecture enables use of a diverse of back ends which can be selected at run time via an XML configuration file

SENSEI provides a *configurable analysis adaptor* which uses an XML file to select and configure one or more back ends at run time. Run time selection of the back end via XML means one user can access Catalyst, another Libsim, yet another Python with no changes to the code. This is depicted in figure 6.6. On the left side of the figure AMReX produces data, the bridge code pushes the data through the configurable analysis adaptor to the back end that was selected at run time.

6.5.2 AMReX Integration

AMReX codes based on `amrex::Amr` can use SENSEI simply by enabling it in the build and run via ParmParse parameters. AMReX codes based on `amrex::AmrMesh` need to additionally invoke the bridge code in `amrex::AmrMeshInSituBridge`.

6.5.3 Compiling with GNU Make

For codes making use of AMReX's build system add the following variable to the code's main `GNUmakefile`.

```
USE_SENSEI_INSITU = TRUE
```

When set, AMReX's make files will query environment variables for the lists of compiler and linker flags, include directories, and link libraries. These lists can be quite elaborate when using more sophisticated back ends, and are best set automatically using the `sensei_config` command line tool that should be installed with SENSEI. Prior to invoking make use the following command to set these variables:

```
source sensei_config
```

Typically, the `sensei_config` tool is in the users PATH after loading the desired SENSEI module. After configuring the build environment with `sensei_config`, proceed as usual.

```
make -j4 -f GNUmakefile
```

6.5.4 ParmParse Configuration

Once an AMReX code has been compiled with SENSEI features enabled, it will need to be enabled and configured at runtime. This is done using ParmParse input file. The following 3 ParmParse parameters are used:

```
sensei.enabled = 1
sensei.config = render_iso_catalyst_2d.xml
sensei.frequency = 2
```

`sensei.enabled` turns SENSEI on or off. `sensei.config` points to the SENSEI XML file which selects and configures the desired back end. `sensei.frequency` controls the number of level 0 time steps in between SENSEI processing.

6.5.5 Back-end Selection and Configuration

The back end is selected and configured at run time using the SENSEI XML file. The XML sets parameters specific to SENSEI and to the chosen back end. Many of the back ends have sophisticated configuration mechanisms which SENSEI makes use of. For example the following XML configuration was used on NERSC's Cori with IAMR to render 10 iso surfaces, shown in figure 6.7, using VisIt Libsim.

```
<sensei>
  <analysis type="libsim" frequency="1" mode="batch"
    visitdir="/usr/common/software/sensei/visit"
    session="rt_sensei_configs/visit_rt_contour_alpha_10.session"
    image-filename="rt_contour_%ts" image-width="1555" image-height="815"
    image-format="png" enabled="1"/>
</sensei>
```

The *session* attribute names a session file that contains VisIt specific runtime configuration. The session file is generated using VisIt GUI on a representative dataset. Usually this data set is generated in a low resolution run of the desired simulation.

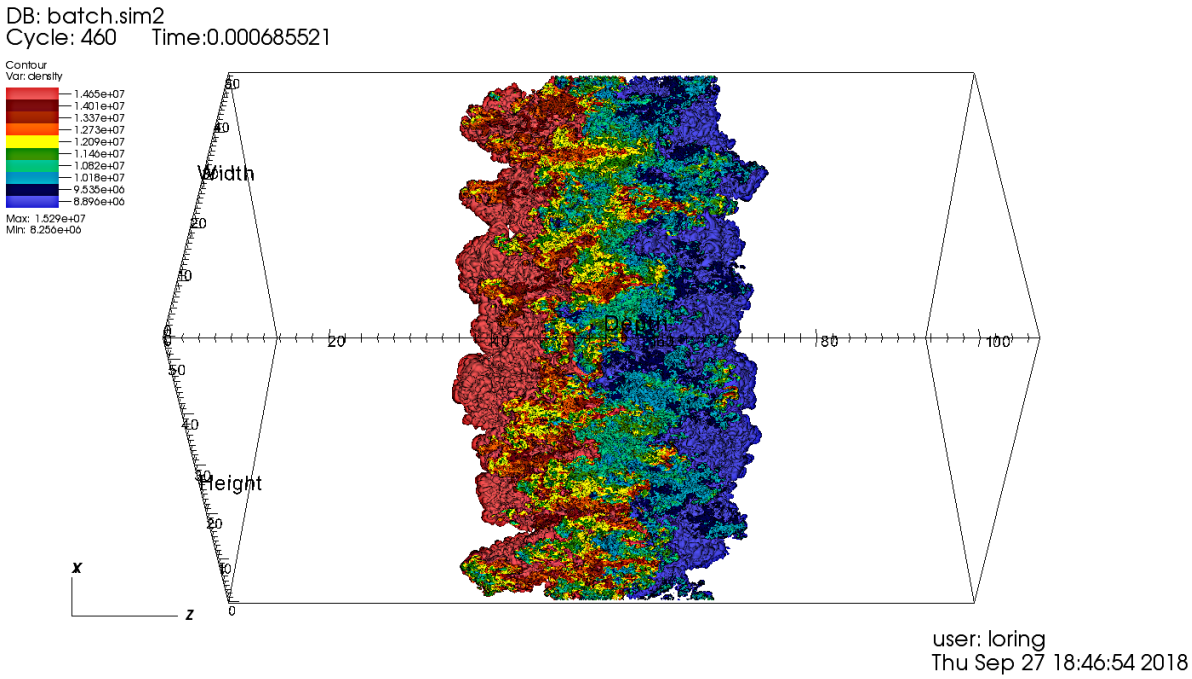
The same run and visualization was repeated using ParaView Catalyst, shown in figure 6.8, by providing the following XML configuration.

```
<sensei>
  <analysis type="catalyst" pipeline="pythonscript"
    filename="rt_sensei_configs/rt_contour.py" enabled="1" />
</sensei>
```

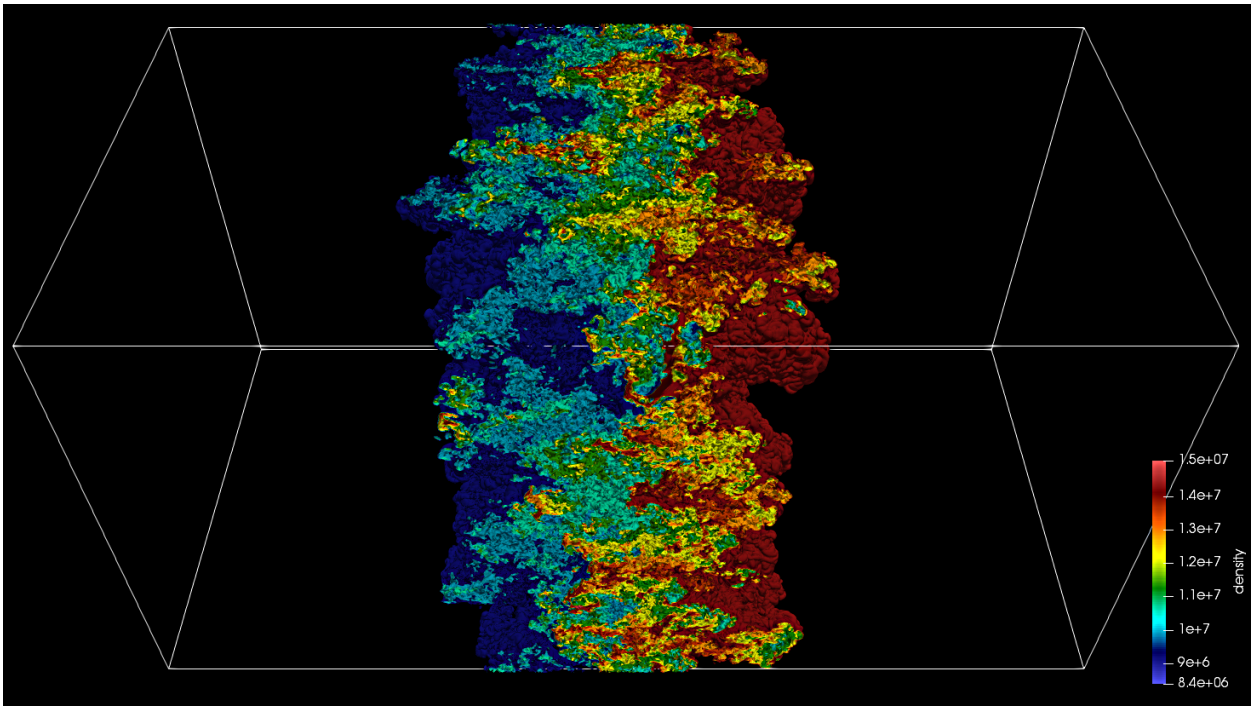
Here the *filename* attribute is used to pass Catalyst a Catalyst specific configuration that was generated using the ParaView GUI on a representative dataset.

6.5.6 Obtaining SENSEI

SENSEI is hosted on Kitware's Gitlab site at https://gitlab.kitware.com/sensei/sensei It's best to checkout the latest release rather than working on the master branch.



6.7: SENSEI-Libsim in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.



6.8: SENSEI-Catalyst in situ visualization of a Raleigh-Taylor instability computed by IAMR on NERSC Cori using 2048 cores.

To ease the burden of wrangling back end installs SENSEI provides two platforms with all dependencies pre-installed, a VirtualBox VM, and a NERSC Cori deployment. New users are encouraged to experiment with one of these.

6.5.7 SENSEI VM

The SENSEI VM comes with all of SENSEI's dependencies and the major back ends such as VisIt and ParaView installed. The VM is the easiest way to test things out. It also can be used to see how installs were done and the environment configured.

6.5.8 NERSC Cori

SENSEI is deployed at NERSC on Cori. The NERSC deployment includes the major back ends such as ParaView Catalyst, VisIt Libsim, and Python.

AmrLevel Tutorial with Catalyst

The following steps show how to run the tutorial with ParaView Catalyst. The simulation will periodically write images during the run.

```
ssh cori.nersc.gov
cd $SCRATCH
git clone https://github.com/AMReX-Codes/amrex.git
cd amrex/Tutorials/Amr/Advection_AmrLevel/Exec/SingleVortex
module use /usr/common/software/sensei/modulefiles
module load sensei/2.1.0-catalyst-shared
source sensei_config
vim GNUmakefile
# USE_SENSEI_INSITU=TRUE
make -j4 -f GNUmakefile
vim inputs
# sensei.enabled=1
# sensei.config=sensei/render_iso_catalyst_2d.xml
salloc -C haswell -N 1 -t 00:30:00 -q debug
cd $SCRATCH/amrex/Tutorials/Amr/Advection_AmrLevel/Exec/SingleVortex
./main2d.gnu.haswell.MPI.ex inputs
```

AmrLevel Tutorial with Libsim

The following steps show how to run the tutorial with VisIt Libsim. The simulation will periodically write images during the run.

```
ssh cori.nersc.gov
cd $SCRATCH
git clone https://github.com/AMReX-Codes/amrex.git
cd amrex/Tutorials/Amr/Advection_AmrLevel/Exec/SingleVortex
module use /usr/common/software/sensei/modulefiles
module load sensei/2.1.0-libsim-shared
source sensei_config
vim GNUmakefile
# USE_SENSEI_INSITU=TRUE
make -j4 -f GNUmakefile
vim inputs
```

(continues on next page)

(continued from previous page)

```
# sensei.enabled=1
# sensei.config=sensei/render_iso_libsim_2d.xml
salloc -C haswell -N 1 -t 00:30:00 -q debug
./main2d.gnu.haswell.MPI.ex inputs
```

CHAPTER 7

Tutorials

This sections includes several self-contained tutorials.

7.1 Tutorial - Non-reacting flow past a cylinder

7.1.1 Introduction

PeleLM enables the representation of complex non-Cartesian geometries using an embedded boundary (EB) method. This method relies on intersecting an arbitrary surface with the Cartesian matrix of uniform cells, and modifies the numerical stencils near cells that are cut by the EB.

The goal of this tutorial is to setup a simple 2-dimentional flow past cylinder case in *PeleLM*. This document provides step by step instructions to properly set-up the domain and boundary conditions, construct an initial solution.

7.1.2 Setting-up your environment

PeleProduction

As explained in section *PeleLM Quickstart*, *PeleLM* relies on a number of supporting softwares:

- *AMReX* is a software frameworks that provides the data structure and enable massive parallelization.
- *IAMR* is a parallel, adaptive mesh refinement (AMR) code that solves the variable-density incompressible Navier-Stokes equations.
- *PelePhysics* is a repository of physics databases and implementation code. In particular, the choice of chemistry and transport models as well as associated functions and capabilities are managed in *PelePhysics*.

All of these codes have their own development cycle, and it can make the setup of a *PeleLM* run a bit tricky. To simplify the process, PeleProduction will be employed. *PeleProduction* is a collection of run folders for various *Pele* codes and processing. It includes git submodules for the dependent codes (such as *PeleLM*, *PelePhysics*, *AMReX*, etc), that can be frozen to a particular commit. This organizational strategy enables to manage the interactions between the various dependent repositories (to keep them all compatible with each other).

Step by step instructions

First, make sure that "git" is installed on your machine—we recommend version 1.7.x or higher. Then, follow these few steps to setup your run environment:

1. Download the *PeleProduction* repository and :

```
git clone https://github.com/AMReX-Combustion/PeleProduction.git

cd PeleProduction
```

2. Switch to the TripleFlame branch :

```
git checkout Tutorials
```

3. The first time you do this, you will need to tell git that there are submodules. Git will look at the `.gitmodules` file in this branch and use that :

```
git submodule init
```

4. Finally, get the correct commits of the sub-repos set up for this branch:

```
git submodule update
```

You are now ready to build the `FlowPastCylinder` case associated with this branch. To do so:

```
cd Tutorials/FlowPastCylinder
```

And follow the next steps !

7.1.3 Numerical setup

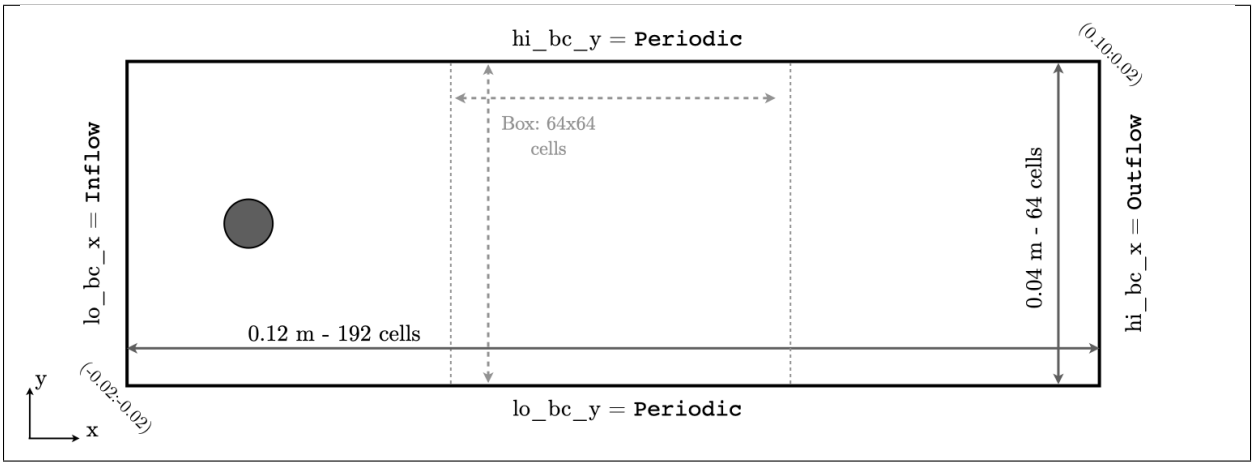
In this section we review the content of the various input files for the flow past cylinder test case. To get additional information about the keywords discussed, the user is referred to section *PeleLM control*.

Test case and boundary conditions

Direct Numerical Simulations (DNS) is performed on a 12x4 cm^2 2D computational domain, with the bottom left corner located at (-0.02:-0.02) and the top right corner at (0.10:0.02). The base grid is decomposed into 192x64 cells and up to 3 levels of refinement (although we will start with a single level). The refinement ratio between each level is set to 2. The maximum box size is fixed at 64, and the base (level 0) grid is composed of 3 boxes, as shown in Fig 7.1.

Periodic boundary conditions are used in the transverse (y) direction, while `Inflow` (dirichlet) and `Outflow` (neumann) boundary conditions are used in the main flow direction (x). The flow goes from left to right. A cylinder of radius 0.0035 m is placed in the middle of the flow at (0.0:0.0).

7.1: Sketch of the computational domain with level 0 box decomposition.



The geometry of the problem is specified in the first block of the `inputs.2d-regt_VS`:

```
#----------------------DOMAIN DEFINITION------------------------
geometry.is_periodic = 0 0             # Periodicity in each direction: 0 => no, 1 =>␣
↪yes
geometry.coord_sys   = 0               # 0 => cart, 1 => RZ
geometry.prob_lo     = -0.02 -0.02     # x_lo y_lo
geometry.prob_hi     =  0.10  0.02     # x_hi y_hi
```

The second block determines the boundary conditions. Note that *Interior* is used to indicate periodic boundary conditions. Refer to Fig 7.1:

```
# >>>>>>>>>>>>>  BC FLAGS <<<<<<<<<<<<<<<<
# Interior, Inflow, Outflow, Symmetry,
# SlipWallAdiab, NoSlipWallAdiab, SlipWallIsotherm, NoSlipWallIsotherm
peleLM.lo_bc = Inflow   SlipWallAdiab
peleLM.hi_bc = Outflow  SlipWallAdiab
```

In the present case, the EB geometry is a simple cylinder (or sphere) which is readily available from the *AMReX* library and only a few paremeters need to be specified by the user. This is done further down in the input file:

```
#------------  INPUTS FOR EMBEDED BOUNDARIES -----------------
eb2.geom_type                    = sphere
eb2.sphere_radius                = 0.0035
eb2.sphere_center                = 0.00 0.00
eb2.sphere_has_fluid_inside      = 0
eb2.small_volfrac                = 1.0e-4
```

Note that the last parameter is used to specify a volume fraction (ratio of the uncovered surface (2D) or volume (3D) over the cell surface or volume) threshold below which a cell is considered fully covered. This prevents the appearance of extremely small partially covered cells which are numerically unstable.

The number of levels, refinement ratio between levels, maximium grid size as well as other related refinement parameters are set under the third block :

```
#-------------------------AMR CONTROL----------------------------
amr.n_cell          = 192 64     # Level 0 number of cells in each direction
amr.v               = 1          # amr verbosity level
amr.max_level       = 0          # maximum level number allowed
```

(continues on next page)

(continued from previous page)

```
amr.ref_ratio       = 2 2 2 2    # refinement ratio
amr.regrid_int      = 2          # how often to regrid
amr.n_error_buf     = 2 2 2 2    # number of buffer cells in error est
amr.grid_eff        = 0.7        # what constitutes an efficient grid
amr.blocking_factor = 16         # block factor in grid generation
amr.max_grid_size   = 64         # maximum box size
```

Problem specifications

This very simple problem only has three user-defined problem parameters: the inflow velocity magnitude, the pressure and the temperature. This setup is also constructed to be able to perform the simulation of mixture perturbation crossing over the cylinder so that a switch is available to run this case rather than the simple vortex shedding past a cylinder. Specifying dirichlet `Inflow` conditions in *PeleLM* can seem daunting at first. But it is actually a very flexible process. We walk the user through the details which involve the following files:

- `pelelm_prob_parm.H`, assemble in a C++ struct `ProbParm` the input variables as well as other variables used in the initialization process.
- `pelelm_prob.cpp`, initialize and provide default values to the entries of `ProbParm` and allow the user to pass run-time value using the *AMReX* parser (`ParmParse`). In the present case, the parser will read the parameters in the `PROBLEM PARAMETERS` block:

```
prob.type         = VortexShedding
prob.meanFlowMag  = 3.0
```

- finally, `pelelm_prob.H` contains the `pelelm_initdata` and `bcnormal` functions responsible for generating the initial and boundary conditions, respectively.

Note that in the present case, the default values of pressure and temperature are employed since their respective keywords are not specified in the input file.

Finally, this test uses a constant set of transport parameters rather relying on the EGLib library (see sec:model:EqSets for more details on EGLib). These transport parameters are specified in the `CONSTANT TRANSPORT` block:

```
#------------  INPUTS TO CONSTANT TRANSPORT -----------------
transport.const_viscosity        = 2.0e-04
transport.const_bulk_viscosity   = 0.0
transport.const_conductivity     = 0.0
transport.const_diffusivity      = 0.0
```

Only the viscosity in the present case, and note that CGS units are employed while specifying these properties. Using these parameters, it is possible to evaluate the Reynolds number, based on the inflow velocity and the cylinder diameter:

$$Re = \frac{\rho U_{inf} D}{\mu} = \frac{1.175 * 3 * 0.007}{2.0e-05} \sim 1200$$

This relatively high value ensures that the flow will exhibit vortex shedding.

Initial solution

An initial field of the main variables is always required to start a simulation. In the present case, the computational domain is filled with air in the condition of pressure and temperature provided by the user (or the default ones). An initial constant velocity of `meanFlowMag` is used, but note that *PeleLM* performs an initial velocity projection to enforce the low Mach number constraint which overwrite this initial velocity.

This initial solution is constructed via the routine `pelelm_initdata()`, in the file `pelelm_prob.H`. Additional information is provided as comments in this file for the eager reader, but nothing is required from the user at this point.

Numerical scheme

The `NUMERICS CONTROL` block can be modified by the user to increase the number of SDC iterations. Note that there are many other parameters controlling the numerical algorithm that the advanced user can tweak, but we will not talk about them in the present Tutorial. The interested user can refer to section *PeleLM algorithm controls*.

7.1.4 Building the executable

The last necessary step before starting the simulation consists of building the PeleLM executable. AMReX applications use a makefile system to ensure that all the required source code from the dependent libraries be properly compiled and linked. The `GNUmakefile` provides some compile-time options regarding the simulation we want to perform. The first line can be modified to specify the absolute path to the *PeleProduction* directory while the next four lines specify the paths towards the source code of *PeleLM*, *AMReX*, *IAMR* and *PelePhysics* and should not be changed.

Next comes the build configuration block:

```
#
# Build configuration
#

# AMREX options
DIM             = 2
USE_EB          = TRUE

# Compiler / parrallel paradigms
COMP            = gnu
USE_MPI         = TRUE
USE_OMP         = FALSE
USE_CUDA        = FALSE
USE_HIP         = FALSE

# MISC options
DEBUG           = FALSE
PRECISION       = DOUBLE
VERBOSE         = FALSE
TINY_PROFILE    = FALSE
```

It allows the user to specify the number of spatial dimensions (2D), trigger the compilation of the EB source code, the compiler (`gnu`) and the parallelism paradigm (in the present case only MPI is used). The other options can be activated for debugging and profiling purposes.

In *PeleLM*, the chemistry model (set of species, their thermodynamic and transport properties as well as the description of their of chemical interactions) is specified at compile time. Chemistry models available in *PelePhysics* can used in *PeleLM* by specifying the name of the folder in *PelePhysics/Support/Fuego/Mechanisms/Models* containing the relevant files, for example:

```
Chemistry_Model = air
```

Here, the model `air`, only contains 2 species (O2 and N2). The user is referred to the PelePhysics documentation for a list of available mechanisms and more information regarding the EOS, chemistry and transport models specified:

```
Eos_dir       := Fuego
Reactions_dir := Null
Transport_dir := Constant
```

You are now ready to build your first *PeleLM* executable !! Type in:

```
make -j4
```

The option here tells *make* to use up to 4 processors to create the executable (internally, *make* follows a dependency graph to ensure any required ordering in the build is satisfied). This step should generate the following file (providing that the build configuration you used matches the one above):

```
PeleLM2d.gnu.MPI.ex
```

You're good to go!

7.1.5 Running the problem on a coarse grid

As a first step towards obtaining the classical Von-Karman alleys, we will now let the flow establish using only the coarse base grid. The simulation will last for 50 ms.

Time-stepping parameters in `input.2d-regt` are specified in the `TIME STEPING CONTROL` block:

```
#----------------------TIME STEPING CONTROL----------------------
max_step       = 300000          # Maximum number of time steps
stop_time      = 0.05            # final physical time
ns.cfl         = 0.3             # cfl number for hyperbolic system
ns.init_shrink = 1.0             # scale back initial timestep
ns.change_max  = 1.1             # max timestep size increase
ns.dt_cutoff   = 5.e-10          # level 0 timestep below which we halt
```

The final simulation time is set to 0.05 s. *PeleLM* solves for the advection, diffusion and reaction processes in time, but only the advection term is treated explicitly and thus it constrains the maximum time step size dt_{CFL}. This constraint is formulated with a classical Courant-Friedrich-Levy (CFL) number, specified via the keyword `ns.cfl`. Additionally, as it is the case here, the initial solution is often made-up by the user and local mixture composition and temperature can result in the introduction of unreasonably fast chemical scales. To ease the numerical integration of this initial transient, the parameter `ns.init_shrink` allows to shrink the inital *dt* (evaluated from the CFL constraint) by a factor (usually smaller than 1), and let it relax towards dt_{CFL} as the simulation proceeds. Since the present case does not involve complex chemiscal processes, this parameter is kept to 1.0.

Input/output from *PeleLM* are specified in the `IO CONTROL` block:

```
#-------------------------IO CONTROL----------------------------
amr.checkpoint_files_output = 1   # Dump check file ? 0: no, 1: yes
amr.check_file      = chk_        # root name of checkpoint file
amr.check_per       = 0.05        # frequency of checkpoints
amr.plot_file       = plt_        # root name of plotfiles
amr.plot_per        = 0.005       # frequency of plotfiles
amr.derive_plot_vars=rhoRT mag_vort avg_pressure gradpx gradpy
amr.grid_log        = grdlog      # name of grid logging file
```

Information pertaining to the checkpoint and plot_file files name and output frequency can be specified there. We have specified here that a checkpoint file will be generated every 50 ms and a plotfile every 5 ms. *PeleLM* will always generate an initial plotfile `plt_00000` if the initialization is properly completed, and a final plotfile at the end of the

simulation. It is possible to request including *derived variables* in the plotfiles by appending their names to the `amr.derive_plot_vars` keyword. These variables are derived from the *state variables* (velocity, density, temperature, ρY_k, ρh) which are automatically included in the plotfile.

You finally have all the information necessary to run the first of several steps. Type in:

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

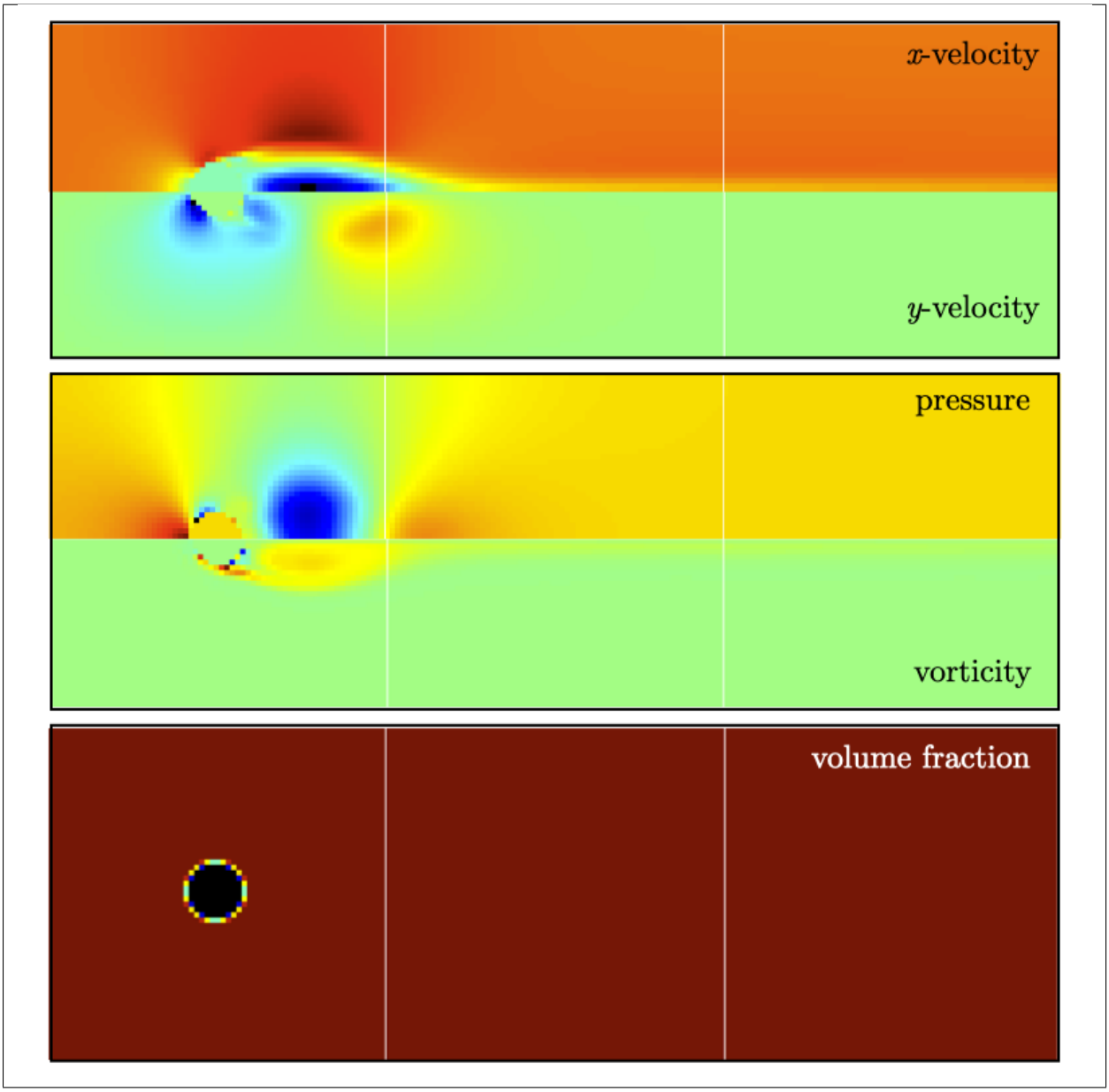
A lot of information is printed directly on the screen during a *PeleLM* simulation, but it will not be detailed in the present tutorial. If you wish to store these information for later analysis, you can instead use:

```
./PeleLM2d.gnu.MPI.ex inputs.2d-regt_VS > logCoarseRun.dat &
```

Whether you have used one or the other command, the computation finishes within a couple of minutes and you should obtain a set of `plt_****` files (and maybe a set appended with .old********** if you used both commands). Use Amrvis to vizualize the results. Use the following command to open the entire set of solutions:

```
amrvis -a plt_?????
```

7.2: Contour plots of velocity components, vorticity, pressure and volume fraction at t = 50 ms on the coarse grid.



At this point, you have established a flow with a large recirculation zone in the wake of the cylinder, but the flow has not yet fully transitioned to periodic vortex shedding. The flow is depicted in Fig 7.2 showing a few of the available contour plots at 50 ms. Note that the value of the fully covered cells is set to zero.

As can be seen from Fig 7.2, the flow has not yet transitioned to the familiar Von-Karman alleys and two aspects of the current simulation can delay or even prevent the onset of vortex shedding:

- the flow is initially symmetric and the transition to the familiar periodic flow is due to the growth of infinitesimal perturbations in the shear layer of the wake. Because the flow is artificially too symmetric, this transition can be delayed until round-off errors sufficiently accumulate.
- the numerical dissipation introduced by this coarse grid results in an effective Reynolds number probably significantly lower than the value estimated above.

Before adding refinement levels, we will first pursue the simulation until the flow reaches a periodic vortex shedding state. To do so, restart the simulation from the checkpoint file generated at the end of the first run and set the final simulation time to 200 ms:

```
#-------------------------IO CONTROL----------------------------
...
amr.restart             = chk_01327 # Restart from checkpoint ?

...

#----------------------TIME STEPING CONTROL----------------------
...
stop_time      = 0.20            # final physical time
```

and restart the simulation

```
./PeleLM2d.gnu.MPI.ex inputs.2d-regt_VS > logCoarseRun2.dat &
```

The flow has now fully transition and you can use Amrvis to visualize the serie of vortex in the wake of the cylinder. In the next step, we will add finer grid patches around the EB geometry and in high vorticity regions.

7.1.6 Refinement of the computation

We will now add a first level of refinement. In the present simulation, the refinement criteria could be based on several characteristics of the flow: velocity gradients, vorticity, pressure, ... In the following, we will simply use vorticity. Additionally, by construction the geometry must be built to the finest level which act as a refinement criteria based on the gradient of volume fraction. This is beneficial in this case in order to help refine the cylinder boundary layer. Start by increasing the number of AMR levels to one in the `AMR CONTROL` block:

```
amr.max_level       = 1           # maximum level number allowed
```

Then provide a definition of the new refinement critera in the `REFINEMENT CONTROL` block:

```
#-------------------REFINEMENT CONTROL-----------------------
# Refinement according to the vorticity, no field_name needed
amr.refinement_indicators     = lowvort highvort
amr.lowvort.max_level         = 1
amr.lowvort.value_less        = -1000
amr.lowvort.field_name        = mag_vort

amr.highvort.max_level         = 1
amr.highvort.value_greater     = 1000
amr.highvort.field_name        = mag_vort

# Refine the EB
ns.refine_cutcells            = 1
```

The first line simply declares a set of refinement indicators which are subsequently defined. For each indicator, the user can provide a limit up to which AMR level this indicator will be used to refine. Then there are multiple possibilities to specify the actual criterion: `value_greater`, `value_less`, `vorticity_greater` or `adjacent_difference_greater`. In each case, the user specify a threshold value and the name of variable on which it applies (except for the `vorticity_greater`). In the example above, the grid is refined up to level 1 at the location where the vorticity magnitude is above 1000 s^{-1} as well as on the cut cells (where the cylinder intersect with the edges of cell). Note that in the present case, the `vorticity_greater` was not used to ensure that regions of both low and high vorticity are refined.

With these new parameters, change the *checkpoint* file from which to restart and allow regridding upon restart by updating the following lines in the `IO CONTROL` block:

```
amr.restart              = chk_06195 # Restart from checkpoint ?
amr.regrid_on_restart    = 1
```

, increase the *stop_time* to 300 ms in the `TIME STEPING CONTROL` block:

```
stop_time      = 0.30            # final physical time
```

and start the simulation again (using multiple processor if available)

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt_VS > log2Levels.dat &
```

Once again, use Amrvis to visualize the effects of refinement: after an initial transient, the flow return to a smooth periodic vortex shedding and the boundary layer of the cylinder is now significantly better captured but still far from fully refined. As a final step, we will add another level of refinement, only at the vicinity of the cylinder since the level 1 resolution appears sufficient to discretize the vortices in the wake. To do so, simply allow for another level of refinement:

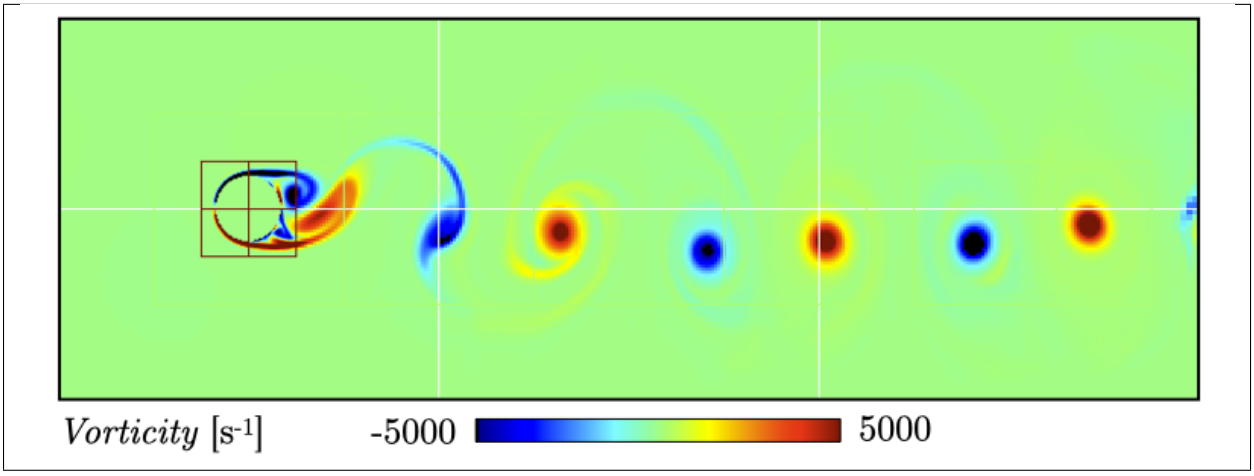
```
amr.max_level        = 2          # maximum level number allowed
```

and since the vorticity refinement criterion only refine up to level 1, the level 2 will only be located around the EB. Update the *checkpoint* file in the `IO CONTROL` block, increase the *stop_time* to 350 ms in the the `TIME STEPING CONTROL` and restart the simulation:

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt_VS > log3Levels.dat &
```

You should obtain a flow with a vorticity field similar to Fig. 7.3. For the purpose of the present tutorial, this will be our final solution but one can see that the flow has not yet return to a periodic vortex shedding and additinal resolution could be added locally to obtain smoother flow features.

7.3: Contour plots of vorticit at t = 350 ms with 2 levels of refinements.

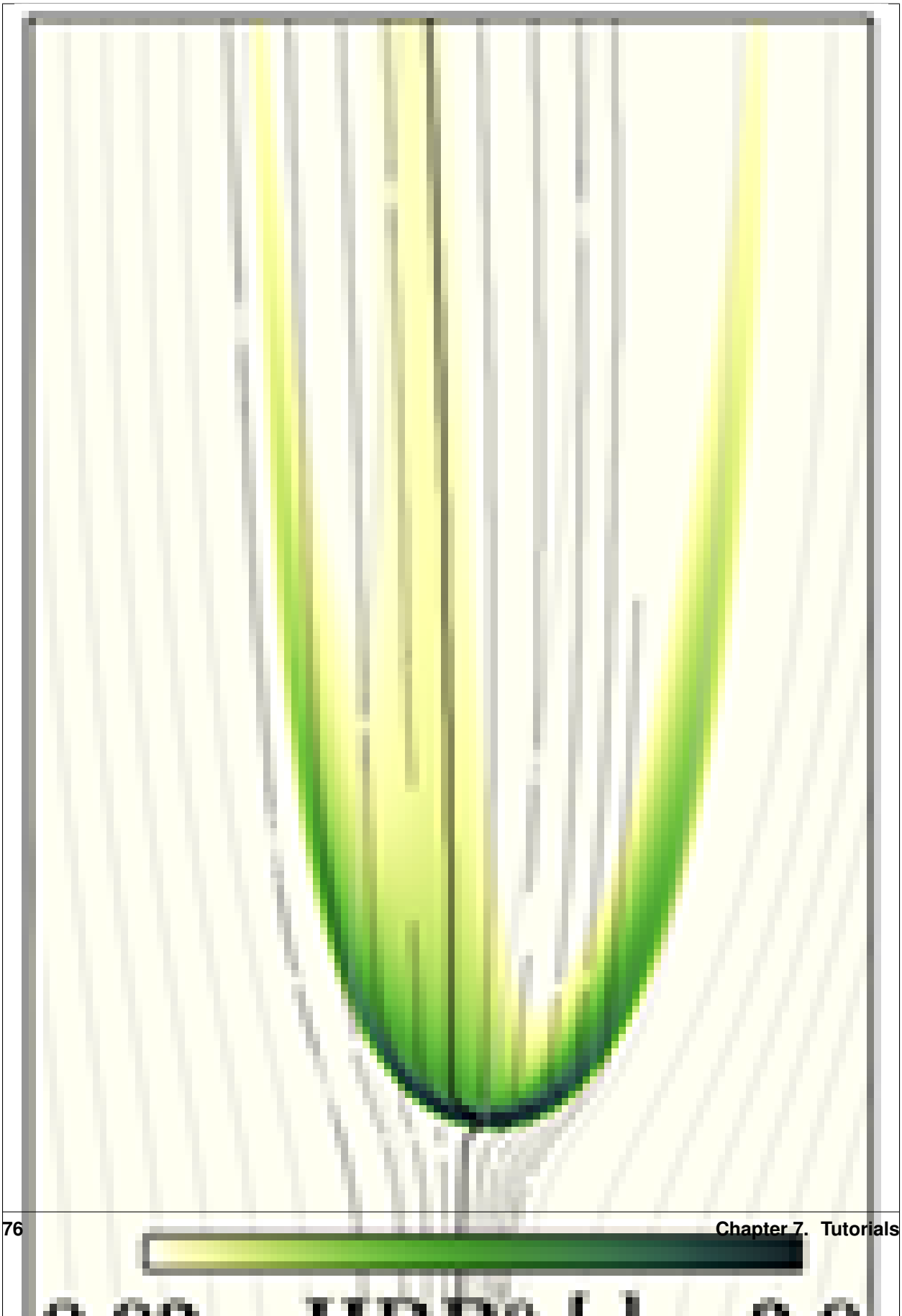


7.2 Tutorial - A simple triple flame

7.2.1 Introduction

Laminar flames have the potential to reveal the fundamental structure of combustion without the added complexities of turbulence. They also aid in our understanding of the more complex turbulent flames. Depending on the fuel involved and the flow configuration, the laminar flames can take on a number of interesting geometries. For example, as practical combustion systems often operate in partially premixed mode, with one or more fuel injections, a wide range of fresh gas compositions can be observed; and these conditions favor the appearance of edge flames, see Fig. 7.4.

7.4: Normalized heat release rate (top) and temperature (bottom) contours of two-dimensional (2D) laminar lifted flames of ethylene.



Edge flames are composed of lean and rich premixed flame wings usually surrounding a central anchoring diffusion flame extending from a single point [PCI2007]. Edge flames play an important role in flame stabilization, re-ignition and propagation. Simple fuels can exhibit up to three burning branches while diesel fuel, with a low temperature combustion mode, can exhibit up to 5 branches.

The goal of this tutorial is to setup a simple 2D laminar triple edge flame configuration with *PeleLM*. This document provides step by step instructions to properly set-up the domain and boundary conditions, construct an initial solution, and provides guidance on how to monitor and influence the initial transient to reach a final steady-state solution. In a final Section, post-processing tools available in *PeleAnalysis* are used to extract information about the triple flame.

7.2.2 Setting-up your environment

PeleProduction

As explained in section *PeleLM Quickstart*, *PeleLM* relies on a number of supporting softwares:

- *AMReX* is a software frameworks that provides the data structure and enable massive parallelization.
- *IAMR* is a parallel, adaptive mesh refinement (AMR) code that solves the variable-density incompressible Navier-Stokes equations.
- *PelePhysics* is a repository of physics databases and implementation code. In particular, the choice of chemistry and transport models as well as associated functions and capabilities are managed in *PelePhysics*.

All of these codes have their own development cycle, and it can make the setup of a *PeleLM* run a bit tricky. To simplify the process, PeleProduction will be employed. *PeleProduction* is a collection of run folders for various *Pele* codes and processing. It includes git submodules for the dependent codes (such as *PeleLM*, *PelePhysics*, *AMReX*, etc), that can be frozen to a particular commit. This organizational strategy enables to manage the interactions between the various dependent repositories (to keep them all compatible with each other).

Step by step instructions

First, make sure that "git" is installed on your machine—we recommend version 1.7.x or higher. Then, follow these few steps to setup your run environment:

1. Download the *PeleProduction* repository and :

```
git clone https://github.com/AMReX-Combustion/PeleProduction.git

cd PeleProduction
```

2. Switch to the TripleFlame branch :

```
git checkout -b Tutorials remotes/origin/Tutorials
```

3. The first time you do this, you will need to tell git that there are submodules. Git will look at the `.gitmodules` file in this branch and use that :

```
git submodule init
```

4. Finally, get the correct commits of the sub-repos set up for this branch:

```
git submodule update
```

You are now ready to build the `TripleFlame` case associated with this branch. To do so:

```
cd Tutorials/TripleFlame
```

And follow the next steps !

7.2.3 Numerical setup

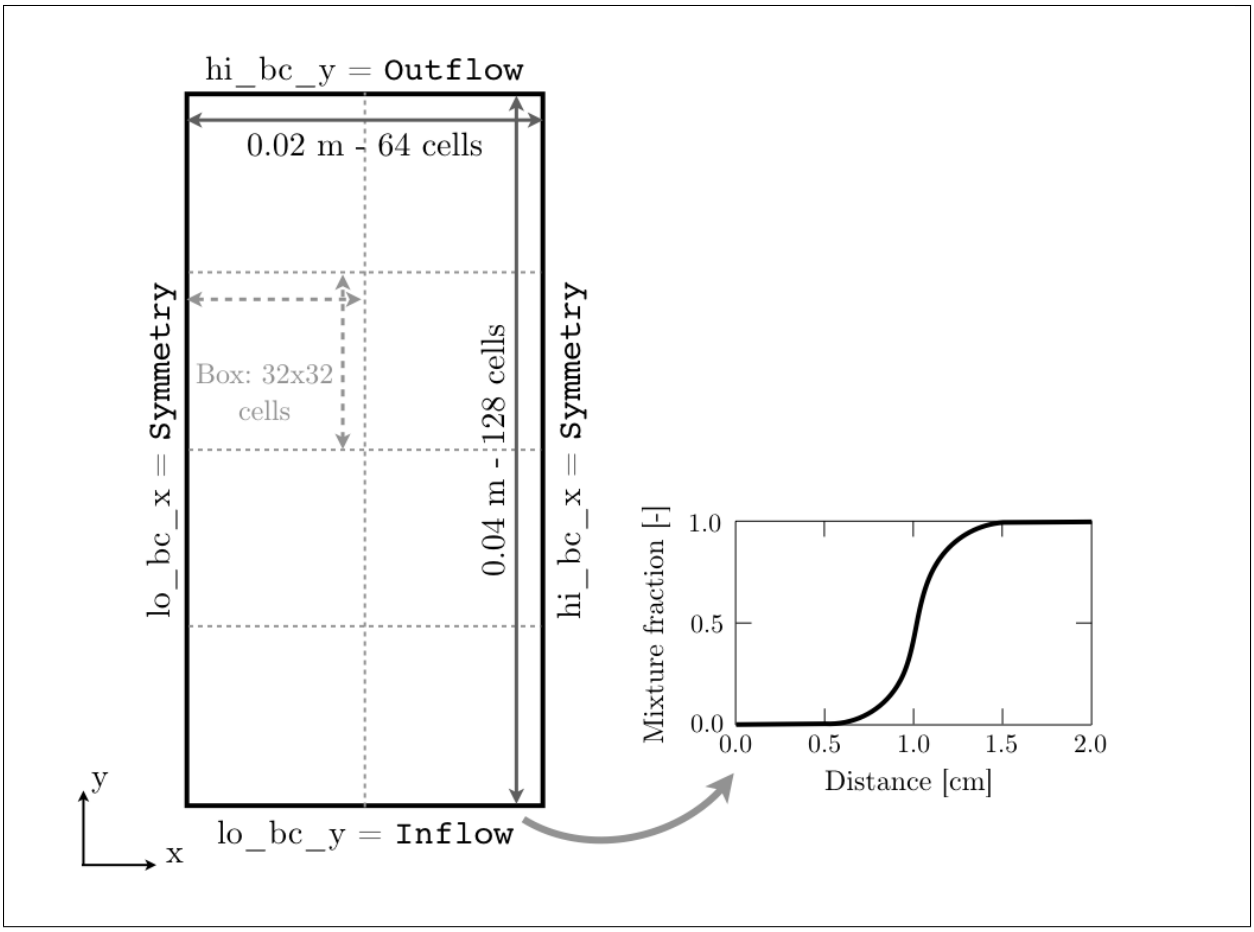
In this section we review the content of the various input files for the Triple Flame test case. To get additional information about the keywords discussed, the user is referred to section *PeleLM control*.

Test case and boundary conditions

Direct Numerical Simulations (DNS) are performed on a 2x4 cm^2 2D computational domain using a 64x128 base grid and up to 4 levels of refinement (although we will start with a lower number of levels). The refinement ratio between each level is set to 2. With 4 levels, this means that the minimum grid size inside the reaction layer will be just below 20 m. The maximum box size is fixed at 32, and the base (level 0) grid is composed of 8 boxes, as shown in Fig 7.5.

Symmetric boundary conditions are used in the transverse (x) direction, while `Inflow` (dirichlet) and `Outflow` (neumann) boundary conditions are used in the main flow direction (y). The flow goes from the bottom to the top of the domain. The specificities of the `Inflow` boundary condition are explained in subsection *Inflow specification*

7.5: Sketch of the computational domain with level 0 box decomposition (left) and input mixture fraction profile (right).



The geometry of the problem is specified in the first block of the `inputs.2d-regt`:

```
#----------------------DOMAIN DEFINITION------------------------
geometry.is_periodic = 0 0       # Periodicity in each direction: 0 => no, 1 => yes
geometry.coord_sys   = 0         # 0 => cart, 1 => RZ
geometry.prob_lo     = 0. 0.     # x_lo y_lo
geometry.prob_hi     = 0.02 0.04 # x_hi y_hi
```

The second block determines the boundary conditions. Refer to Fig 7.5:

```
# >>>>>>>>>>>>>  BC FLAGS <<<<<<<<<<<<<<<<
# Interior, Inflow, Outflow, Symmetry,
# SlipWallAdiab, NoSlipWallAdiab, SlipWallIsotherm, NoSlipWallIsotherm
peleLM.lo_bc = Symmetry  Inflow
peleLM.hi_bc = Symmetry  Outflow
```

The number of levels, refinement ratio between levels, maximium grid size as well as other related refinement parameters are set under the third block :

```
#-------------------------AMR CONTROL----------------------------
amr.n_cell          = 64 128     # Level 0 number of cells in each direction
amr.v               = 1          # amr verbosity level
amr.max_level       = 1          # maximum level number allowed
amr.ref_ratio       = 2 2 2 2    # refinement ratio
amr.regrid_int      = 2          # how often to regrid
amr.n_error_buf     = 1 1 1 2    # number of buffer cells in error est
amr.grid_eff        = 0.9        # what constitutes an efficient grid
amr.grid_eff        = 0.7        # what constitutes an efficient grid
amr.blocking_factor = 16         # block factor in grid generation
amr.max_grid_size   = 32         # maximum box size
```

Inflow specification

The edge flame is stabilized against an incoming mixing layer with a uniform velocity profile. The mixing layer is prescribed using an hyperbolic tangent of mixture fraction z between 0 and 1, as can be seen in Fig 7.5:

$$z(x) = 0.5\Big(1 + tanh\Big(\frac{x - 0.6(x_{hi} + x_{lo})}{0.05(x_{hi} - x_{lo})}\Big)\Big)$$

where z is based on the classical elemental composition [CF1990]:

$$z = \frac{\beta - \beta_{ox}}{\beta_{fu} - \beta_{ox}}$$

where β is Bilger's coupling function, and subscript ox and fu correspond to oxidizer and fuel streams respectively.

Specifying dirichlet `Inflow` conditions in *PeleLM* can seem daunting at first. But it is actually a very flexible process. We walk the user through the details of it for the Triple Flame case just described. The files involved are:

- `pelelm_prob_parm.H`, assemble in a C++ namespace `ProbParm` the input variables as well as other variables used in the initialization process.
- `pelelm_prob.cpp`, initialize and provide default values to the entries of `ProbParm` and allow the user to pass run-time value using the *AMReX* parser (`ParmParse`). In the present case, the parser will read the parameters in the `PROBLEM PARAMETERS` block:

```
prob.P_mean = 101325.0
prob.T_in = 300.0
prob.V_in = 0.85
prob.Zst = 0.055
```

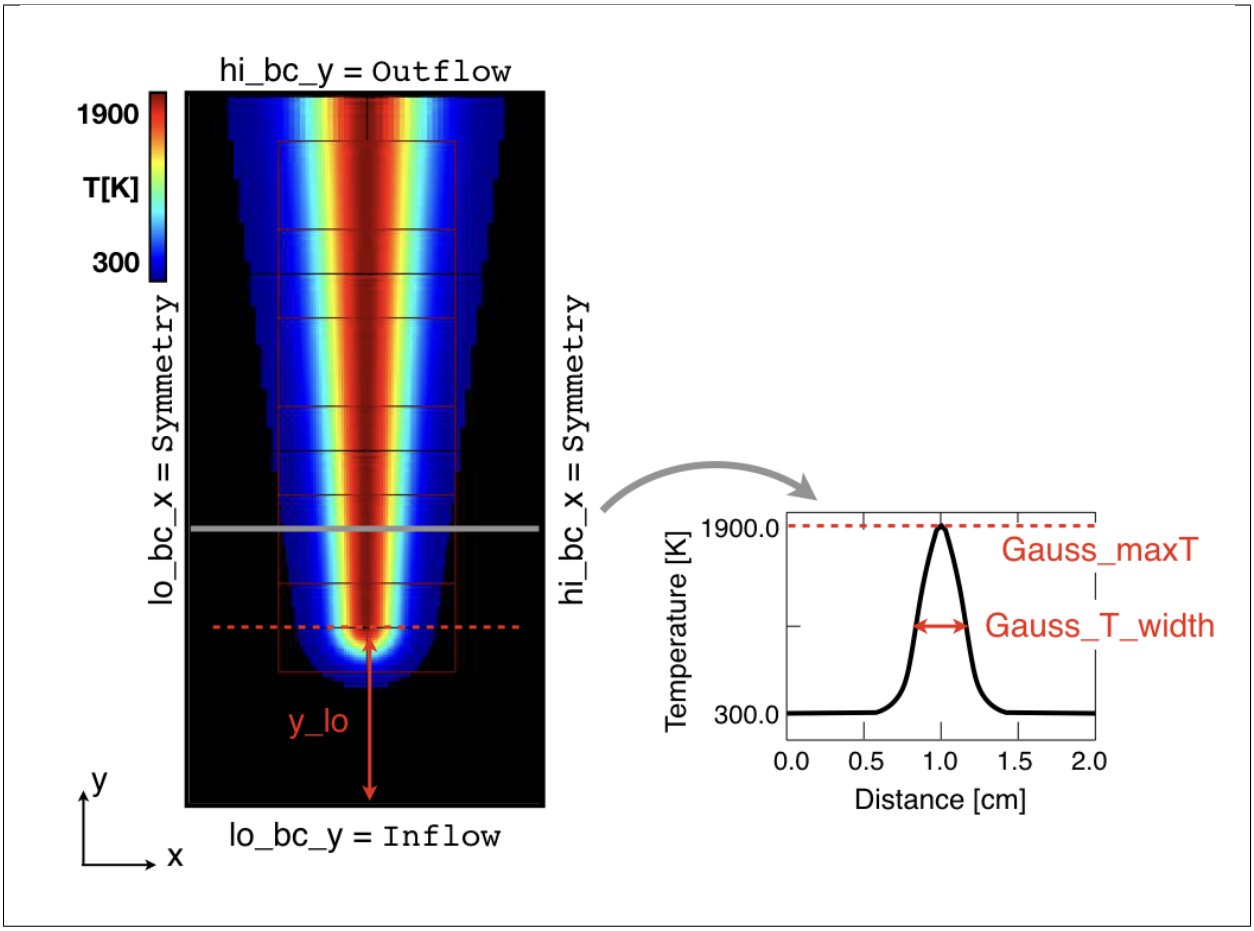
- finally, `pelelm_prob.H` contains the `pelelm_initdata` and `bcnormal` functions responsible for generating the initial and boundary conditions, resspectively.

Note that in our specific case, we compute the input value of the mass fractions (Y) *directly* in `bcnormal`, using the `ProbParm` variables. We do not need any additional information, because we hard coded the hyperbolic tangent profile of z (see previous formula) and there is a direct relation with the mass fraction profiles. The interested reader can look at the function `set_Y_from_Ksi` and `set_Y_from_Phi` in `pelelm_prob.H`.

Initial solution

An initial field of the main variables is always required to start a simulation. Ideally, you want for this initial solution to approximate the final (steady-state in our case) solution as much as possible. This will speed up the initial transient and avoid many convergence issues. In the present tutorial, an initial solution is constructed by imposing the same inlet hyperbolic tangent of mixture fraction than described in subsection *Inflow specification* everywhere in the domain; and reconstructing the species mass fraction profiles from it. To ensure ignition of the mixture, a progressively widening Gaussian profile of temperature is added, starting from about 1 cm, and stretching until the outlet of the domain. The initial temperature field is shown in Fig 7.6, along with the parameters controlling the shape of the hot spot.

7.6: Initial temperature field (left) as well as widening gaussian 1D y-profiles (right) and associated parameters. The initial solution contains 2 levels.



This initial solution is constructed via the routine `pelelm_initdata()`, in the file `pelelm_prob.H`. Additional information is provided as comments in this file for the eager reader, but nothing is required from the user at this point.

Numerical scheme

The `NUMERICS CONTROL` block can be modified by the user to increase the number of SDC iterations. Note that there are many other parameters controlling the numerical algorithm that the advanced user can tweak, but we will not talk about them in the present Tutorial. The interested user can refer to section *PeleLM algorithm controls*.

7.2.4 Building the executable

The last necessary step before starting the simulation consists of building the PeleLM executable. AMReX applications use a makefile system to ensure that all the required source code from the dependent libraries be properly compiled and linked. The `GNUmakefile` provides some compile-time options regarding the simulation we want to perform. The first four lines of the file specify the paths towards the source code of *PeleLM*, *AMReX*, *IAMR* and *PelePhysics* and should not be changed.

Next comes the build configuration block:

```
#
# Build configuration
#
DIM             = 2
COMP            = gnu
DEBUG           = FALSE
USE_MPI         = TRUE
USE_OMP         = FALSE
USE_CUDA        = FALSE
PRECISION       = DOUBLE
VERBOSE         = FALSE
TINY_PROFILE    = FALSE
```

It allows the user to specify the number of spatial dimensions (2D), the compiler (`gnu`) and the parallelism paradigm (in the present case only MPI is used). The other options can be activated for debugging and profiling purposes.

In *PeleLM*, the chemistry model (set of species, their thermodynamic and transport properties as well as the description of their of chemical interactions) is specified at compile time. Chemistry models available in *PelePhysics* can used in *PeleLM* by specifying the name of the folder in *PelePhysics/Support/Fuego/Mechanisms/Models* containing the relevant files, for example:

```
Chemistry_Model = drm19
```

Here, the methane kinetic model `drm19`, containing 21 species is employed. The user is referred to the PelePhysics documentation for a list of available mechanisms and more information regarding the EOS, chemistry and transport models specified:

```
Eos_dir       := Fuego
Reactions_dir := Fuego
Transport_dir := Simple
```

Finally, *PeleLM* utilizes the chemical kinetic ODE integrator CVODE. This Third Party Librabry (TPL) is not shipped with the *PeleLM* distribution but can be readily installed through the makefile system of *PeleLM*. To do so, type in the following command:

```
make TPL
```

Note that the installation of *CVODE* requires CMake 3.12.1 or higher.

You are now ready to build your first *PeleLM* executable !! Type in:

```
make -j4
```

The option here tells *make* to use up to 4 processors to create the executable (internally, *make* follows a dependency graph to ensure any required ordering in the build is satisfied). This step should generate the following file (providing that the build configuration you used matches the one above):

```
PeleLM2d.gnu.MPI.ex
```

You're good to go !

7.2.5 Initial transient phase

First step: the initial solution

When performing time-dependent numerical simulations, it is good practice to verify the initial solution. To do so, we will run *PeleLM* for a single time step, to generate an initial plotfile `plt_00000`.

Time-stepping parameters in `input.2d-regt` are specified in the `TIME STEPING CONTROL` block:

```
#----------------------TIME STEPING CONTROL----------------------
max_step       = 1               # maximum number of time steps
stop_time      = 4.00            # final physical time
ns.cfl         = 0.1             # cfl number for hyperbolic system
ns.init_shrink = 0.01            # scale back initial timestep
ns.change_max  = 1.1             # max timestep size increase
ns.dt_cutoff   = 5.e-10          # level 0 timestep below which we halt
```

The maximum number of time steps is set to 1 for now, while the final simulation time is 4.0 s. Note that, when both `max_step` and `stop_time` are specified, the more stringent constraint will control the termination of the simulation. *PeleLM* solves for the advection, diffusion and reaction processes in time, but only the advection term is treated explicitly and thus it constrains the maximum time step size dt_{CFL}. This constraint is formulated with a classical Courant-Friedrich-Levy (CFL) number, specified via the keyword `ns.cfl`. Additionally, as it is the case here, the initial solution is often made-up by the user and local mixture composition and temperature can result in the introduction of unreasonably fast chemical scales. To ease the numerical integration of this initial transient, the parameter `ns.init_shrink` allows to shrink the inital *dt* (evaluated from the CFL constraint) by a factor (usually smaller than 1), and let it relax towards dt_{CFL} as the simulation proceeds.

Input/output from *PeleLM* are specified in the `IO CONTROL` block:

```
#-------------------------IO CONTROL----------------------------
#amr.restart            = chk01000 # Restart from checkpoint ?
#amr.regrid_on_restart = 1         # Perform regriding upon restart ?
amr.checkpoint_files_output = 0    # Dump check file ? 0: no, 1: yes
amr.check_file      = chk          # root name of checkpoint file
amr.check_int       = 100          # number of timesteps between checkpoints
amr.plot_file       = plt          # root name of plotfiles
amr.plot_int        = 20           # number of timesteps between plotfiles
amr.derive_plot_vars=rhoRT mag_vort avg_pressure gradpx gradpy diveru mass_fractions␣
↪mixfrac
amr.grid_log        = grdlog       # name of grid logging file
```

The first two lines (commented out for now) are only used when restarting a simulation from a *checkpoint* file and will be useful later during this tutorial. Information pertaining to the checkpoint and plot_file files name and output frequency can be specified there. *PeleLM* will always generate an initial plotfile `plt_00000` if the initialization is properly completed, and a final plotfile at the end of the simulation. It is possible to request including *derived variables* in the plotfiles by appending their names to the `amr.derive_plot_vars` keyword. These variables are derived from the *state variables* (velocity, density, temperature, ρY_k, ρh) which are automatically included in the plotfile. Note also that the name of the `probin` file used to specify the initial/boundary conditions is defined here.

You finally have all the information necessary to run the first of several steps to generate a steady triple flame. Type in:

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

A lot of information is printed directly on the screen during a *PeleLM* simulation, but it will not be detailed in the present tutorial. If you wish to store these information for later analysis, you can instead use:

```
./PeleLM2d.gnu.MPI.ex inputs.2d-regt > logCheckInitialSolution.dat &
```

Whether you have used one or the other command, within 30 s you should obtain a `plt_00000` and a `plt_00001` files (or even more, appended with .old********** if you used both commands). Use Amrvis to vizualize `plt_00000` and make sure the solution matches the one shown in Fig. 7.6.

Running the problem on a coarse grid

As mentioned above, the initial solution is relatively far from the steady-state triple flame we wish to obtain. An inexpensive and rapid way to transition from the initial solution to an established triple flame is to perform a coarse (using only 2 AMR levels) simulation using a single SDC iteration for a few initial number of time steps (here we start with 1000). To do so, update (or verify !) these associated keywords in the `input.2d-regt`:

```
#------------------------AMR CONTROL----------------------------
...
amr.max_level     = 1             # maximum level number allowed
...
#---------------------TIME STEPING CONTROL---------------------
...
max_step          = 1000          # maximum number of time steps
...
#-------------------NUMERICS CONTROL-------------------------
...
ns.sdc_iterMAX    = 1             # Number of SDC iterations
```

In order to later on continue the simulation with refined parameters, we need to trigger the generation of a checkpoint file, in the `IO CONTROL` block:

```
amr.checkpoint_files_output = 1   # Dump check file ? 0: no, 1: yes
```

To be able to complete this first step relatively quickly, it is advised to run *PeleLM* using at least 4 MPI processes. It will then take a couple of hours to reach completion. To be able to monitor the simulation while it is running, use the following command:

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt > logCheckInitialTransient.dat &
```

A plotfile is generated every 20 time steps (as specified via the `amr.plot_int` keyword in the `IO CONTROL` block). This will allow you to visualize and monitor the evolution of the flame. Use the following command to open multiple plotfiles at once with Amrvis:

```
amrvis -a plt????0
```

An animation of the flame evolution during this initial transient is provided in Fig 7.7.

7.7: Temperature (left) and divu (right) fields from 0 to 2000 time steps (0-?? ms).

Steady-state problem: activating the flame control

The speed of propagation of a triple flame is not easy to determine a-priori. As such it is useful, at least until the flame settles, to have some sort of stabilization mechanism to prevent flame blow-off or flashback. In the present configuration, the position of the flame front can be tracked at each time step (using an isoline of temperature) and the input velocity is adjusted to maintain its location at a fixed distance from the inlet (1 cm in the present case).

The parameters of the active control are listed in *INPUTS TO ACTIVE CONTROL* block of `inputs.2d-regt`:

```
# --------------  INPUTS TO ACTIVE CONTROL  -----------------
active_control.on = 1                    # Use AC ?
active_control.use_temp = 1              # Default in fuel mass, rather use iso-T␣
↪position ?
active_control.temperature = 1400.0      # Value of iso-T ?
active_control.tau = 1.0e-4              # Control tau (should ~ 10 dt)
active_control.height = 0.01             # Where is the flame held ? Default assumes␣
↪coordinate along Y in 2D or Z in 3D.
active_control.v = 1                     # verbose
active_control.velMax = 2.0              # Optional: limit inlet velocity
active_control.changeMax = 0.1           # Optional: limit inlet velocity changes␣
↪(absolute)
active_control.flameDir  = 1             # Optional: flame main direction. Default:␣
↪AMREX_SPACEDIM-1
active_control.pseudo_gravity = 1        # Optional: add density proportional force to␣
↪compensate for the acceleration
                                         #           of the gas due to inlet velocity␣
↪changes
```

The first keyword activates the active control and the second one specify that the flame will be tracked based on an iso-line of temperature, the value of which is provided in the third keyword. The following parameters controls the relaxation of the inlet velocity to the steady state velocity of the triple flame. `tau` is a relaxation time scale, that should be of the order of ten times the simulation time-step. `height` is the user-defined location where the triple flame should settle, `changeMax` and `velMax` control the maximum velocity increment and maximum inlet velocity, respectively. The user is referred to [CAMCS2006] for an overview of the method and corresponding parameters. The `pseudo_gravity` triggers a manufactured force added to the momemtum equation to compensate for the acceleration of different density gases.

Once these paremeters are set, you continue the previous simulation by uncommenting the first two lines of the `IO CONTROL` block in the input file:

```
amr.restart            = chk01000 # Restart from checkpoint ?
amr.regrid_on_restart = 1         # Perform regriding upon restart ?
```

The first line provides the last *checkpoint* file generated during the first simulation performed for 1000 time steps. Note that the second line, forcing regriding of the simulation upon restart, is not essential at this point. Finally, update the `max_step` to allow the simulation to proceed further:

```
#----------------------TIME STEPING CONTROL----------------------
...
max_step          = 2000          # maximum number of time steps
```

You are now ready launch *PeleLM* again for another 1000 time steps !

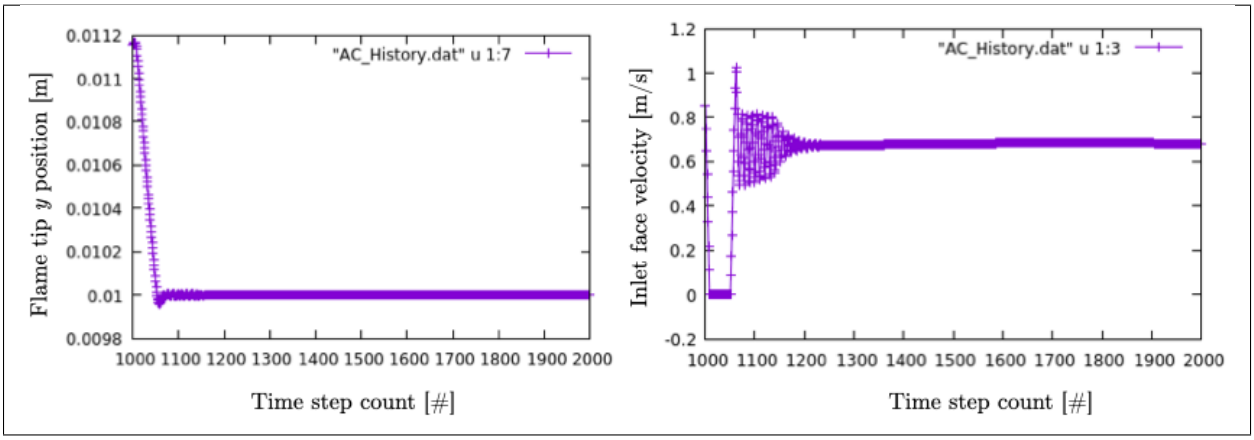
```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt > logCheckControl.dat &
```

As the simulation proceeds, an ASCII file tracking the flame position and inlet velocity (as well as other control variables) is generated: `AC_History`. You can follow the motion of the flame tip by plotting the eigth column against the first one (flame tip vs. time step count). If *gnuplot* is available on your computer, use the following to obtain the graphs of Fig 7.8:

```
gnuplot
plot "AC_History" u 1:7 w lp
plot "AC_History" u 1:3 w lp
exit
```

The second plot corresponds to the inlet velocity.

7.8: Flame tip position (left) and inlet velocity (right) as function of time step count from 1000 to 2000 step using the inlet velocity control.

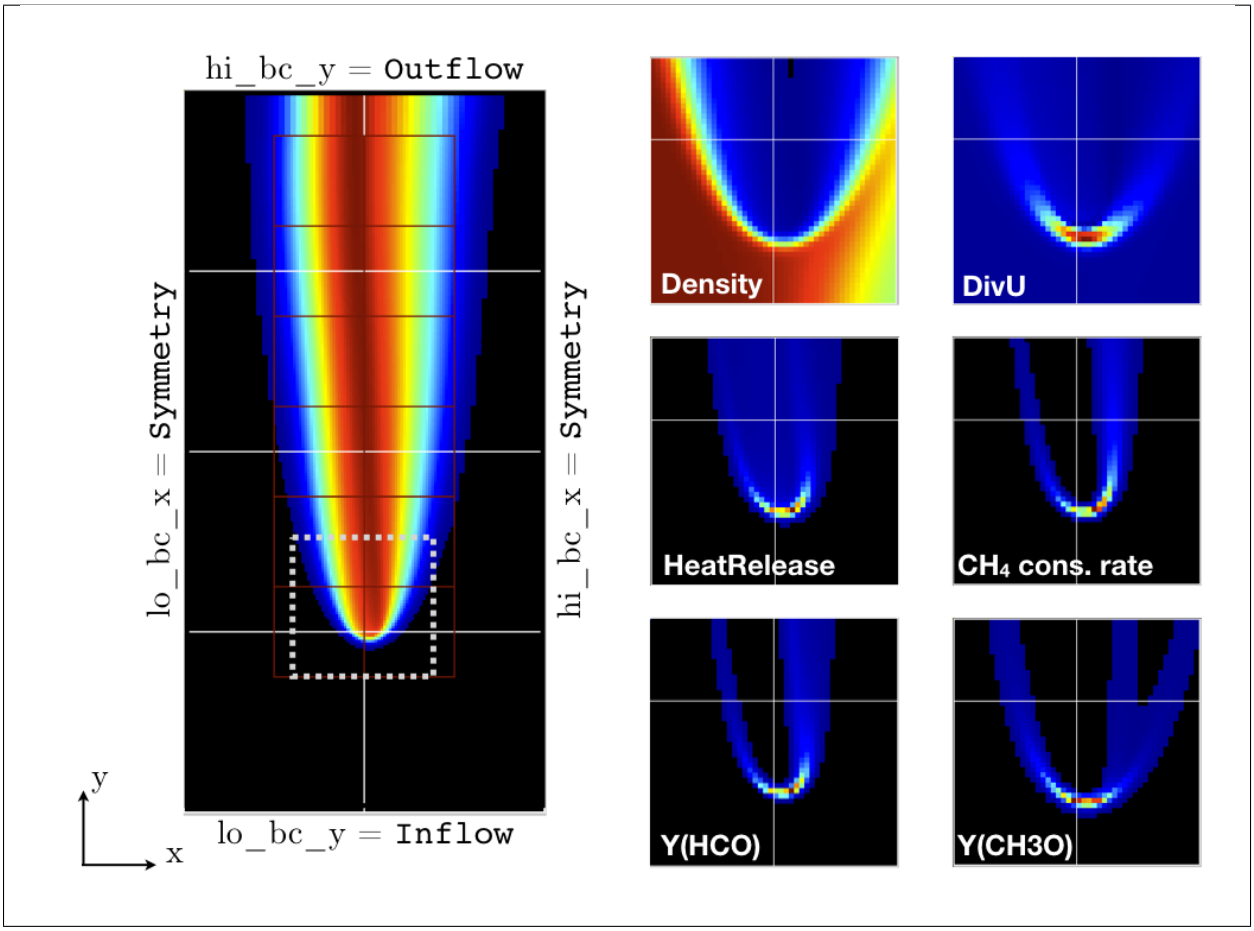


At this point, you have a stabilized methane/air triple flame and will now use AMR features to improve the quality of your simulation.

7.2.6 Refinement of the computation

Before going further, it is important to look at the results of the current simulation. The left panel of Fig. 7.9 displays the temperature field, while a zoom-in of the flame edge region colored by several important variables is provided on the right side. Note that *DivU*, the *HeatRelease* and the *CH4_consumption* are good markers of the reaction/diffusion processes in our case. What is striking from these images is the lack of resolution of the triple flame, particularly in the reaction zone. We also clearly see square unsmooth shapes in the field of intermediate species, where *Y(HCO)* is found to closely match the region of high *CH4_consumption* while *Y(CH3O)* is located closer to the cold gases, on the outer layer of the triple flame.

7.9: Details of the triple flame tip obtained with the initial coarse 2-level mesh.



Our first level of refinement must specifically target the reactive layer of the flame. As seen from Fig. 7.9, one can choose from several variables to reach that goal. In the following, we will use the CH3O species as a tracer of the flame position. Start by increasing the number of AMR levels by one in the *AMR CONTROL* block:

```
amr.max_level       = 2          # maximum level number allowed
```

Then provide a definition of the new refinement critera in the *REFINEMENT CONTROL* block:

```
#--------------------REFINEMENT CONTROL-----------------------
amr.refinement_indicators = hi_temp gradT flame_tracer   # Declare set of refinement␣
↪indicators

amr.hi_temp.max_level     = 1
amr.hi_temp.value_greater = 800
amr.hi_temp.field_name    = temp

amr.gradT.max_level                   = 1
amr.gradT.adjacent_difference_greater = 200
amr.gradT.field_name                  = temp

amr.flame_tracer.max_level     = 2
amr.flame_tracer.value_greater = 1.0e-6
```

(continues on next page)

(continued from previous page)

```
amr.flame_tracer.field_name    = Y(CH3O)
```

The first line simply declares a set of refinement indicators which are subsequently defined. For each indicator, the user can provide a limit up to which AMR level this indicator will be used to refine. Then there are multiple possibilities to specify the actual criterion: `value_greater`, `value_less`, `vorticity_greater` or `adjacent_difference_greater`. In each case, the user specify a threshold value and the name of variable on which it applies (except for the `vorticity_greater`). In the example above, the grid is refined up to level 1 at the location wheres the temperature is above 800 K or where the temperature difference between adjacent cells exceed 200 K. These two criteria were used up to that point. The last indicator will now enable to add level 2 grid patches at location where the flame tracer (*Y(CH3O)*) is above 1.0e-6.

With these new parameters, update the *checkpoint* file from which to restart:

```
amr.restart           = chk02000 # Restart from checkpoint ?
```

and increase the `max_step` to 2300 and start the simulation again !

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt > log3Levels.dat &
```

Visualization of the 3-levels simulation results indicates that the flame front is now better repesented on the fine grid, but there are still only a couple of cells across the flame front thickness. The flame tip velocity, captured in the *AC_history*, also exhibits a significant change with the addition of the third level (even past the initial transient). In the present case, the flame tip velocity is our main quantity of interest and we will now add another refinement level to ensure that this quantity is fairly well capture. We will use the same refinement indicators and simply update the `max_level` as well as the level at which each refinement criteria is used:

```
amr.max_level       = 3          # maximum level number allowed

...

amr.restart           = chk02300 # Restart from checkpoint ?

...

amr.gradT.max_level                    = 2

...

amr.flame_tracer.max_level     = 3
```

and increase the `max_step` to 2600. The temporal evolution of the inlet velocity also shows that our active control parameters induce rather strong oscillations of the velocity before it settles. To illustrate how we can tune the AC parameters to limit this behavior, we will increase the `tau` parameter:

```
active_control.tau = 4.0e-4            # Control tau (should ~ 10 dt)
```

Let's start the simulation again !

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt > log4Levels.dat &
```

Finally, we will now improve *PeleLM* algorithm accuracy itself. So far, for computational expense reasons, we have only used a single SDC iteration which provide a relatively weak coupling between the slow advection and the fast diffusion/reaction processes, as well as a loose enforcement of the velocity divergence constrain (see PeleLM description for more information). We will now increase the number of SDC iteration to two, allowing to reach the theoretical second order convergence property of the algorithm:

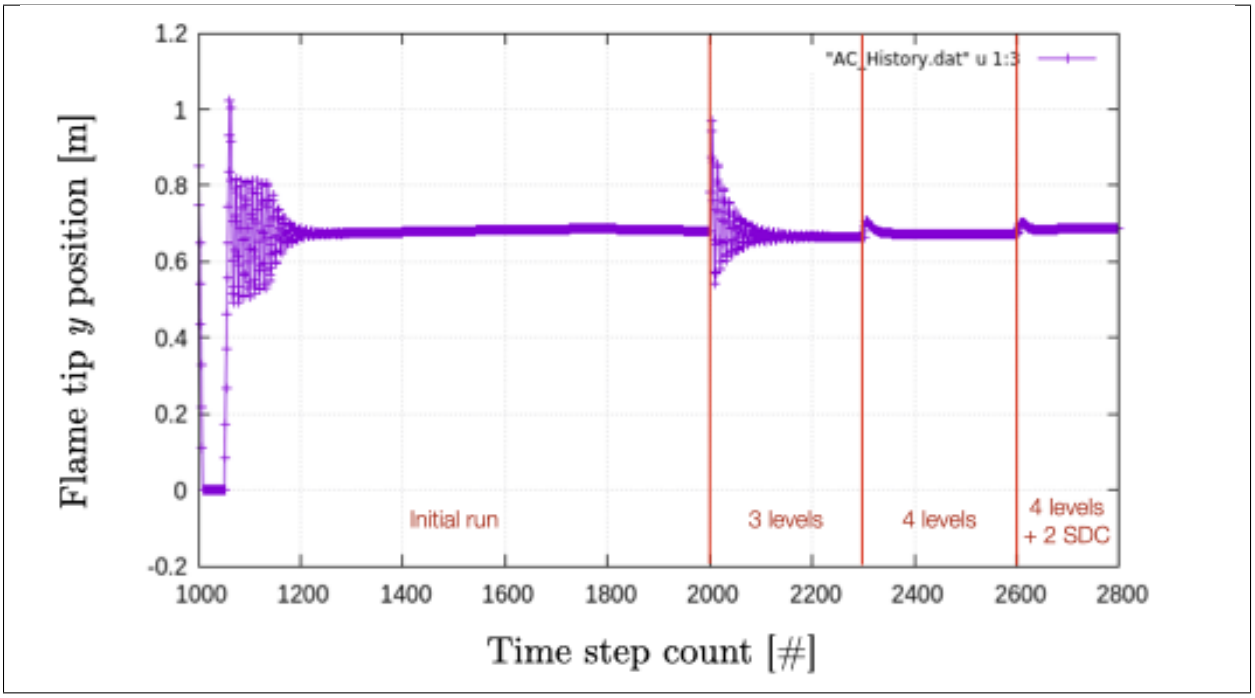
```
#--------------------NUMERICS CONTROL-------------------------
 ...
 ns.sdc_iterMAX     = 2               # Number of SDC iterations
```

and further continue the simulation to reach 2800 time steps. Note that, as with an increase of the maximum refinement level, increasing the number of SDC iterations incurs a significant increase of the computational time per coarse time step. Let's complete this final step:

```
mpirun -n 4 ./PeleLM2d.gnu.MPI.ex inputs.2d-regt > log4Levels_2SDC.dat &
```

Figure 7.10 shows the entire history of the inlet velocity starting when the AC was activated (1000th time step). We can see that every change in the numerical setup induced a slight change in the triple flame propagation velocity, eventually leading to a nearly constant value, sufficient for the purpose of this tutorial.

7.10: Inlet velocity history during the successive simulations performed during this tutorial.



At this point, the simulation is considered complete and the next section provide some pointer to further analyze the results.

7.2.7 Analysis

CHAPTER 8

Indices and tables

- genindex
- search

Bibliography

[PCI2007] S. Chung, Stabilization, propagation and instability of tribrachial triple flames, Proceedings of the Combustion Institute 31 (2007) 877–892

[CF1990] R. Bilger, S. Starner, R. Kee, On reduced mechanisms for methane-air combustion in nonpremixed flames, Combustion and Flames 80 (1990) 135-149

[CAMCS2006] J. Bell, M. Day, J. Grcar, M. Lijewski, Active Control for Statistically Stationary Turbulent PremixedFlame Simulations, Communications in Applied Mathematics and Computational Science 1 (2006) 29-51