CASTRO

An adaptive, parallel, radiation hydrodynamics code
for self-gravitating astrophysical flows

User’s Guide

April 4, 2013
Chapter Listing

list of figures xii

list of tables xiii

1 Introduction 1

2 Getting Started 3

3 Inputs Files 7

4 Units and Constants 21

5 Equations 23

6 Gravity 27

7 Rotation 31

8 Single-Level Flow Chart 35

9 Level Sets 45

10 AMR 55

11 Equation of State and Burning Network 59
<table>
<thead>
<tr>
<th>Chapter Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 ConvertCheckpoint</td>
<td>63</td>
</tr>
<tr>
<td>13 Initializing CASTRO with MAESTRO Data</td>
<td>69</td>
</tr>
<tr>
<td>14 Visualization</td>
<td>73</td>
</tr>
<tr>
<td>15 Software Framework</td>
<td>77</td>
</tr>
<tr>
<td>16 Verification Test Problems</td>
<td>83</td>
</tr>
<tr>
<td>17 Managing Jobs on Jaguar</td>
<td>99</td>
</tr>
<tr>
<td>18 Scaling</td>
<td>101</td>
</tr>
<tr>
<td>19 Suggestions, Warnings, and Gotchas</td>
<td>103</td>
</tr>
<tr>
<td>References</td>
<td>105</td>
</tr>
</tbody>
</table>
Contents

list of figures xii

list of tables xiii

1 Introduction 1

2 Getting Started 3
   2.1 Downloading the Code ........................................ 3
   2.2 Building the Code ............................................. 3
   2.3 Running the Code .............................................. 4
   2.4 Visualization of the Results .................................. 4

3 Inputs Files 7
   3.1 Problem Geometry ............................................. 7
      3.1.1 List of Parameters ....................................... 7
      3.1.2 Examples of Usage ....................................... 7
   3.2 Domain Boundary Conditions ................................ 8
      3.2.1 List of Parameters ....................................... 8
      3.2.2 Notes ................................................... 8
      3.2.3 Examples of Usage ....................................... 8
   3.3 Resolution .................................................... 8
      3.3.1 List of Parameters ....................................... 8
      3.3.2 Notes ................................................... 8
   3.4 Tagging ....................................................... 9
      3.4.1 List of Parameters ....................................... 9
      3.4.2 Notes ................................................... 9
   3.5 Regridding .................................................. 10
      3.5.1 Overview ................................................ 10
      3.5.2 List of Parameters ..................................... 10
      3.5.3 Notes ................................................... 10
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.4 Examples of Usage</td>
<td>10</td>
</tr>
<tr>
<td>3.5.5 How Grids are Created</td>
<td>11</td>
</tr>
<tr>
<td>3.6 Simulation Time</td>
<td>11</td>
</tr>
<tr>
<td>3.6.1 List of Parameters</td>
<td>11</td>
</tr>
<tr>
<td>3.6.2 Notes</td>
<td>11</td>
</tr>
<tr>
<td>3.6.3 Examples of Usage</td>
<td>12</td>
</tr>
<tr>
<td>3.7 Time Step</td>
<td>12</td>
</tr>
<tr>
<td>3.7.1 List of Parameters</td>
<td>12</td>
</tr>
<tr>
<td>3.7.2 Notes</td>
<td>12</td>
</tr>
<tr>
<td>3.8 Subcycling</td>
<td>13</td>
</tr>
<tr>
<td>3.8.1 List of Parameters</td>
<td>13</td>
</tr>
<tr>
<td>3.8.2 Examples of Usage</td>
<td>13</td>
</tr>
<tr>
<td>3.9 Restart Capability</td>
<td>13</td>
</tr>
<tr>
<td>3.9.1 List of Parameters</td>
<td>14</td>
</tr>
<tr>
<td>3.9.2 Notes</td>
<td>14</td>
</tr>
<tr>
<td>3.9.3 Examples of Usage</td>
<td>14</td>
</tr>
<tr>
<td>3.10 Controlling PlotFile Generation</td>
<td>15</td>
</tr>
<tr>
<td>3.10.1 List of Parameters</td>
<td>15</td>
</tr>
<tr>
<td>3.10.2 Notes</td>
<td>15</td>
</tr>
<tr>
<td>3.10.3 Examples of Usage</td>
<td>15</td>
</tr>
<tr>
<td>3.11 Screen Output</td>
<td>16</td>
</tr>
<tr>
<td>3.11.1 List of Parameters</td>
<td>16</td>
</tr>
<tr>
<td>3.11.2 Notes</td>
<td>16</td>
</tr>
<tr>
<td>3.11.3 Examples of Usage</td>
<td>16</td>
</tr>
<tr>
<td>3.12 Gravity</td>
<td>17</td>
</tr>
<tr>
<td>3.12.1 List of Parameters</td>
<td>17</td>
</tr>
<tr>
<td>3.12.2 Notes</td>
<td>17</td>
</tr>
<tr>
<td>3.13 Diffusion</td>
<td>17</td>
</tr>
<tr>
<td>3.13.1 List of Parameters</td>
<td>17</td>
</tr>
<tr>
<td>3.13.2 Notes</td>
<td>17</td>
</tr>
<tr>
<td>3.14 Rotation</td>
<td>18</td>
</tr>
<tr>
<td>3.14.1 List of Parameters</td>
<td>18</td>
</tr>
<tr>
<td>3.14.2 Notes</td>
<td>18</td>
</tr>
<tr>
<td>3.15 Physics</td>
<td>18</td>
</tr>
<tr>
<td>3.15.1 List of Parameters</td>
<td>18</td>
</tr>
<tr>
<td>3.15.2 Notes</td>
<td>18</td>
</tr>
<tr>
<td><strong>4 Units and Constants</strong></td>
<td>21</td>
</tr>
<tr>
<td>4.1 Units and Constants</td>
<td>21</td>
</tr>
<tr>
<td><strong>5 Equations</strong></td>
<td>23</td>
</tr>
<tr>
<td>5.1 Conservation Forms</td>
<td>23</td>
</tr>
<tr>
<td>5.2 Primitive Forms</td>
<td>25</td>
</tr>
<tr>
<td><strong>6 Gravity</strong></td>
<td>27</td>
</tr>
<tr>
<td>6.1 Types of Approximations</td>
<td>27</td>
</tr>
<tr>
<td>6.2 GR correction</td>
<td>29</td>
</tr>
</tbody>
</table>
CONTENTS

12.2.1 Converting the Checkpoint File .............................................. 65
12.2.2 Restarting from a Grown Checkpoint File ................................... 66
12.2.3 Cylindrical Coordinates .......................................................... 66

13 Initializing CASTRO with MAESTRO Data ......................................... 69
  13.1 Overview ................................................................................. 69
  13.2 MAESTRO Plotfile Requirements .............................................. 69
  13.3 List of Parameters ................................................................. 70
    13.3.1 Examples of Usage ............................................................ 70
  13.4 New Subroutines in Prob_Xd.f90 ............................................... 71
  13.5 Additional Notes ................................................................. 71
    13.5.1 Multilevel Restart .............................................................. 71

14 Visualization .................................................................................. 73
  14.1 2D and 3D .............................................................................. 73
    14.1.1 amrvis ............................................................................. 73
    14.1.2 Visit ............................................................................... 73
  14.2 Controlling What's in the PlotFile .............................................. 74
  14.3 1D ......................................................................................... 74

15 Software Framework ....................................................................... 77
  15.1 Code structure ....................................................................... 77
  15.2 Castro Data Structures ............................................................ 78
    15.2.1 State Data ...................................................................... 78
    15.2.2 Other Quantities .............................................................. 79
  15.3 Setting Up Your Own Problem .................................................. 79
  15.4 Boundaries ............................................................................ 80
    15.4.1 Boundaries Between Grids .................................................. 80
    15.4.2 Physical Boundaries .......................................................... 80
  15.5 Parallel I/O ............................................................................ 81

16 Verification Test Problems ............................................................. 83
  16.1 Hydrodynamics Test Problems .................................................. 83
    16.1.1 Sod's Problem (and Other Shock Tube Problems) ............... 83
    16.1.2 Sedov Problem ................................................................ 87
    16.1.3 Rayleigh-Taylor ............................................................... 89
  16.2 Gravity Test Problems ............................................................... 91
  16.3 Radiation Test Problems ............................................................ 91
    16.3.1 Light Front ...................................................................... 92
    16.3.2 Diffusion of a Gaussian Pulse ........................................... 92
    16.3.3 Radiation Source Problem ............................................... 92
    16.3.4 Radiating Sphere ............................................................. 92
  16.4 Regression Testing .................................................................. 92
    16.4.1 Test Suite Inputs File ....................................................... 94
    16.4.2 Initializing the Test Suite .................................................... 96
    16.4.3 Regular Use ..................................................................... 96
    16.4.4 Updating Benchmarks ....................................................... 98
12.1 Data from checkpoint file before and after the domain has been coarsened and grown. This case uses \texttt{star\_at\_center = 0} and \texttt{ref\_ratio=2}. The first grown example has \texttt{grown\_factor}=2, the second has \texttt{grown\_factor}=3. In all figures the level 0 grids are shown in white, the level 1 grids in red, the level 2 grids in yellow, and in the grown figures, the level 3 grids are in pink. .......................... 67

12.2 Data from checkpoint file before and after the domain has been coarsened and grown. This case uses \texttt{star\_at\_center = 0} and \texttt{ref\_ratio=2}. The first grown example has \texttt{grown\_factor}=2, the second has \texttt{grown\_factor}=3. In all figures the level 0 grids are shown in white, the level 1 grids in red, the level 2 grids in yellow, and in the grown figure, the level 3 grids are in pink. .......................... 68

16.1 Castro solution for Sod’s problem run in 3-d, with the newest ppm limiters, along the \textit{x}, \textit{y}, and \textit{z} axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line. .......................... 84

16.2 Castro solution for Sod’s problem run in 3-d, with the piecewise-linear Godunov method with limiters, along the \textit{x}, \textit{y}, and \textit{z} axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line. .......................... 85

16.3 Castro solution for the double rarefaction problem run in 3-d, along the \textit{x}, \textit{y}, and \textit{z} axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line. .......................... 86

16.4 Castro solution for the strong shock problem run in 3-d, along the \textit{x}, \textit{y}, and \textit{z} axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line. .......................... 87

16.5 Castro solution for the Sedov blast wave problem run in 1-d spherical, 2-d axisymmetric, and 3-d Cartesian coordinates. Each of these geometries produces a spherical Sedov explosion. .......................... 90

16.6 Castro solution for the Sedov blast wave problem run in 2-d Cartesian coordinates. This corresponds to a cylindrical Sedov explosion. .......................... 90

16.7 Rayleigh-Taylor with different PPM types. .......................... 91
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.8</td>
<td>Castro solution for radiating source test problem. Heating and cooling solutions are shown as a function of time, compared to the analytic solution. The gray photon solver was used.</td>
</tr>
<tr>
<td>16.9</td>
<td>Castro solution for radiating sphere problem, showing the radiation energy density as a function of energy group. This test was run with 64 photon energy groups.</td>
</tr>
<tr>
<td>16.10</td>
<td>Main test suite results page. Each row indicates a single test suite run, arranged by date, and each column indicates a different test problem. Note: this page is from the <em>Maestro</em> code, but a <em>Castro</em> test suite run will produce similar output.</td>
</tr>
<tr>
<td>16.11</td>
<td>The test suite output for a single day’s run. Each row indicates a separate test, showing whether they passed or failed. Clicking on the test name will give more information about that particular test on that day. Note: this page is from the <em>Maestro</em> code, but a <em>Castro</em> test suite run will produce similar output.</td>
</tr>
<tr>
<td>18.1</td>
<td>Scaling behavior of Sod problem on franklin.nersc.gov</td>
</tr>
<tr>
<td>18.2</td>
<td>Scaling behavior of ScalingTest problem on jaguarpf.ccs.ornl.gov</td>
</tr>
</tbody>
</table>
15.1 Conversions from physical to mathematical BCs . . . . . . . . . . . . . . . . . . . . . . 80
16.1 Sedov inputs files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
16.2 Analysis routines for Sedov . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
CHAPTER 1

Introduction

Welcome to the CASTRO User’s Guide!

In this User’s Guide we describe how to download and run CASTRO, a massively parallel code that solves the multicomponent compressible hydrodynamic equations for astrophysical flows including self-gravity, nuclear reactions and radiation. CASTRO uses an Eulerian grid and incorporates adaptive mesh refinement (AMR). Our approach to AMR uses a nested hierarchy of logically-rectangular grids with simultaneous refinement in both space and time.

For more information about CASTRO, including the first two papers (published in ApJ) that describe CASTRO with and without the radiation solvers, please visit the CCSE web site, at

ccse.lbl.gov/Research/CASTRO
2.1 Downloading the Code

CASTRO is built on top of the BoxLib framework. In order to run CASTRO, you must download two separate git modules.

First, make sure git is installed on your machine – and we recommend version 1.7.x or higher.

1. Download the BoxLib repository by typing
   
   git clone https://ccse.lbl.gov/pub/Downloads/BoxLib.git

   This will create a directory called BoxLib on your machine. Put this somewhere out of the way and set the environment variable, BOXLIB_HOME, on your machine to the path name where you have put BoxLib. You will want to periodically update BoxLib by typing
   
   git pull

   in the BoxLib directory.

2. Now download the CASTRO repository by typing
   
   git clone https://ccse.lbl.gov/pub/Downloads/Castro.git

2.2 Building the Code

1. From the directory in which you checked out the Castro git repo, type
   
   cd Castro/Exec/Sedov
This will put you into a directory in which you can run the Sedov problem in 1-d, 2-d or 3-d.

2. In Sedov, edit the GNUmakefile, and set

   DIM = 2 (for example)
   COMP = your favorite C++ compiler
   FCOMP = your favorite Fortran compiler (which must compile F90)
   DEBUG = FALSE

   We like COMP = gcc and FCOMP = gfortran. If you want to try other compilers and they don’t work, please let us know.

   To build a serial (single-processor) code, set USE_MPI = FALSE. This will compile the code without the MPI library. If you want to do a parallel run, then you would set USE_MPI = TRUE. In this case, the build system will need to know about your MPI installation. This can be done by editing the makefiles in the BoxLib tree, but a simple method is to set the shell environment variable BOXLIB_USE_MPI_WRAPPERS=1. If this is set, then the build system will fall back to using the local MPI compiler wrappers (e.g. mpic++ and mpif90) to do the build.

3. Now type ”make”. The resulting executable will look something like ”Castro2d.Linux.gcc.gfortran.ex”, which means this is a 2-d version of the code, made on a Linux machine, with COMP = gcc and FCOMP = gfortran.

### 2.3 Running the Code

1. Type ”Castro2d.Linux.gcc.gfortran.ex inputs.2d.cyl_in_cartcoords” This will run the 2-d cylindrical Sedov problem in Cartesian (x-y coordinates). You can see other possible options, which should be clear by the names of the inputs files.

2. You will notice that running the code generates directories that look like plt00000, plt00020, etc, and chk00000, chk00020, etc. These are ”plotfiles” and ”checkpoint” files. The plotfiles are used for visualization, the checkpoint files are used for restarting the code.

### 2.4 Visualization of the Results

1. To visualize the plotfiles, you can use a freely available visualization package like VisIt, or you can try “Amrvis.” To get Amrvis, type

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

   Then cd into Amrvis, ddit the GNUmakefile there to set DIM = 2, and again set COMP and FCOMP to compilers that you have. Leave DEBUG = FALSE. Then type ”make”. This will make an executable that looks like ”amrvis2d...ex”.

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

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

   Then cd into volpack and type make.
Note: This 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. leastif gives some functionality and will allow you to build the amrvis executable, but amrvis will not run properly.

Note: On most Linux distributions, 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 package management tool to install them, which varies from distribution to distribution, but is straightforward. I can provide detailed instructions if anyone needs them.

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

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

2. Return to the Castro/Exec/Sedov directory. Type "amrvis2d plt00152" to see a single plotfile, or "amrvis2d -a plt*", which will animate the sequence of plotfiles. Try playing around with this – note you can change which variable you are looking at, you can select a region and click "Dataset" (under View) in order to look at the actual numbers, etc. You can also export the pictures in several different formats – under "File", see "Export".

Please know that we do have a number of conversion routines to other formats (such as matlab), but it is hard to describe them all. If you would like to display the data in another format, please let us know (again, asalmgren@lbl.gov) and we will point you to whatever we have that can help.

You have now completed a brief introduction to CASTRO.
CHAPTER 3

Inputs Files

The Castro executables read run-time information from an "inputs" file (which you put on the command line) and from a "probin" file, the name of which is usually defined in the inputs file, but which defaults to "probin". To set the "probin" file name in the inputs file:

\texttt{amr.probin\_file = my\_special\_probin}

for example, has the Fortran code read a file called "my\_special\_probin"

3.1 Problem Geometry

3.1.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>geometry.prob_lo</td>
<td>physical location of low corner of the domain</td>
<td>Real</td>
<td>must be set</td>
</tr>
<tr>
<td>geometry.prob_hi</td>
<td>physical location of high corner of the domain</td>
<td>Real</td>
<td>must be set</td>
</tr>
<tr>
<td>geometry.coord_sys</td>
<td>coordinate system</td>
<td>0 = Cartesian, 1 = r-z, 2 = spherical</td>
<td>must be set</td>
</tr>
<tr>
<td>geometry.is_periodic</td>
<td>is the domain periodic in this direction</td>
<td>0 if false, 1 if true</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

3.1.2 Examples of Usage

- \texttt{geometry.prob\_lo = 0 0 0}
  defines the low corner of the domain at (0,0,0) in physical space.

- \texttt{geometry.prob\_hi = 1.e8 2.e8 2.e8}
  defines the high corner of the domain at (1.e8,2.e8,2.e8) in physical space.
• \texttt{geometry.coord\_sys} = 0  
defines the coordinate system as Cartesian  

• \texttt{geometry.is\_periodic} = 0 1 0  
says the domain is periodic in the y-direction only.

\section*{3.2 Domain Boundary Conditions}

\subsection*{3.2.1 List of Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{castro.lo_bc}</td>
<td>boundary type of each low face</td>
<td>0,1,2,3,4,5</td>
<td>must be set</td>
</tr>
<tr>
<td>\texttt{castro.hi_bc}</td>
<td>boundary type of each high face</td>
<td>0,1,2,3,4,5</td>
<td>must be set</td>
</tr>
</tbody>
</table>

\subsection*{3.2.2 Notes}

Boundary types are:

\begin{align*}
0 & \text{ – Interior / Periodic} \\
1 & \text{ – Inflow} \\
2 & \text{ – Outflow} \\
3 & \text{ – Symmetry} \\
4 & \text{ – Slip Wall} \\
5 & \text{ – No Slip Wall}
\end{align*}

Note – \texttt{castro.lo\_bc} and \texttt{castro.hi\_bc} must be consistent with \texttt{geometry.is\_periodic} – if the domain is periodic in a particular direction then the low and high bc’s must be set to 0 for that direction.

\subsection*{3.2.3 Examples of Usage}

\begin{itemize}
  \item \texttt{castro.lo\_bc} = 1 4 0
  \item \texttt{castro.hi\_bc} = 2 4 0
  \item \texttt{geometry.is\_periodic} = 0 0 1
\end{itemize}

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

\section*{3.3 Resolution}

\subsection*{3.3.1 List of Parameters}

Note: if \texttt{amr.max\_level} = 0 then you do not need to set \texttt{amr.ref\_ratio} or \texttt{amr.regrid\_int}.  

### 3.4 Tagging

#### 3.4.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr.n.cell</td>
<td>number of cells in each direction at the coarsest level</td>
<td>Integer &gt; 0</td>
<td>must be set</td>
</tr>
<tr>
<td>amr.max_level</td>
<td>number of levels of refinement above the coarsest level</td>
<td>Integer ≥ 0</td>
<td>must be set</td>
</tr>
<tr>
<td>amr.ref_ratio</td>
<td>ratio of coarse to fine grid spacing between subsequent levels</td>
<td>Integer &gt; 0 2 or 4</td>
<td>must be set</td>
</tr>
<tr>
<td>amr.regrid_int</td>
<td>how often to regrid</td>
<td>Integer &gt; 0</td>
<td>must be set</td>
</tr>
<tr>
<td>amr.regrid_on_restart</td>
<td>should we regrid immediately after restarting</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>castro.allow_untagging</td>
<td>are cells allowed to be &quot;untagged&quot;</td>
<td>0 or 1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 3.4.2 Notes

- Typically cells at a given level can be tagged as needing refinement by any of a number of criteria, but cannot be "untagged", i.e. once tagged no other criteria can untag them. If we set `castro.allow_untagging = 1` then the user is allowed to "untag" cells in the Fortran tagging routines.
3.5 Regridding

3.5.1 Overview

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

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

3.5.2 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr.regrid_file</td>
<td>name of file from which to read the grids</td>
<td>text</td>
<td>no file</td>
</tr>
<tr>
<td>amr.grid_eff</td>
<td>grid efficiency</td>
<td>Real &gt; 0 and &lt; 1</td>
<td>0.7</td>
</tr>
<tr>
<td>amr.n_error_buf</td>
<td>radius of additional tagging around already tagged cells</td>
<td>Integer ≥ 0</td>
<td>1</td>
</tr>
<tr>
<td>amr.max_grid_size</td>
<td>maximum size of a grid in any direction</td>
<td>Integer &gt; 0</td>
<td>128 in 2D, 32 in 3D</td>
</tr>
<tr>
<td>amr.blocking_factor</td>
<td>grid size must be a multiple of this</td>
<td>Integer &gt; 0</td>
<td>2</td>
</tr>
<tr>
<td>amr.refine_grid_layout</td>
<td>refine grids more if # of processors &gt; # of grids</td>
<td>0 if false, 1 if true</td>
<td>1</td>
</tr>
</tbody>
</table>

3.5.3 Notes

- 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
- amr.max_grid_size at every level must be even
- amr.blocking_factor at every level must be a power of 2
- the domain size must be a multiple of amr.blocking_factor at level 0
- amr.max_grid_size must be a multiple of amr.blocking_factor at every level

3.5.4 Examples of Usage

- amr.regrid_file = fixed_grids
  In this case the list of grids at each fine level are contained in the file, fixed_grids, which will be read 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 = 0.9
  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.

- amr.max_grid_size = 64
  The final grids will be no longer than 64 cells on a side at every level.
• \texttt{amr.max_grid_size} = 64 32 16
  The 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.

• \texttt{amr.blocking_factor} = 32
  The dimensions of all the final grids will be multiples of 32 at all levels.

• \texttt{amr.blocking_factor} = 32 16 8
  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.

Having grids that are large enough to coarsen multiple levels in a V-cycle is essential for good multigrid performance in simulations that use self-gravity.

### 3.5.5 How Grids are Created

The gridding algorithm proceeds in this order:

1. Grids are created using the Berger-Rigoutsis clustering algorithm modified to ensure that all new fine grids are divisible by \texttt{amr.blocking_factor}.

2. Next, the grid list is chopped up if any grids are larger than \texttt{max_grid_size}. Note that because \texttt{amr.max_grid_size} is a multiple of \texttt{amr.blocking_factor} the \texttt{amr.blocking_factor} criterion is still satisfied.

3. Next, if \texttt{amr.refine_grid_layout} = 1 and there are more processors than grids, and
   - if \texttt{amr.max_grid_size} / 2 is a multiple of \texttt{amr.blocking_factor}
     then the grids will be redefined, at each level independently, so that the maximum length of a grid at level \( \ell \), in any dimension, is \texttt{amr.max_grid_size[\ell]} / 2.

4. Finally, if \texttt{amr.refine_grid_layout} = 1, and there are still more processors than grids, and
   - if \texttt{amr.max_grid_size} / 4 is a multiple of \texttt{amr.blocking_factor}
     then the grids will be redefined, at each level independently, so that the maximum length of a grid at level \( \ell \), in any dimension, is \texttt{amr.max_grid_size[\ell]} / 4.

### 3.6 Simulation Time

#### 3.6.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{max_step}</td>
<td>maximum number of level 0 time steps</td>
<td>Integer ( \geq 0 )</td>
<td>-1</td>
</tr>
<tr>
<td>\texttt{stop_time}</td>
<td>final simulation time</td>
<td>Real ( \geq 0 )</td>
<td>-1.0</td>
</tr>
</tbody>
</table>

#### 3.6.2 Notes

To control the number of time steps, you can limit by the maximum number of level 0 time steps (\texttt{max_step}) or by the final simulation time (\texttt{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 pass it.

### 3.6.3 Examples of Usage

- **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.

### 3.7 Time Step

If `castro.do_hydro = 1`, then typically the code chooses a time step based on the CFL number \( dt = cfl \times \frac{dx}{\max(u+c)} \).

#### 3.7.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>castro.cfl</code></td>
<td>CFL number</td>
<td>Real &gt; 0 and ( \leq 1 )</td>
<td>0.8</td>
</tr>
<tr>
<td><code>castro.init_shrink</code></td>
<td>factor by which to shrink the initial time step</td>
<td>Real &gt; 0 and ( \leq 1 )</td>
<td>1.0</td>
</tr>
<tr>
<td><code>castro.change_max</code></td>
<td>factor by which the time step can grow in subsequent steps</td>
<td>Real ( \geq 1 )</td>
<td>1.1</td>
</tr>
<tr>
<td><code>castro.fixed_dt</code></td>
<td>level 0 time step regardless of cfl or other settings</td>
<td>Real &gt; 0</td>
<td>unused if not set</td>
</tr>
<tr>
<td><code>castro.initial_dt</code></td>
<td>initial level 0 time step regardless of other settings</td>
<td>Real &gt; 0</td>
<td>unused if not set</td>
</tr>
<tr>
<td><code>castro.dt_cutoff</code></td>
<td>time step below which calculation will abort</td>
<td>Real &gt; 0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

#### 3.7.2 Examples of Usage

- **castro.cfl = 0.9**
  defines the timestep as \( dt = cfl \times \frac{dx}{\max(u+c)} \).

- **castro.init_shrink = 0.01**
  sets the initial time step to 1% of what it would be otherwise.

- **castro.change_max = 1.1**
  allows the time step to increase by no more than 10% in this case. Note that the time step can shrink by any factor; this only controls the extent to which it can grow.

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

- **castro.initial_dt = 1.e-4**
  sets the initial level 0 time step to be 1.e-4 regardless of `castro.cfl` or `castro.fixed_dt`. The time step can grow in subsequent steps by a factor of `castro.change_max` each step.
3.8—Subcycling

- **castro.dt_cutoff = 1.e-20**
tells the code to abort if the time step ever gets below 1.e-20. This is a safety mechanism so that if things go nuts you don’t burn through your entire computer allocation because you don’t realize the code is misbehaving.

### 3.8 Subcycling

Castro supports a number of different modes for subcycling in time.

- If **amr.subcycling_mode**=Auto (default), then the code will run with equal refinement in space and time. In other words, if level \( n + 1 \) is a factor of 2 refinement above level \( n \), then \( n + 1 \) will take 2 steps of half the duration for every level \( n \) step.

- If **amr.subcycling_mode**=None, then the code will not refine in time. All levels will advance together with a timestep dictated by the level with the strictest \( dt \). Note that this is identical to the deprecated command **amr.nosub = 1**.

- If **amr.subcycling_mode**=Manual, then the code will subcycle according to the values supplied by **subcycling_iterations**.

#### 3.8.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr.subcycling_mode</td>
<td>How shall we subcycle</td>
<td>Auto, None or Manual</td>
<td>Auto</td>
</tr>
<tr>
<td>amr.subcycling_iterations</td>
<td>Number of cycles at each level</td>
<td>1 or ref_ratio</td>
<td></td>
</tr>
</tbody>
</table>

must be set in Manual mode

#### 3.8.2 Examples of Usage

- **amr.subcycling_mode**=Manual
  Subcycle in manual mode with largest allowable timestep.

- **amr.subcycling_iterations = 1 2 1 2**
  Take 1 level 0 timestep at a time (required). Take 2 level 1 timesteps for each level 0 step, 1 timestep at level 2 for each level 1 step, and take 2 timesteps at level 3 for each level 2 step.

- **amr.subcycling_iterations = 2**
  Alternative form. Subcycle twice at every level (except level 0).

### 3.9 Restart Capability

Castro 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):
### 3.9.1 List of Parameters

#### 3.9.2 Notes

- You should specify either `amr.check_int` or `amr.check_per`. Do not try to specify both.

- Note that if `amr.check_per` is used then in order to hit that exact time the code may modify the time step slightly, which will change your results ever so slightly than if you didn’t set this flag.

- Note that `amr.plotfile_on_restart` and `amr.checkpoint_on_restart` only take effect if `amr.regrid_on_restart` is in effect.

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

- 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.

#### 3.9.3 Examples of Usage

- `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`
3.10 Controlling PlotFile Generation

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

### 3.10.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr.plot_file</td>
<td>prefix for plotfiles</td>
<td>Text</td>
<td>&quot;plt&quot;</td>
</tr>
<tr>
<td>amr.plot_int</td>
<td>how often (by level 0 time steps) to write plot files</td>
<td>Integer &gt; 0</td>
<td>-1</td>
</tr>
<tr>
<td>amr.plot_per</td>
<td>how often (by simulation time) to write plot files</td>
<td>Real &gt; 0</td>
<td>-1.0</td>
</tr>
<tr>
<td>amr.plot_vars</td>
<td>name of state variables to include in plotfiles</td>
<td>ALL, NONE or list</td>
<td>ALL</td>
</tr>
<tr>
<td>amr.derive_plot_vars</td>
<td>name of derived variables to include in plotfiles</td>
<td>ALL, NONE or list</td>
<td>NONE</td>
</tr>
<tr>
<td>amr.plotfile_on_restart</td>
<td>should we write plot files</td>
<td>0 or 1</td>
<td>1</td>
</tr>
<tr>
<td>amr.plot_files_output</td>
<td>should we write a plotfile immediately after restarting</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>amr.plot_nfiles</td>
<td>how parallel is the writing of the plotfiles</td>
<td>Integer ≥ 1</td>
<td>64</td>
</tr>
<tr>
<td>castro.plot_phiGrav</td>
<td>Should we plot the gravitational potential</td>
<td>0 or 1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

### 3.10.2 Notes

- You should specify either `amr.plot_int` or `amr.plot_per`. Do not try to specify both.
- Note that if `amr.plot_per` is used then in order to hit that exact time the code may modify the time step slightly, which will change your results ever so slightly than if you didn’t set this flag.
- See the Software Section for more details on parallel I/O and the `amr.plot_nfiles` parameter.
- 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.
- `castro.plot_phiGrav` is only relevant if `castro.do_grav = 1` and `gravity.gravity_type = PoissonGrav`.

### 3.10.3 Examples of Usage

- `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`
• \texttt{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 \texttt{plt\_run00000}, \texttt{plt\_run00043}, \texttt{plt\_run00061}, etc, where \( t = 0.1 \) after 43 level 0 steps, \( t = 0.2 \) after 61 level 0 steps, etc.

### 3.11 Screen Output

#### 3.11.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{amr.v}</td>
<td>verbosity of Amr.cpp</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{castro.v}</td>
<td>verbosity of Castro.cpp</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{gravity.v}</td>
<td>verbosity of Gravity.cpp</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{diffusion.v}</td>
<td>verbosity of Diffusion.cpp</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{mg.v}</td>
<td>verbosity of multigrid solver (for gravity)</td>
<td>0,1,2,3,4</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{amr.grid_log}</td>
<td>name of the file to which the grids are written</td>
<td>Text</td>
<td>not used if not set</td>
</tr>
<tr>
<td>\texttt{amr.run_log}</td>
<td>name of the file to which certain output is written</td>
<td>Text</td>
<td>not used if not set</td>
</tr>
<tr>
<td>\texttt{amr.run_log_terse}</td>
<td>name of the file to which certain (terser) output is written</td>
<td>Text</td>
<td>not used if not set</td>
</tr>
<tr>
<td>\texttt{amr.sum_interval}</td>
<td>if &gt; 0, how often (in level 0 time steps) to compute and print integral quantities</td>
<td>Integer</td>
<td>-1</td>
</tr>
<tr>
<td>\texttt{castro.do_special_tagging}</td>
<td></td>
<td>0 or 1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 3.11.2 Notes

• \texttt{castro.do\_special\_tagging} = 1 allows the user to set a special flag based on user-specified criteria. This can be used, for example, to calculate the bounce time in a core collapse simulation; the bounce time is defined as the first time at which the maximum density in the domain exceeds a user-specified value. This time can then be printed into a special file as a useful diagnostic.

#### 3.11.3 Examples of Usage

• \texttt{amr.grid\_log} = grdlog

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 \texttt{grdlog}.

• \texttt{amr.run\_log} = runlog

Every time step the code prints certain statements to the screen (if \texttt{amr.v} = 1), such as

\[ \text{STEP} = 1 \quad \text{TIME} = 1.91717746 \quad \text{DT} = 1.91717746 \]

\[ \text{PLOTFILE: file = plt00001} \]

Here these statements will be written into \texttt{runlog} as well.

• \texttt{amr.run\_log\_terse} = runlogterse

This file, \texttt{runlogterse} differs from \texttt{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.
3.12 Gravity

3.12.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>castro.do_grav</td>
<td>Include gravity as a forcing term</td>
<td>0 if false, 1 if true</td>
<td>0</td>
</tr>
<tr>
<td>gravity.gravity_type</td>
<td>if castro.do_grav = 1, how shall gravity be calculated</td>
<td>ConstantGrav, PoissonGrav, MonopoleGrav</td>
<td>1</td>
</tr>
<tr>
<td>gravity.const_grav</td>
<td>if gravity.gravity_type = ConstantGrav, set the value of constant gravity</td>
<td>Real</td>
<td>0</td>
</tr>
<tr>
<td>gravity.no_sync</td>
<td>if gravity.gravity_type = PoissonGrav, whether to perform the &quot;sync solve&quot;</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>gravity.no_composite</td>
<td>if gravity.gravity_type = PoissonGrav, whether to perform a composite solve</td>
<td>0 or 1</td>
<td>0</td>
</tr>
</tbody>
</table>

3.12.2 Notes

Gravity types are ConstantGrav, PoissonGrav, or MonopoleGrav. See the Gravity section for more detail.

- To include gravity you must set
  - USE_GRAV = TRUE in the GNUmakefile
  - castro.do_grav = 1 in the inputs file

- gravity.gravity_type is only relevant if castro.do_grav = 1

- gravity.no_sync and gravity.no_composite are only relevant if gravity.gravity_type = PoissonGrav, i.e. the code does a full Poisson solve for self-gravity.

3.13 Diffusion

3.13.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>castro.diffuse_temp</td>
<td>Include thermal diffusion</td>
<td>0 if false, 1 if true</td>
<td>0</td>
</tr>
<tr>
<td>diffusion.diff_coeff</td>
<td>Real &gt; 0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

3.13.2 Notes

- To include diffusion you must set
– USE_DIFFUSION = TRUE in the GNUmakefile
– castro.diffuse_temp = 1 in the inputs file

- You can run a pure diffusion problem (with no hydrodynamics) by setting
  – castro.diffuse_temp = 1
  – castro.do_hydro = 0
- diffusion.diff_coeff is only relevant if castro.diffuse_temp = 1

### 3.14 Rotation

#### 3.14.1 List of Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>castro.do_rotation</td>
<td>Include rotation as a forcing term</td>
<td>0 if false, 1 if true</td>
<td>0</td>
</tr>
<tr>
<td>castro.rotational_frequency</td>
<td>Frequency (Hz) of rotation</td>
<td>Real</td>
<td>0.0</td>
</tr>
</tbody>
</table>

#### 3.14.2 Notes

This is for constant, solid-body rotation about a fixed axis. See the Rotation section for more detail.

- To include rotation you must set
  – USE_ROTATION = TRUE in the GNUMakefile
  – castro.do_rotation = 1 in the inputs file
- castro.rotational_frequency is only relevant if castro.do_rotation = 1

### 3.15 Physics

#### 3.15.1 List of Parameters

#### 3.15.2 Notes

- You must have USE_POINTMASS = TRUE in the GNUmakefile for castro.point_mass to be relevant.
- castro.gamma is only relevant for a gamma law gas.
- castro.use_colglaz = 1 is only implemented in 1D
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Acceptable Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>castro.do_hydro</code></td>
<td>Time-advance the fluid dynamical equations</td>
<td>0 if false, 1 if true</td>
<td>must be set</td>
</tr>
<tr>
<td><code>castro.do_react</code></td>
<td>Include reactions</td>
<td>0 if false, 1 if true</td>
<td>must be set</td>
</tr>
<tr>
<td><code>castro.add_ext_src</code></td>
<td>Include additional user-specified source term</td>
<td>0 if false, 1 if true</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.point_mass</code></td>
<td>Point mass at the center of the star</td>
<td>Real ≥ 0</td>
<td>0.0</td>
</tr>
<tr>
<td><code>castro.do_sponge</code></td>
<td>Call a user-supplied sponging routine after the solution update</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.normalize_species</code></td>
<td>Enforce that $\sum_i X_i = 1$</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.fix_mass_flux</code></td>
<td>Enforce constant mass flux at domain boundary</td>
<td>0 or 1</td>
<td>1</td>
</tr>
<tr>
<td><code>castro.allow_negative_energy</code></td>
<td>Is internal energy allowed to be negative</td>
<td>0 or 1</td>
<td>1</td>
</tr>
<tr>
<td><code>castro.ppm_type</code></td>
<td>Use piecewise linear vs PPM algorithm</td>
<td>0, 1, 2</td>
<td>1</td>
</tr>
<tr>
<td><code>castro.use_colglaz</code></td>
<td>Use the Colella/Glaz algorithm?</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.gamma</code></td>
<td>Sets the value of $\gamma$</td>
<td>Real</td>
<td>0.0</td>
</tr>
<tr>
<td><code>castro.spherical_star</code></td>
<td></td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.show_center_of_mass</code></td>
<td></td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td><code>castro.small_dens</code></td>
<td></td>
<td>Real</td>
<td>-1.e20</td>
</tr>
<tr>
<td><code>castro.small_temp</code></td>
<td></td>
<td>Real</td>
<td>-1.e20</td>
</tr>
<tr>
<td><code>castro.small_pres</code></td>
<td></td>
<td>Real</td>
<td>-1.e20</td>
</tr>
</tbody>
</table>
CHAPTER 4

Units and Constants

4.1 Units and Constants

We currently support only CGS units in CASTRO. All inputs and problem initialization should be specified in CGS. No internal conversions of units occur within the code, so the output must be interpreted appropriately.
### Chapter 4. Units and Constants

<table>
<thead>
<tr>
<th>Location</th>
<th>Variable</th>
<th>CGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs file</td>
<td><code>geometry.prob_lo</code> and <code>geometry.prob_hi</code></td>
<td>cm</td>
</tr>
<tr>
<td>Hydro Initialization</td>
<td>density</td>
<td>g / cm³</td>
</tr>
<tr>
<td>Hydro Initialization</td>
<td>velocities</td>
<td>cm/s</td>
</tr>
<tr>
<td>Hydro Initialization</td>
<td>temperature</td>
<td>K</td>
</tr>
<tr>
<td>Hydro Initialization</td>
<td>energies</td>
<td>erg = g (cm/s)²</td>
</tr>
<tr>
<td>Constants Supplied</td>
<td>gravitational constant ((G))</td>
<td>6.67428e-8 cm (cm/s)² g⁻¹</td>
</tr>
<tr>
<td>Constants Supplied</td>
<td>Avogadro’s number ((n_A))</td>
<td>6.02214129e23 g⁻¹</td>
</tr>
<tr>
<td>Constants Supplied</td>
<td>Boltzmann’s constant ((k_B))</td>
<td>1.3806488e-16 erg / K</td>
</tr>
<tr>
<td>Output</td>
<td>Pressure</td>
<td>g (cm/s)² / cm³</td>
</tr>
<tr>
<td>Output</td>
<td>Time</td>
<td>s</td>
</tr>
</tbody>
</table>
Equations

5.1 Conservation Forms

We begin with the fully compressible equations for the conserved state vector, $\mathbf{U} = (\rho, \rho \mathbf{u}, \rho E, \rho A_k, \rho X_k, \rho Y_k)$:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u}) + S_{\text{ext}, \rho}, \quad (5.1)$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} = -\nabla \cdot (\rho \mathbf{u} \mathbf{u}) - \nabla p + \rho g + S_{\text{ext}, \rho \mathbf{u}}, \quad (5.2)$$

$$\frac{\partial (\rho E)}{\partial t} = -\nabla \cdot (\rho \mathbf{u} E + p \mathbf{u} ) + \rho \mathbf{u} \cdot g - \sum_k \rho q_k \dot{\omega}_k + \nabla \cdot \kappa \nabla T + S_{\text{ext}, \rho E}, \quad (5.3)$$

$$\frac{\partial (\rho A_k)}{\partial t} = -\nabla \cdot (\rho \mathbf{u} A_k) + S_{\text{ext}, \rho A_k}, \quad (5.4)$$

$$\frac{\partial (\rho X_k)}{\partial t} = -\nabla \cdot (\rho \mathbf{u} X_k) + \rho \dot{\omega}_k + S_{\text{ext}, \rho X_k}, \quad (5.5)$$

$$\frac{\partial (\rho Y_k)}{\partial t} = -\nabla \cdot (\rho \mathbf{u} Y_k) + S_{\text{ext}, \rho Y_k}. \quad (5.6)$$

Here $\rho, \mathbf{u}, T, p,$ and $\kappa$ are the density, velocity, temperature, pressure, and thermal conductivity, respectively, and $E = e + \mathbf{u} \cdot \mathbf{u}/2$ is the total energy with $e$ representing the internal energy. In addition, $X_k$ is the abundance of the $k^{\text{th}}$ isotope, with associated production rate, $\dot{\omega}_k$, and energy release, $q_k$. Here $\mathbf{g}$ is the gravitational vector, and $S_{\text{ext}, \rho}, S_{\text{ext}, \rho \mathbf{u}}$, etc., are user-specified source terms. $A_k$ is an advected quantity, i.e., a tracer. We also carry around auxiliary variables, $Y_k$, which have a user-defined evolution equation, but by default are treated as advected quantities.

In the code we also carry around $T$ and $\rho e$ in the conservative state vector even though they are derived from the other conserved quantities. The ordering of the elements within $\mathbf{U}$ is defined (in 3D) by
• URHO: $\rho$

• UMX: $\rho u$

• UMY: $\rho v$

• UMZ: $\rho w$

• UEDEN: $\rho E$

• UEINT: $\rho e$ - this is computed from the other quantities using $\rho e = \rho E - \rho u \cdot u / 2$.

• UTEMP: $T$ - this is computed from the other quantities using the EOS

• UFA: $\rho A_1$, the first advected quantity

• UFS: $\rho X_1$, the first species

• UFX: $\rho Y_1$, the first auxiliary variable

There are $nadv$ advected quantities, which range from UFA: UFA+$nadv$-1. In addition, there are $nspec$ species (defined in the network directory), which range from UFS: UFS+$nspec$-1. Finally, there are $naux$ auxiliary variables, from UFX:UFX+$naux$-1, and $nadv$ advected quantities, which range from UFA: UFA + $nadv$ - 1. The advected quantities have no effect at all on the rest of the solution but can be useful as tracer quantities. The auxiliary variables are passed into the equation of state routines along with the species; An example of an auxiliary variable is the electron fraction, $Ye$, in core collapse simulations.
5.2 Primitive Forms

Here are the primitive forms of the equations for the primitive state vector, $Q = (\rho, u, p, \rho e, A_k, X_k, Y_k)$:

\[
\frac{\partial \rho}{\partial t} = -u \cdot \nabla \rho - \rho \nabla \cdot u + S_{\text{ext,}\rho}, \quad (5.7)
\]

\[
\frac{\partial u}{\partial t} = -u \cdot \nabla u - \frac{1}{\rho} \nabla p + g + \frac{1}{\rho}(S_{\text{ext,}\rho u} - u S_{\text{ext,}\rho}), \quad (5.8)
\]

\[
\frac{\partial p}{\partial t} = -u \cdot \nabla p - \rho c^2 \nabla \cdot u + \left(\frac{\partial p}{\partial \rho}\right)_{e,X} S_{\text{ext,}\rho} \\
+ \frac{1}{\rho} \sum_k \left(\frac{\partial p}{\partial X_k}\right)_{\rho,e,X,j \neq k} (\rho \dot{\omega}_k + S_{\text{ext,}\rho X_k} - X_k S_{\text{ext,}\rho}) \\
+ \frac{1}{\rho} \left(\frac{\partial p}{\partial \rho}\right)_{e,X} \left[-\varepsilon S_{\text{ext,}\rho} - \sum_k \rho q_k \dot{\omega}_k + \nabla \cdot \kappa \nabla T \right. \\
\left. + S_{\text{ext,}\rho E} - u \cdot \left(S_{\text{ext,}\rho u} - \frac{u}{2} S_{\text{ext,}\rho}\right)\right], \quad (5.9)
\]

\[
\frac{\partial (\rho e)}{\partial t} = -u \cdot \nabla (\rho e) - (\rho e + p) \nabla \cdot u - \sum_k \rho q_k \dot{\omega}_k + \nabla \cdot \kappa \nabla T + S_{\text{ext,}\rho E} \\
- u \cdot \left(S_{\text{ext,}\rho u} - \frac{1}{2} S_{\text{ext,}\rho u}\right), \quad (5.10)
\]

\[
\frac{\partial A_k}{\partial t} = -u \cdot \nabla A_k + \frac{1}{\rho}(S_{\text{ext,}\rho A_k} - A_k S_{\text{ext,}\rho}), \quad (5.11)
\]

\[
\frac{\partial X_k}{\partial t} = -u \cdot \nabla X_k + \dot{\omega}_k + \frac{1}{\rho}(S_{\text{ext,}\rho X_k} - X_k S_{\text{ext,}\rho}), \quad (5.12)
\]

\[
\frac{\partial Y_k}{\partial t} = -u \cdot \nabla Y_k + \frac{1}{\rho}(S_{\text{ext,}\rho Y_k} - Y_k S_{\text{ext,}\rho}). \quad (5.13)
\]

In the code we also carry around $T$ in the primitive state vector. All of the primitive variables are derived from the conservative state vector, as described in Section 8.1.1. The ordering of the elements within $Q$ is defined (in 3D) by

- QRHO: $\rho$
- QU: $u$
- QV: $v$
- QW: $w$
- QPRES: $p$
- QREINT: $\rho e$
- QTEMP: $T$
- QFA: $A_1$, the first advected quantity
- QFS: $X_1$, the first species
• QFX: $Y_1$, the first auxiliary variable

The full primitive variable form (without the advected or auxiliary quantities) is

$$\frac{\partial Q}{\partial t} + \sum_d A_d \frac{\partial Q}{\partial x_d} = S_Q. \quad (5.14)$$

For example, in 2D:

$$\begin{bmatrix}
\rho \\
u \\
v \\
p \\
\rho e \\
X_k 
\end{bmatrix}_t + \begin{bmatrix}
\rho & \rho & 0 & 0 & 0 & 0 \\
0 & \rho & 0 & \frac{1}{\rho} & 0 & 0 \\
0 & 0 & \rho & 0 & 0 & 0 \\
0 & \rho e + p & 0 & 0 & u & 0 \\
0 & 0 & 0 & 0 & 0 & u \\
X_k 
\end{bmatrix}
\begin{bmatrix}
\rho & v \\
0 & v \\
0 & \rho c^2 & v \\
\rho e & 0 & p & v \\
0 & 0 & 0 & 0 \\
X_k 
\end{bmatrix}_x + \begin{bmatrix}
\rho \\
v \\
p \\
p e + p & 0 & v & 0 \\
0 & 0 & 0 & 0 \\
\rho e \\
\rho e \\
X_k 
\end{bmatrix}_y = S_Q \quad (5.15)$$

The eigenvalues are:

$$\Lambda(A_x) = \{u - c, u, u, u, u, u + c\}, \quad \Lambda(A_y) = \{v - c, v, v, v, v + c\}. \quad (5.16)$$

The right column eigenvectors are:

$$R(A_x) = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 1 \\
-\frac{c}{\rho} & 0 & 0 & 0 & 0 & \frac{c}{\rho} \\
0 & 0 & 1 & 0 & 0 & 0 \\
\frac{c^2}{\rho} & 0 & 0 & 0 & 0 & \frac{c^2}{\rho} \\
h & 0 & 0 & 1 & h & 0 \\
0 & 0 & 0 & 0 & 1 & 0 
\end{bmatrix}, \quad R(A_y) = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 1 \\
-\frac{c}{c} & 0 & 0 & 0 & 0 & \frac{c}{c} \\
0 & 0 & 1 & 0 & 0 & 0 \\
\frac{c^2}{c} & 0 & 0 & 0 & 0 & \frac{c^2}{c} \\
h & 0 & 0 & 1 & h & 0 \\
0 & 0 & 0 & 0 & 1 & 0 
\end{bmatrix}. \quad (5.17)$$

The left row eigenvectors, normalized so that $R_d \cdot L_d = I$ are:

$$L_x = \begin{bmatrix}
0 & -\frac{\rho}{2c} & 0 & \frac{1}{2c^2} & 0 & 0 \\
1 & 0 & 0 & -\frac{1}{2c^2} & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{h}{c^2} & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & \frac{\rho}{2c} & 0 & \frac{1}{2c^2} & 0 & 0 
\end{bmatrix}, \quad L_y = \begin{bmatrix}
0 & 0 & -\frac{\rho}{2c} & \frac{1}{2c^2} & 0 & 0 \\
0 & 1 & 0 & 0 & -\frac{1}{2c} & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{h}{c^2} & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & \frac{\rho}{2c} & 0 & \frac{1}{2c^2} & 0 & 0 
\end{bmatrix}. \quad (5.18)$$
There are currently four options for how gravity is calculated – these are controlled by setting `gravity.gravity_type`. The options are `ConstantGrav`, `PoissonGrav`, `Monopole Grav`, or `PrescribedGrav`. Note that these are only relevant if `USE_GRAV = TRUE` in the GNUmakefile and `castro.do_grav = 1` in the inputs file. If both of these are set then the user is required to specify the gravity type in the inputs file or the program will abort.

Note that `MonopoleGrav` and `PoissonGrav` is only correct for spherical stars, i.e. in 1D we must have `coord_sys = 2`, in 2D we must have `coord_sys = 1`, and in 3D we only support `coord_sys = 0`.

### 6.1 Types of Approximations

- **ConstantGrav**: Gravity can be defined as constant in direction and magnitude, defined in the inputs file by

  \[
  \text{gravity.const_grav} = -9.8
  \]

  for example, to set the gravity to have magnitude 9.8 in the negative y-direction if in 2D, negative z-direction if in 3-D.

- **PoissonGrav**: The most general case is a self-induced gravitational field,

  \[
  \mathbf{g}(x, t) = \nabla \phi
  \]

  where \( \phi \) is defined by solving

  \[
  \Delta \phi = -4\pi G\rho.
  \]

  (We note that the sign convention used for \( \phi \) here is opposite that traditionally used in astrophysics, but the resulting gravitational acceleration will be the same.)
We only allow **PoissonGrav** in 2D or 3D because in 1D, computing the monopole approximation in spherical coordinates is faster and more accurate than solving the Poisson equation.

In 2D or 3D we either have doubly or triply periodic boundary conditions, or we define boundary conditions for \( \phi \) using the monopole approximation at the coarsest level. (Without this the boundary conditions would be homogeneous Dirichlet which results in loss of sphericity of the gravitational field.) We first compute a 1D radial profile of the average density at the coarsest level, then integrate that to define a 1D radial profile of the gravitational acceleration as below. We then integrate \( g \) to define \( \phi \), starting with \( \phi = 0 \) at the center.

- **MonopoleGrav**:
  
  - In 1D spherical coordinates we compute
    
    \[
g(r) = G \times \frac{\text{Mass}_{\text{enclosed}}}{r^2},
    \]
    
    where \( \text{Mass}_{\text{enclosed}} \) is calculated from the density at the time of the call. For levels above the coarsest level we define the extent of that level’s radial arrays as ranging from the center of the star \( (r = 0) \) to the cell at that level farthest away from the origin. If there are gaps between fine grids in that range then we interpolate the density from a coarser level in order to construct a continuous density profile. We note that the location of values in the density profile and in the gravitational field exactly match the location of data at that level so there is no need to interpolate between points when mapping the 1D radial profile of \( g \) back onto the original grid.

  - In 2D or 3D we compute a 1D radial average of density and use this to compute gravity as a one-dimensional integral, then interpolate the gravity vector back onto the Cartesian grid cells. At the coarsest level we define the extent of the 1D arrays as ranging from the center of the star to the farthest possible point in the grid (plus a few extra cells so that we can fill ghost cell values of gravity). At finer levels we first define a single box that contains all boxes on that fine level, then we interpolate density from coarser levels as needed to fill the value of density at every fine cell in that box. The extent of the radial array is from the center of the star to the nearest cell on one of the faces of the single box. This ensures that all cells at that maximum radius of the array are contained in this box.

We then average the density onto a 1D radial array. We note that there is a mapping from the Cartesian cells to the radial array and back; unlike the 1D case this requires interpolation. We use quadratic interpolation with limiting so that the interpolation does not create new maxima or minima.

The default resolution of the radial arrays at a level is the grid cell spacing at that level, i.e. \( \Delta r = \Delta x \). One optimization we have recently added is that one can define `castro.drdfac` as a number greater than 1 (2 or 4 are recommended) and the spacing of the radial array will then satisfy \( \Delta x / \Delta r = \text{drdfac} \). Individual Cartesian grid cells are subdivided by \( \text{drdfac} \) in each coordinate direction for the purposing of averaging the density, and the integration that creates \( g \) is done at the finer resolution of the new \( \Delta r \).

Note that the center of the star is defined in the subroutine `PROBINIT` and the radius is computed as the distance from that center.
6.2—GR correction

Note: The PrescribedGrav option and text here were contributed by Jan Frederik Engels of University of Gottingen.

With this option, gravity can be defined as a function that is specified by the user. The option is allowed in 2D and 3D. To define the gravity vector, copy `prescribe_grav_2d.f90` from `Src/2d` to your run directory (analogously copy `prescribe_grav_3d.f90` from `Src/3d` if you’re working in 3D). The makefile system will always choose this local copy of the file over the one in another directory. Then define the components of gravity inside a loop over the grid inside the file. If your problem uses a radial gravity in the form \( g(r) \), you can simply adapt `ca.prescribe_grav.gravityprofile`, otherwise you will have to adapt `ca.prescribe_grav`, both are located in `prescribed_grav_2d.90`.

6.2 GR correction

Note: The GR code and text here were contributed by Ken Chen of Univ. of Minnesota.

In the cases of compact objects or very massive stars, the general relativity (GR) effect starts to play a role. First, we consider the hydrostatic equilibrium due to effects of GR then derive GR-correction term for Newtonian gravity. The correction term is applied to the monopole approximation only when `USE_GR = TRUE` is set in the GNUmakefile.

The formulae of GR-correction here are based on [5]. For detailed physics, please refer to [13]. For describing very strong gravitational field, we need to use Einstein field equations

\[
R_{ik} - \frac{1}{2} g_{ik} R = \frac{\kappa}{c^2} T_{ik} , \quad \kappa = \frac{8\pi G}{c^2}, \tag{6.3}
\]

where \( R_{ik} \) is the Ricci tensor, \( g_{ik} \) is the metric tensor, \( R \) is the Riemann curvature, \( c \) is the speed of light and \( G \) is gravitational constant. \( T_{ik} \) is the energy momentum tensor, which for ideal gas has only the non-vanishing components \( T_{00} = \rho c^2 \), \( T_{11} = T_{22} = T_{33} = P \) (contains rest mass and energy density, \( P \) is pressure). We are interested in spherically symmetric mass distribution. Then the line element \( ds \) for given spherical coordinate \((r, \vartheta, \varphi)\) has the general form

\[
ds^2 = e^\nu c^2 dt^2 - e^\lambda dr^2 - r^2(d\vartheta^2 + \sin^2 \vartheta d\varphi), \tag{6.4}
\]

with \( \nu = \nu(r) \), \( \lambda = \lambda(r) \). Now we can put the expression of \( T_{ik} \) and \( ds \) into (6.3), then field equations can be reduced to 3 ordinary differential equations:

\[
\frac{\kappa P}{c^2} = e^{-\lambda} \left( \frac{\nu'}{r} + \frac{1}{r^2} \right) - \frac{1}{r^2}, \tag{6.5}
\]

\[
\frac{\kappa P}{c^2} = \frac{1}{2} e^{-\lambda} \left( \frac{\nu''}{r} + \frac{1}{2} \nu'^2 + \frac{\nu' - \lambda'}{r} - \frac{\nu' \lambda'}{2} \right), \tag{6.6}
\]

\[
\kappa \varrho = e^{-\lambda} \left( \frac{\lambda'}{r} - \frac{1}{r^2} \right) + \frac{1}{r^2}, \tag{6.7}
\]

where primes means the derivatives with respect to \( r \). After multiplying with \( 4\pi r^2 \), (6.7) can be integrated and yields

\[
\kappa m = 4\pi r (1 - e^{-\lambda}), \tag{6.8}
\]
the \( m \) is called “gravitational mass” inside \( r \) defined as

\[
m = \int_0^r 4\pi r^2 \varrho dr.
\]  

(6.9)

For the \( r = R \), \( m \) becomes the mass \( M \) of the star. \( M \) contains not only the rest mass but the whole energy (divided by \( c^2 \)), that includes the internal and gravitational energy. So the \( \varrho = \varrho_0 + U/c^2 \) contains the whole energy density \( U \) and rest-mass density \( \varrho_0 \). Differentiation of \((6.5)\) with respect to \( r \) gives \( P = P'(\lambda, \lambda', \nu, \nu', r) \), where \( \lambda, \lambda', \nu, \nu' \) can be eliminated by \((6.5), (6.6), (6.7)\). Finally we reach Tolman-Oppenheimer-Volkoff (TOV) equation for hydrostatic equilibrium in general relativity:

\[
\frac{dP}{dr} = -\frac{Gm}{r^2} \varrho (1 + \frac{P}{\varrho c^2})(1 + 4\pi \frac{r^3 P}{mc^2})(1 - 2\frac{Gm}{rc^2})^{-1}.
\]  

(6.10)

For Newtonian case \( c^2 \to \infty \), it reverts to usual form

\[
\frac{dP}{dr} = -\frac{Gm}{r^2} \varrho.
\]  

(6.11)

Now we take effective monopole gravity as

\[
\tilde{g} = -\frac{Gm}{r^2} (1 + \frac{P}{\varrho c^2})(1 + 4\pi \frac{r^3 P}{mc^2})(1 - 2\frac{Gm}{rc^2})^{-1}.
\]  

(6.12)

For general situations, we neglect the \( U/c^2 \) and potential energy in \( m \) because they are usually much smaller than \( \varrho_0 \). Only when \( T \) reaches \( 10^{13} \)K \((KT \approx m_p c^2, m_p \) is proton mass) before it really makes a difference. So \((6.12)\) can be expressed as

\[
\tilde{g} = -\frac{GM_{\text{enc}}}{r^2} (1 + \frac{P}{\varrho c^2})(1 + 4\pi \frac{r^3 P}{M_{\text{enc}} c^2})(1 - 2\frac{GM_{\text{enc}}}{rc^2})^{-1},
\]  

(6.13)

where \( M_{\text{enc}} \) is identical to \( \text{Mass}_{\text{enclosed}} \) in the previous section.
Rotation

Currently, CASTRO supports constant, solid-body rotation about a fixed (in space and time) axis in 2D and 3D by transforming the evolution equations to the rotating frame of reference. The rotational frequency (in Hz) is specified by setting `castro.rotational_frequency` in the inputs file. Note that this parameter is only relevant if `USE_ROTATION = TRUE` in the GNUmakefile and `castro.do_rotation = 1` in the inputs file. The rotational frequency specified by `castro.rotational_frequency` is internally converted to an angular frequency for use in the source term equations.

The axis of rotation currently depends on the dimensionality of the problem and the value of `coord_sys`; in all cases, however, the default axis of rotation points from `center`, which is typically defined in a Prob.$(DIM)d.f90 routine, to the typical “vertical direction.” The vertical direction is defined as follows:

- **2D**
  - `coord_sys = 0`, (x,y): out of the (x,y)-plane along the “z”-axis
  - `coord_sys = 1`, (r,z): along the z-axis

- **3D**
  - `coord_sys = 0`, (x,y,z): along the z-axis

To change these defaults, the `omega` vector in the `ca_rotate` routine found in the `Rotate.$(DIM)d.f90` file.

For completeness, we show below a derivation of the source terms that appear in the momentum and total energy evolution equations upon switching to a rotating reference frame.
7.1 Coordinate transformation to rotating frame

Consider an intertial reference frame $C$ and a non-inertial reference frame $\tilde{C}$ whose origins are separated by the vector $l$. The non-inertial frame is rotating about the axis $\omega$ with a constant angular velocity $\omega$; furthermore, we assume the direction of the rotational axis is fixed. Consider a fluid element at the point $P$ whose location is given by $r$ in $C$ and by $\tilde{r}$ in $\tilde{C}$:

$$r = \tilde{r} + l,$$

or in component notation

$$r_i e_i = \tilde{r}_i \tilde{e}_i + l_i e_i,$$  \hspace{1cm} (7.1)

where $e_i$ and $\tilde{e}_i$ are the $i$th unit vectors in the $C$ and $\tilde{C}$ coordinate systems, respectively. The total time rate of change of 7.2 is given by

$$\frac{Dr_i}{Dt} e_i = \frac{D\tilde{r}_i}{Dt} \tilde{e}_i + \frac{D\tilde{e}_i}{Dt} e_i + \frac{Dl_i}{Dt} e_i,$$  \hspace{1cm} (7.2)

where we have used the fact that the unit vectors of the inertial frame $C$ are not moving (or at least can be considered stationary, and the change in $l$ gives the relative motion of the two coordinate systems). By definition, a unit vector can not change its length, and therefore the only change of $\tilde{e}_i$ with time can come from changing direction. This change is carried out by a rotation about the $\omega$ axis, and the tip of the unit vector moves circumferentially, that is

$$\frac{D\tilde{e}_i}{Dt} = \omega \times \tilde{e}_i.$$  \hspace{1cm} (7.3)

Plugging 7.4 into 7.3 and switching back to vector notation, we have

$$\frac{Dr}{Dt} = \frac{D\tilde{r}}{Dt} + \omega \times \tilde{r} + \frac{Dl}{Dt}.$$  \hspace{1cm} (7.4)

The left hand side of 7.5 is interpreted as the velocity of the fluid element as seen in the inertial frame; the first term on the right hand side is the velocity of the fluid element as seen by a stationary observer in the rotating frame $\tilde{C}$. The second and third terms on the right hand side of 7.5 describe the additional velocity due to rotation and translation of the frame $\tilde{C}$ as seen in $C$. In other words,

$$v = \tilde{v} + \omega \times \tilde{r} + v_l,$$  \hspace{1cm} (7.5)

where we use $v_l$ to represent the translational velocity.

Similarly, by taking a second time derivative of 7.6 we have

$$\frac{Dv}{Dt} = \frac{D\tilde{v}}{Dt} + 2\omega \times \tilde{v} + \omega \times [\omega \times \tilde{r}] + \frac{Dv_l}{Dt}.$$  \hspace{1cm} (7.6)

Henceforth we will assume the two coordinate systems are not translating relative to one another, $v_l = 0$. It is also worth mentioning that derivatives with respect to spatial coordinates do not involve additional terms due to rotation, i.e. $\nabla \cdot v = \nabla \cdot \tilde{v}$. Because of this, the continuity equation remains unchanged in the rotating frame:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \tilde{v}),$$  \hspace{1cm} (7.7)

or

$$\frac{D\rho}{Dt} = -\rho \nabla \cdot \tilde{v}.$$  \hspace{1cm} (7.8)
7.2 Momentum equation in rotating frame

The usual momentum equation applies in an inertial frame:

$$\frac{D(\rho v)}{Dt} = -\rho v \cdot \nabla v - \nabla p + \rho g.$$  \hspace{1cm} (7.10)

Using the continuity equation, \(7.9\) and substituting for the terms in the rotating frame from \(7.7\), we have from \(7.10\)

$$\rho \left( \frac{D\tilde{v}}{Dt} + 2\omega \times \tilde{v} + \omega \times [\omega \times \tilde{r}] \right) - \rho v \cdot \nabla v = -\rho v \cdot \nabla v - \nabla p + \rho g$$

$$\rho \left( \frac{\partial \tilde{v}}{\partial t} + \tilde{v} \cdot \nabla \tilde{v} \right) = -\nabla p + \rho g - 2\rho \omega \times \tilde{v} - \rho \omega \times [\omega \times \tilde{r}]$$

$$\frac{\partial (\rho \tilde{v})}{\partial t} = -\nabla \cdot (\rho \tilde{v} \tilde{v}) + \nabla p + \rho g - 2\rho \omega \times \tilde{v}$$

or

$$\frac{D(\rho \tilde{v})}{Dt} = -\rho \tilde{v} \cdot \nabla \tilde{v} + \nabla p + \rho g - 2\rho \omega \times \tilde{v} - \rho \omega \times [\omega \times \tilde{r}].$$  \hspace{1cm} (7.11)

7.3 Energy equations in rotating frame

From \(7.12\) we have the velocity evolution equation in a rotating frame

$$\frac{D\tilde{v}}{Dt} = -\frac{1}{\rho} \nabla p + g - 2\omega \times \tilde{v} - \omega \times [\omega \times \tilde{r}].$$  \hspace{1cm} (7.13)

The kinetic energy equation can be obtained from \(7.13\) by multiplying by \(\tilde{v}\):

$$\rho \tilde{v} \cdot \frac{D\tilde{v}}{Dt} = -\tilde{v} \cdot \nabla p + \rho \tilde{v} \cdot g - 2\rho \tilde{v} \cdot [\omega \times \tilde{v}] - \rho \tilde{v} \cdot [\omega \times [\omega \times \tilde{r}]]$$

$$\frac{1}{2} \frac{D(\rho \tilde{v} \cdot \tilde{v})}{Dt} = -\frac{1}{2} \tilde{v} \cdot \nabla p + \rho \tilde{v} \cdot g - \rho \tilde{v} \cdot [(\omega \times \tilde{r}) \omega - \rho \omega^2 \tilde{r}]$$

The internal energy is simply advected, and, from the first law of thermodynamics, can change due to \(pdV\) work:

$$\frac{D(p\tilde{c})}{Dt} = -(p + \tilde{c}) \nabla \cdot \tilde{v}.$$  \hspace{1cm} (7.15)

Combining \(7.14\) and \(7.15\) we can get the evolution of the total specific energy in the rotating frame, \(\rho \tilde{E} = p + \frac{1}{2} \rho \tilde{v} \cdot \tilde{v}\):

$$\frac{D(p\tilde{c})}{Dt} + \frac{1}{2} \frac{D(\rho \tilde{v} \cdot \tilde{v})}{Dt} = -\left( p + \frac{1}{2} \rho \tilde{v} \cdot \tilde{v} \right) \nabla \cdot \tilde{v} - \tilde{v} \cdot \nabla p + \rho \tilde{v} \cdot g - \rho \tilde{v} \cdot [(\omega \times \tilde{r}) \omega - \rho \omega^2 \tilde{r}]$$

$$\frac{D(\rho \tilde{E})}{Dt} = -\rho \tilde{E} \nabla \cdot \tilde{v} - \tilde{v} \cdot (\rho \tilde{v}) + \rho \tilde{v} \cdot g - \rho \tilde{v} \cdot [(\omega \times \tilde{r}) \omega - \rho \omega^2 \tilde{r}]$$  \hspace{1cm} (7.16)

or

$$\frac{\partial (\rho \tilde{E})}{\partial t} = -\nabla \cdot (\rho \tilde{E} \tilde{v} + \rho \tilde{v}) + \rho \tilde{v} \cdot g - \rho \tilde{v} \cdot [(\omega \times \tilde{r}) \omega - \rho \omega^2 \tilde{r}].$$  \hspace{1cm} (7.17)
Chapter 7. Rotation

7.4 Switching to the rotating reference frame

If we choose to be a stationary observer in the rotating reference frame, we can drop all of the tildes, which indicated terms in the non-inertial frame \( \tilde{C} \). Doing so, and making sure we account for the offset, \( l \), between the two coordinate systems, we obtain the following equations for hydrodynamics in a rotating frame of reference:

\[
\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v}) \quad (7.18)
\]
\[
\frac{\partial (\rho \mathbf{v})}{\partial t} = -\nabla \cdot (\rho \mathbf{v} \mathbf{v}) - \nabla p + \rho \mathbf{g} - 2\rho \omega \times \mathbf{v} - \rho (\omega \cdot \mathbf{r}) \omega + \rho \omega^2 \mathbf{r} \quad (7.19)
\]
\[
\frac{\partial (\rho E)}{\partial t} = -\nabla \cdot (\rho E \mathbf{v} + p \mathbf{v}) + \rho \mathbf{v} \cdot \mathbf{g} - \rho (\omega \cdot \mathbf{r}) (\omega \cdot \mathbf{v}) + \rho \omega^2 (\mathbf{v} \cdot \mathbf{r}) . \quad (7.20)
\]

Comparing the above equations with the fully compressible equations in 5.6 we see that the Coriolis \((-2\rho \omega \times \mathbf{v})\) and centrifugal \((-\rho \omega \times [\omega \times \mathbf{r}] = -\rho (\omega \cdot \mathbf{r}) \omega + \rho \omega^2 \mathbf{r})\) terms can be swept into the “external source term” \( S_{\text{ext},\rho u} \). The corresponding terms in the energy evolution equation above can be swept into the external source term \( S_{\text{ext},\rho E} \) of the compressible equations.
CHAPTER 8

Single-Level Flow Chart

Our equations look like:

$$\frac{\partial U}{\partial t} = \nabla \cdot F + S_{\text{react}} + S,$$

(8.1)

where $F$ is the flux vector, $S_{\text{react}}$ are the reaction source terms, and $S$ are the non-reaction source terms, which includes any user-defined external sources, $S_{\text{ext}}$. We use Strang splitting to discretize the advection-reaction equations. In summary, for each time step, we update the conservative variables, $U$, by reacting for half a time step, advecting for a full time step (ignoring the reaction terms), and reacting for half a time step. In summary,

$$U^n = U^n + \frac{\Delta t}{2} S^n_{\text{react}},$$

(8.2)

$$U^{n+1} = U^n - \Delta t \nabla \cdot F^{n+\frac{1}{2}} + \Delta t \frac{S^n + S^{n+1}}{2},$$

(8.3)

$$U^{n+1} = U^{n+1} + \frac{\Delta t}{2} S^{n+1}_{\text{react}},$$

(8.4)

The construction of $F$ is purely explicit, and based on an unsplit second-order Godunov method. We predict the standard primitive variables, as well as $\rho e$, at time-centered edges and use an approximate Riemann solver construct fluxes. At the beginning of the time step, we assume that $U$ and $\phi$ are defined consistently, i.e., $\rho^n$ and $\phi^n$ satisfy equation (6.2).

CASTRO also supports radiation (Chapter ??) and level sets (Chapter 9). We omit the details in this section. Here is the single-level algorithm:

**Step 1:** React $\Delta t/2$. 

35
Update the solution due to the effect of reactions over half a time step.

\[
(pE)^n = (pE)^n - \frac{\Delta t}{2} \sum_k (pq_k \dot{\omega}_k)^n, \tag{8.5}
\]

\[
(pX_k)^n = (pX_k)^n + \frac{\Delta t}{2} (\rho \dot{\omega}_k)^n. \tag{8.6}
\]

**Step 2: Solve for gravity.**

Solve for gravity using:

\[
g^n = \nabla \phi^n, \quad \Delta \phi^n = -4\pi G\rho^n, \tag{8.7}
\]

or use one of the simpler gravity types.

**Step 3: Compute explicit source terms.**

We now compute explicit source terms for each variable in \(Q\) and \(U\). The primitive variable source terms will be used to construct time-centered fluxes. The conserved variable source will be used to advance the solution. We neglect reaction source terms since they are accounted for in **Steps 1** and **6**. The source terms are:

\[
S^n_Q = \begin{pmatrix}
S_\rho \\
S_u \\
S_{\rho E} \\
S_{A_k} \\
S_{X_k} \\
S_{Y_k}
\end{pmatrix}^n = \begin{pmatrix}
S_{\text{ext},\rho} \\
\rho g + \frac{1}{\rho} S_{\text{ext},\rho u} \\
\frac{1}{\rho} \frac{\partial g}{\partial \rho} S_{\text{ext},\rho E} + \frac{\partial g}{\partial p} S_{\text{ext},\rho p} \\
\nabla \cdot \kappa \nabla T + S_{\text{ext},\rho E} \\
\frac{1}{\rho} S_{\text{ext},\rho A_k} \\
\frac{1}{\rho} S_{\text{ext},\rho X_k} \\
\frac{1}{\rho} S_{\text{ext},\rho Y_k}
\end{pmatrix}^n, \tag{8.8}
\]

\[
S^n_U = \begin{pmatrix}
S_{\rho u} \\
S_{\rho E} \\
S_{\rho A_k} \\
S_{\rho X_k} \\
S_{\rho Y_k}
\end{pmatrix}^n = \begin{pmatrix}
\rho g + S_{\text{ext},\rho u} \\
\rho u \cdot g + \nabla \cdot \kappa \nabla T + S_{\text{ext},\rho E} \\
S_{\text{ext},\rho A_k} \\
S_{\text{ext},\rho X_k} \\
S_{\text{ext},\rho Y_k}
\end{pmatrix}^n. \tag{8.9}
\]

**Step 3: Advect \(\Delta t\).**

The goal is to advance

\[
U^{n+1} = U^n - \Delta t \nabla \cdot F^{n+\frac{1}{2}} + \Delta t S^n. \tag{8.10}
\]

neglecting reaction terms. Note that since the source term is not time centered, this is not a second-order method. After the advective update, we correct the solution, effectively time-centering the source term. The advection step is complicated, and more detail is given in Section 8.1. Here is the summarized version:

1. Compute primitive variables.
2. Predict primitive variables to time-centered edges.
Step 4: Solve for updated gravity.

Solve for gravity using:
\[ g^{n+1} = \nabla \phi^{n+1}; \quad \Delta \phi^{n+1} = -4\pi G \rho^{n+1}, \quad (8.11) \]
or use one of the simpler gravity types.

Step 5: Correct solution with time-centered source terms.

We need to correct the solution by effectively time-centering the source terms. These corrections are to be performed sequentially since new source term evaluations may depend on previous corrections.

First, we correct the solution with the updated gravity:
\[ (\rho u)^{n+1} = (\rho u)^{n+1} + \frac{\Delta t}{2} \left[ (\rho g)^{n+1} - (\rho g)^n \right], \quad (8.12) \]
\[ (\rho E)^{n+1} = (\rho E)^{n+1} + \frac{\Delta t}{2} \left[ (\rho u \cdot g)^{n+1} - (\rho u \cdot g)^n \right]. \quad (8.13) \]

Next, we correct U with updated external sources. For example, for the momentum, we correct using
\[ (\rho u)^{n+1} = (\rho u)^{n+1} + \frac{\Delta t}{2} \left( S_{\text{ext},\rho u}^{n+1} - S_{\text{ext},\rho u}^n \right). \quad (8.14) \]
We correct \( \rho E, \rho A_k, \rho X_k, \) and \( \rho Y_k \) in an analogous manner.

Finally, we correct the solution with updated thermal diffusion using
\[ (\rho E)^{n+1} = (\rho E)^{n+1} + \frac{\Delta t}{2} (\nabla \cdot \kappa \nabla T^{n+1} - \nabla \cdot \kappa \nabla T^n). \quad (8.15) \]

Step 6: React \( \Delta t/2. \)

Update the solution due to the effect of reactions over half a time step.
\[ (\rho E)^{n+1} = (\rho E)^{n+1} - \frac{\Delta t}{2} \sum_k (\rho q_k \omega_k)^{n+1}, \quad (8.16) \]
\[ (\rho X_k)^{n+1} = (\rho X_k)^{n+1} + \frac{\Delta t}{2} (\rho \omega_k)^{n+1}. \quad (8.17) \]

Step 7: Modify auxiliary variables.

This is problem-dependent. By default we treat the auxiliary variables as advected quantities, so no additional steps are required.

Thus concludes the single-level algorithm description.

8.0.1 Castro::advance()

if (doReact)
    strangChem()
end if
if (doGrav)
    define oldGravityVector
end if
if (Diffusion)
    getOldDiffusionTerm()
end if
if (addExtSource)
    getSource() at old time
end if
AdvanceSolution()
if (doGrav)
    define newGravityVector
    correct solution due to new gravity
end if
if (addExtSource)
    getSource() at new time
    correct solution due to new source
end if
if (Diffusion)
    getNewDiffusionTerm()
    correct solution due to new diffusion term
    computeTemp()
end if
if (doReact)
    strangChem()
end if
if (advanceAux)
    advanceAux()
end if
if (LevelSet)
    advanceLevelSet()
8.1 Advection Step

There are four major steps in the advective update, detailed below.

8.1.1 Compute Primitive Variables

We compute the primitive variables from the conserved variables.

- $\rho, \rho e$ - directly copy these from the conserved state vector
- $u, A_k, X_k, Y_k$ - copy these from the conserved state vector, dividing by $\rho$
- $p, T$ - use the EOS. First, if $\text{castro.allow\_negative\_energy} = 0$ (it defaults to 1) and $e < 0$, we do the following:
  
  1. Use the EOS to set $e = e(\rho, T_{\text{small}}, X_k)$.
  2. If $e < 0$, abort the program with an error message.

Now, use the EOS to compute $p, T = p, T(\rho, e, X_k)$.

We also compute the flattening coefficient, $\chi \in [0, 1]$, used in the edge state prediction to further limit slopes near strong shocks. We use the same flattening procedure described in the the FLASH paper. A flattening coefficient of 1 indicates that no additional limiting takes place; a flattening coefficient of 0 means we effectively drop order to a first-order Godunov scheme (this convention is opposite of that used in the FLASH paper). For each cell, we compute the flattening coefficient for each spatial direction, and choose the minimum value over all directions. As an example, to compute the flattening for the $x$-direction, here are the steps:

1. Define $\zeta$
   
   $$\zeta_i = \frac{p_{i+1} - p_{i-1}}{\max(p_{\text{small}}, |p_{i+2} - p_{i-2}|)}.$$
   
2. Define $\tilde{\chi}$
   
   $$\tilde{\chi}_i = \min \{1, \max[0, a(\zeta_i - b)]\},$$

   where $a = 10$ and $b = 0.75$ are tunable parameters. We are essentially setting $\tilde{\chi}_i = a(\zeta_i - b)$, and then constraining $\tilde{\chi}_i$ to lie in the range $[0, 1]$. Then, if either $u_{i+1} - u_{i-1} < 0$ or

   $$\frac{p_{i+1} - p_{i-1}}{\min(p_{i+1}, p_{i-1})} \leq c,$$

   where $c = 1/3$ is a tunable parameter, then set $\tilde{\chi}_i = 0$.

3. Define $\chi$
   
   $$\chi_i = \begin{cases} 
   1 - \max(\tilde{\chi}_i, \tilde{\chi}_{i-1}) & p_{i+1} - p_{i-1} > 0 \\
   1 - \max(\tilde{\chi}_i, \tilde{\chi}_{i+1}) & \text{otherwise} 
   \end{cases}.$$

end if
8.1.2 Edge State Prediction

We wish to compute a left and right state of primitive variables at each edge to be used as inputs to the Riemann problem. We use a version of the Colella and Sekora 2009 PPM algorithm, which has been further modified to eliminate sensitivity due to roundoff error (modifications via personal communication with Colella). Note that CASTRO also has options for the original PPM algorithm of Colella and Woodward 1984, and piecewise-linear algorithm described in Saltzman 1994. We also use characteristic tracing with corner coupling in 3D, as described in Miller and Colella 2002. We give full details of the PPM algorithm, as it has not appeared before in the literature, and summarize the developments from Miller and Colella 2002.

The PPM algorithm is used to compute time-centered edge states by extrapolating the base-time data in space and time. The edge states are dual-valued, i.e., at each face, there is a left state and a right state estimate. The spatial extrapolation is one-dimensional, i.e., transverse derivatives are ignored. We also use a flattening procedure to further limit the edge state values. The Miller and Colella 2002 algorithm, which we describe later, incorporates the transverse terms, and also describes the modifications required for equations with additional characteristics besides the fluid velocity. There are four steps to compute these dual-valued edge states (here, we use \( s \) to denote an arbitrary scalar from \( Q \), and we write the equations in 1D, for simplicity):

- **Step 1**: Compute \( s_{i,+} \) and \( s_{i,-} \), which are spatial interpolations of \( s \) to the hi and lo side of the face with special limiters, respectively. Begin by interpolating \( s \) to edges using a 4th-order interpolation in space:

\[
\frac{s_{i+\frac{1}{2}}}{2} = \frac{7}{12} (s_{i+1} + s_i) - \frac{1}{12} (s_{i+2} + s_{i-1}).
\]  

(8.22)

Then, if \( (s_{i+\frac{1}{2}} - s_i)(s_{i+1} - s_{i+\frac{1}{2}}) < 0 \), we limit \( s_{i+\frac{1}{2}} \) a nonlinear combination of approximations to the second derivative. The steps are as follows:

1. Define:

\[
(D^2s)_{i+\frac{1}{2}} = 3 (s_i - 2s_{i+\frac{1}{2}} + s_{i+1})
\]  

(8.23)

\[
(D^2s)_{i+\frac{1}{2},L} = s_{i-1} - 2s_i + s_{i+1}
\]  

(8.24)

\[
(D^2s)_{i+\frac{1}{2},R} = s_i - 2s_{i+1} + s_{i+2}
\]  

(8.25)

2. Define

\[
s = \text{sign} \left[ (D^2s)_{i+\frac{1}{2}} \right],
\]  

(8.26)

\[
(D^2s)_{i+\frac{1}{2},\text{lim}} = s \max \left\{ \min \left[ C s \left| (D^2s)_{i+\frac{1}{2},L} \right|, C s \left| (D^2s)_{i+\frac{1}{2},R} \right|, s \left| (D^2s)_{i+\frac{1}{2}} \right| \right], 0 \right\}
\]  

(8.27)

where \( C = 1.25 \) as used in Colella and Sekora 2009. The limited value of \( s_{i+\frac{1}{2}} \) is

\[
s_{i+\frac{1}{2}} = \frac{1}{2} (s_i + s_{i+1}) - \frac{1}{6} (D^2s)_{i+\frac{1}{2},\text{lim}}.
\]  

(8.28)

Now we implement an updated implementation of the Colella and Sekora 2009 algorithm which eliminates sensitivity to roundoff. First we need to detect whether a particular cell corresponds to an “extremum”. There are two tests.
8.1—Advection Step

For the first test, define

\[
\alpha_{i,\pm} = s_{i\pm\frac{1}{2}} - s_i. \tag{8.29}
\]

If \( \alpha_{i,+} \alpha_{i,-} \geq 0 \), then we are at an extremum.

We only apply the second test if either \(|\alpha_{i,\pm}| > 2|\alpha_{i,\mp}|\). If so, we define:

\[
(Ds)_{i,\text{face},-} = s_{i-\frac{1}{2}} - s_{i-\frac{3}{2}} \tag{8.30}
\]

\[
(Ds)_{i,\text{face},+} = s_{i+\frac{1}{2}} - s_{i-\frac{1}{2}} \tag{8.31}
\]

\[
(Ds)_{i,\text{face,min}} = \min[|(Ds)_{i,\text{face},-}|, |(Ds)_{i,\text{face},+}|]. \tag{8.32}
\]

\[
(Ds)_{i,\text{cc},-} = s_i - s_{i-1} \tag{8.33}
\]

\[
(Ds)_{i,\text{cc},+} = s_{i+1} - s_i \tag{8.34}
\]

\[
(Ds)_{i,\text{cc,min}} = \min[|(Ds)_{i,\text{cc},-}|, |(Ds)_{i,\text{cc},+}|]. \tag{8.35}
\]

If \((Ds)_{i,\text{face,min}} \geq (Ds)_{i,\text{cc,min}}\), set \((Ds)_{i,\pm} = (Ds)_{i,\text{face},\pm}\). Otherwise, set \((Ds)_{i,\pm} = (Ds)_{i,\text{cc},\pm}\). Finally, we are at an extremum if \((Ds)_{i,+}(Ds)_{i,-} \leq 0\).

Thus concludes the extremum tests. The remaining limiters depend on whether we are at an extremum.

If we are at an extremum, we modify \(\alpha_{i,\pm}\). First, we define

\[
(D^2s)_{i} = 6(\alpha_{i,+} + \alpha_{i,-}) \tag{8.36}
\]

\[
(D^2s)_{iL} = s_{i-2} - 2s_{i-1} + s_i \tag{8.37}
\]

\[
(D^2s)_{iR} = s_i - 2s_{i+1} + s_{i+2} \tag{8.38}
\]

\[
(D^2s)_{iC} = s_{i-1} - 2s_i + s_{i+1} \tag{8.39}
\]

Then, define

\[
s = \text{sign}\left[\left(D^2s\right)_i\right], \tag{8.40}
\]

\[
(D^2s)_{i,\text{lim}} = \max\left\{\min\left[s(D^2s)_i, C\left|D^2s\right|_{iL}, C\left|D^2s\right|_{iR}, C\left|D^2s\right|_{iC}\right], 0\right\}. \tag{8.41}
\]

Then,

\[
\alpha_{i,\pm} = \frac{\alpha_{i,\pm}(D^2s)_{i,\text{lim}}}{\max\left[(D^2s)_i, 1 \times 10^{-10}\right]} \tag{8.42}
\]

If we are not at an extremum and \(|\alpha_{i,\pm}| > 2|\alpha_{i,\mp}|\), then define

\[
s = \text{sign}(\alpha_{i,\mp}) \tag{8.43}
\]

\[
\delta I_{\text{ext}} = \frac{-\alpha_{i,\pm}^2}{4(\alpha_{j,+} + \alpha_{j,-})}, \tag{8.44}
\]

\[
\delta s = s_{i+1} - s_i. \tag{8.45}
\]

If \(\delta I_{\text{ext}} \geq \delta s\), then we perform the following test. If \(\delta s - \alpha_{i,\mp} \geq 1 \times 10^{-10}\), then

\[
\alpha_{i,\pm} = -2\delta s - 2s \left[\left(\delta s\right)^2 - \delta s \alpha_{i,\mp}\right]^{1/2} \tag{8.46}
\]

otherwise,

\[
\alpha_{i,\pm} = -2\alpha_{i,\mp} \tag{8.47}
\]
Finally, \( s_{i, \pm} = s_i + \alpha_{i, \pm} \).

- **Step 2:** Construct a quadratic profile using \( s_{i,-}, s_i, \) and \( s_{i,+} \).

\[
s_i^f(x) = s_{i,-} + \xi [s_{i,+} - s_{i,-} + s_{6,i}(1 - \xi)] ,
\]

\[
s_6 = 6s_i - 3(s_{i,-} + s_{i,+}) ,
\]

\[
\xi = \frac{x - ih}{h} , \quad 0 \leq \xi \leq 1 .
\]

- **Step 3:** Integrate quadratic profiles. We are essentially computing the average value swept out by the quadratic profile across the face assuming the profile is moving at a speed \( \lambda_k \).

Define the following integrals, where \( \sigma_k = |\lambda_k| \Delta t/h \):

\[
I_{i,+}(\sigma_k) = \frac{1}{\sigma_k h} \int_{(i+\frac{1}{2})h - \sigma_k h}^{(i+\frac{1}{2})h} s_i^f(x) \, dx
\]

\[
I_{i,-}(\sigma_k) = \frac{1}{\sigma_k h} \int_{(i-\frac{1}{2})h + \sigma_k h}^{(i-\frac{1}{2})h} s_i^f(x) \, dx
\]

Plugging in (8.48) gives:

\[
I_{i,+}(\sigma_k) = s_{i,+} - \frac{\sigma_k}{2} \left[ s_{i,+} - s_{i,-} - \left( 1 - \frac{2}{3} \sigma_k \right) s_{6,i} \right] .
\]

\[
I_{i,-}(\sigma_k) = s_{i,-} + \frac{\sigma_k}{2} \left[ s_{i,+} - s_{i,-} + \left( 1 - \frac{2}{3} \sigma_k \right) s_{6,i} \right] .
\]

- **Step 4:** Obtain 1D edge states by performing a 1D extrapolation to get left and right edge states. Note that we include an explicit source term contribution.

\[
s_{L,i,\pm} = s_i - \chi_i \sum_{k: \lambda_k \geq 0} l_k \cdot [s_i - I_{i,+}(\sigma_k)] r_k + \frac{\Delta t}{2} S_{i}^n .
\]

\[
s_{R,i,\pm} = s_i - \chi_i \sum_{k: \lambda_k < 0} l_k \cdot [s_i - I_{i,-}(\sigma_k)] r_k + \frac{\Delta t}{2} S_{i}^n .
\]

Here, \( r_k \) is the \( k \)th right column eigenvector of \( R(A_d) \) and \( l_k \) is the \( k \)th left row eigenvector of \( L(A_d) \). The flattening coefficient is \( \chi_i \).

In order to add the transverse terms in an spatial operator unsplit framework, the details follow exactly as given in Section 4.2.1 in Miller and Colella 2002, except for the details of the Riemann solver, which are given below.

### 8.1.3 Riemann Problem

Inputs from the edge state prediction are \( \rho_{L/R}, u_{L/R}, v_{L/R}, p_{L/R}, \) and \( (pe)_{L/R} \) (\( v \) represents all of the transverse velocity components). We also compute \( \gamma \) at cell centers and copy these to edges directly to get the left and right states, \( \gamma_{L/R} \). We also define \( c_{\text{avg}} \) as a face-centered value that is
the average of the neighboring cell-centered values of \( c \). We have also computed \( \rho_{\text{small}}, p_{\text{small}}, \) and \( c_{\text{small}} \) using cell-centered data.

Here are the steps. First, define \((\rho c)_{\text{small}} = \rho_{\text{small}} c_{\text{small}}\). Then, define:

\[
(\rho c)_{L/R} = \max \left[ (\rho c)_{\text{small}}, \left| \gamma_{L/R} \right| p_{L/R}, \rho_{L/R} \right].
\]  

(8.57)

Define star states:

\[
p^* = \max \left[ p_{\text{small}}, \frac{[(\rho c)_{LR} + (\rho c)_{RL} + (\rho c)_L(\rho c)_R(u_L - u_R)]}{(\rho c)_L + (\rho c)_R} \right],
\]

(8.58)

\[
u^* = \frac{[(\rho c)_L u_L + (\rho c)_R u_R] + (p_L - p_R)}{(\rho c)_L + (\rho c)_R}.
\]

(8.59)

If \( u^* \geq 0 \) then define \( \rho_0, u_0, p_0, (\rho e)_0 \) and \( \gamma_0 \) to be the left state. Otherwise, define them to be the right state. Then, set

\[
\rho_0 = \max(\rho_{\text{small}}, \rho_0),
\]

(8.60)

and define

\[
c_0 = \max \left( c_{\text{small}}, \sqrt{\frac{\gamma_0 p_0}{\rho_0}} \right),
\]

(8.61)

\[
p^* = \rho_0 + \frac{p^* - p_0}{c_0^2},
\]

(8.62)

\[
(\rho e)^* = (\rho e)_0 + \frac{(\rho e)_0 + p_0}{\rho_0 c_0^2},
\]

(8.63)

\[
c^* = \max \left( c_{\text{small}}, \sqrt{\frac{\gamma_0 p^*}{\rho^*}} \right)
\]

(8.64)

Then,

\[
c_{\text{out}} = c_0 - \text{sign}(u^*) u_0,
\]

(8.65)

\[
c_{\text{in}} = c^* - \text{sign}(u^*) u^*,
\]

(8.66)

\[
c_{\text{shock}} = \frac{c_{\text{in}} + c_{\text{out}}}{2}.
\]

(8.67)

If \( p^* - p_0 \geq 0 \), then \( c_{\text{in}} = c_{\text{out}} = c_{\text{shock}} \). Then, if \( c_{\text{out}} = c_{\text{in}} \), we define \( c_{\text{temp}} = c_{\text{avg}} \). Otherwise, \( c_{\text{temp}} = c_{\text{out}} - c_{\text{in}} \). We define the fraction

\[
f = \frac{1}{2} \left[ 1 + \frac{c_{\text{out}} + c_{\text{in}}}{c_{\text{temp}}} \right],
\]

(8.68)

and constrain \( f \) to lie in the range \( f \in [0, 1] \).

To get the final “Godunov” state, for the transverse velocity, we upwind based on \( u^* \).

\[
v_{\text{gdnv}} = \begin{cases} v_L, & u^* \geq 0 \\ v_R, & \text{otherwise} \end{cases}
\]

(8.69)
Then, define
\[
\begin{align*}
\rho_{\text{gdnv}} &= f \rho^* + (1 - f) \rho_0, \\
u_{\text{gdnv}} &= f u^* + (1 - f) u_0, \\
p_{\text{gdnv}} &= f p^* + (1 - f) p_0, \\
(\rho e)_{\text{gdnv}} &= f (\rho e)^* + (1 - f) (\rho e)_0.
\end{align*}
\] (8.70)
(8.71)
(8.72)
(8.73)

Finally, if \(c_{\text{out}} < 0\), set \(\rho_{\text{gdnv}} = \rho_0, u_{\text{gdnv}} = u_0, p_{\text{gdnv}} = p_0, \) and \((\rho e)_{\text{gdnv}} = (\rho e)_0\). If \(c_{\text{in}} \geq 0\), set \(\rho_{\text{gdnv}} = \rho^*, u_{\text{gdnv}} = u^*, p_{\text{gdnv}} = p^*, \) and \((\rho e)_{\text{gdnv}} = (\rho e)^*\).

### 8.1.4 Compute Fluxes and Update

Compute the fluxes as a function of the primitive variables, and then advance the solution:
\[
U^{n+1} = U^n - \Delta t \nabla \cdot F^{n+\frac{1}{2}} + \Delta t S^n.
\] (8.74)

Again, note that since the source term is not time centered, this is not a second-order method. After the advective update, we correct the solution, effectively time-centering the source term.
CHAPTER 9

Level Sets

9.1 Introduction

CASTRO has a level set package. The level sets can track a moving interface, but currently do not feed back into the solution. We have a scalar field \( \phi \) which we advect using the equation,

\[
\frac{\partial \phi}{\partial t} + \mathbf{U} \cdot \nabla \phi + (\kappa_a - \kappa_b\kappa)|\nabla \phi| = 0,
\]

where \( \kappa \) is the curvature,

\[
\kappa = \nabla \cdot \left( \frac{\nabla \phi}{|\nabla \phi|} \right).
\]

Typically, the “zero-levelset”, i.e., locations where \( \phi = 0 \), represents some interface we care about. Mathematically, \( \phi \) can be any scalar field, but we initialize \( \phi \) to be a signed distance function, such that \( \phi \) represents the physical distance from the zero-levelset. The sign of \( \phi \) indicates which side of the interface you’re on, and the sign convention can be different from example to example. We will use the convection in [7], where \( \phi < 0 \) in the unburnt region. By choosing \( \phi \) to be a signed distance function, \( |\nabla \phi| = 1 \).

In practice, since we only care about tracking an interface, we only store and update \( \phi \) in a narrow band of cells, aptly named the narrowband. Using a narrowband lowers computational expense, since we only advance \( \phi \) in cells within the narrowband, and when initializing or reinitializing \( \phi \) (see Section 9.2), we only compute \( \phi \) within the narrowband. For multilevel problems, the finest level must have enough cells to contain the entire narrowband. You may have to write special tagging conditions on \( \phi \) to make this happen. Outside of the narrowband, we typically set \( \phi \) to some arbitrarily large positive or negative number (depending on which side of the interface you’re on). In practice, we typically use \( \phi = \pm 1 \times 10^{30} \), but for these notes we refer to these values as \( \phi = \pm \infty \).
9.1.1 An Example

The CASTRO/Exec/Sedov_LS/ directory contains a series of Sedov blast wave problems that use a level set representation to track some interface. Note that the makefile includes “USE_LEVELSET=TRUE”, and the executable has a “.LevelSet” string. Any of the five inputs files (3 in 2D, 2 in 3D) work just fine, except for regions near non-periodic boundary conditions. The “probin” files are where you set the values of $\kappa_a$, $\kappa_b$, and some other parameters which will be described shortly.

9.2 Terminology

At the beginning of a simulation, we initialize $\phi$ within the narrowband using a user-defined description of the initial interface. We also label each cell that is within the narrowband and greater than a specified distance from the interface as a “mine”. As the simulation progresses, we need to “reinitialize” $\phi$ when the interface crosses a mine, which is detected by monitoring when the sign of $\phi$ changes in a mine. During the reinitialization step, we recompute which cells lie in the narrowband, recompute $\phi$ for all cells within the narrowband, redefine which cells are mines, and set $\phi = \pm \infty$ for all cells outside of the narrowband. For adaptive problems, we also reinitialize after regridding.

Note that the level set code uses the convention that the cell-center of cell $(i,j)$ is located at the physical coordinates $(i\Delta x, j\Delta y)$. This means that the lower-left corner of the domain does not have physical coordinates $(0,0)$, but rather $(-\Delta x/2,-\Delta y/2)$.

We carry around some auxiliary data structures to help us advance $\phi$:

- **type** (in the fortran code) or **LStype** (in the c++ code) is an “Integer Multifab” that associates each cell with one of four values:
  - 0 = part of the narrowband, not a mine
  - 1 = part of the narrowband, is a mine
  - 2 = tentative
  - 3 = outside of the narrowband

- **nbandwidth** is a fixed real number that indicates half the width of the narrowband.

- **mineloc** is a fixed real number that indicates half the width of the part of the narrowband that are not mines. It follows that **mineloc < nbandwidth**.

- **lvlerr** is a fixed real number used for the tagging condition for adaptive problems. We tag if $|\phi| < \text{lvlerr}$.

- **nband** (in the fortran code) or **LSnband** (in the c++ code) is a list of cells and associated coordinates that lie within narrowband. For example, $\text{nband}_{(1,1)}$ is the x-coordinate of the first cell in the list and $\text{nband}_{(3,2)}$ is the y-coordinate of the third cell in the list. Each grid has its own local **nband**. We mark the end of the list by setting the coordinates of the next cell to be some large negative integer, i.e., $\text{nband}_{(\text{nbandnum}+1,:)} = \text{-LARGEINT}$.
is dimensioned as a two-dimensional integer array. This was done so that each grid carries its own local copy of \texttt{nb}\texttt{and}, as opposed to each processor carrying its own local copy of \texttt{nb}\texttt{and}.

- \texttt{nb}\texttt{and}\texttt{num} is an integer indicating the number of cells in \texttt{nb}\texttt{and}.
- \texttt{mine} (in the fortran code) or \texttt{LSmine} (in the c++ code) is a list of cells and associated coordinates that are mines, and therefore, must also be part of \texttt{nb}\texttt{and} (see \texttt{nb}\texttt{and} for an example of how elements are referenced). Each grid has its own local \texttt{mine}. We mark the end of the list in the same was as \texttt{nb}\texttt{and}. Similar to \texttt{nb}\texttt{and}, \texttt{mine} is defined as an Integer Multifab even though it is used in fortran as a two-dimensional integer array.
- \texttt{heap} is a list of cells and associated coordinates used to dynamically sort tentative cells as they are evaluated in the Fast Marching Method (see \texttt{nb}\texttt{and} for an example of how elements are referenced). Each grid has its own local \texttt{heap}. \texttt{heap} contains only cells at the edge of the region containing known values. Once values in \texttt{heap} become known, they are removed from \texttt{heap}. See Section 9.4 for more details about \texttt{heap} sorting.
- \texttt{heaploc} is a two-dimensional integer array that indicates what position in \texttt{heap} a particular cell maps to. For example, if \texttt{heap}(5,1) = 8 and \texttt{heap}(5,2) = 9, then \texttt{heaploc}(8,9) = 5. If \texttt{heaploc}_i = -1, then \texttt{i} is not in \texttt{heap}.
- \texttt{intfacen} is a list of cells and associated coordinates that are next to an interface and have a negative value of $\phi$. We say that a cell is next to an interface if you can construct a box with the cell-center and neighboring three cell-centers as the vertices, in which all four cells must be within the original narrowband and do not all have the same sign of $\phi$. This indicates that the narrowband crosses the box. An additional constraint required for a cell to be considered next to an interface is that the nearest point to the interface must lie within the box in which we constructed the bicubic polynomial to find the distance to the interface. Each grid has its own local \texttt{intfacen}.
- \texttt{intfacep} is the same as \texttt{intfacen} except for positive values of $\phi$.
- \texttt{intfacen}\texttt{umn} is the number of cells in \texttt{intfacen}.
- \texttt{intfacen}\texttt{ump} is the number of cells in \texttt{intfacep}.

### 9.3 Functions

Following is a list of functions involving levelsets. They are written assuming the problem is 2D, but can be generalized to 3D in a straightforward manner.

#### 9.3.1 \texttt{INITPHI}

1. Compute $\phi$ everywhere as a signed distance function.
2. For each cell where $|\phi| < \texttt{mineloc}$, we set \texttt{type} = 0.
3. For each cell where $|\phi| < \texttt{nb}\texttt{andwidth}$, we set \texttt{type} = 1.
4. For all other cells, we set \texttt{type} = 3 and $\phi = \texttt{sign}(\infty, \phi)$. 
9.3.2 ADVANCE

1. Call **NARROWBAND** to compute \( \text{nband} \) and \( \text{mine} \) from \( \text{type} \), making sure to mark the end of both lists.

2. Call **LSCFL** to compute a stable level set advance time step for the current level, \( \Delta t_\phi \), and then set \( \Delta t_\phi = \min(\Delta t - \sum_{\text{prev}} \Delta t_\phi, \Delta t_\phi) \), where \( \Delta t \) is the full state time step for that level.

3. Call **PHIUPD** to advance \( \phi \) by \( \Delta t_\phi \).

4. If the interface crossed any mines, call **REINIT**.

5. If \( \min(\Delta t - \sum_{\text{prev}} \Delta t_\phi, \Delta t_\phi) \neq \Delta t - \sum_{\text{prev}} \Delta t_\phi \), repeat steps 2 - 4.

9.3.3 LSCFL

For each cell in \( \text{nband} \), define

\[
\text{speed} = \sqrt{\left(\frac{u_{i+\frac{1}{2}}e_x + u_{i-\frac{1}{2}}e_x}{2}\right)^2 + \left(\frac{v_{i+\frac{1}{2}}e_y + v_{i-\frac{1}{2}}e_y}{2}\right)^2 + |\kappa_a - \kappa_b\kappa|}. \tag{9.3}
\]

Define the following derivatives:

\[
\phi_x = \frac{\phi_{i+e_x} - \phi_{i-e_x}}{2\Delta x}, \tag{9.4}
\]

\[
\phi_y = \frac{\phi_{i+e_y} - \phi_{i-e_y}}{2\Delta y}, \tag{9.5}
\]

\[
\phi_{xx} = \frac{\phi_{i+e_x} - 2\phi_i + \phi_{i-e_x}}{\Delta x^2}, \tag{9.6}
\]

\[
\phi_{yy} = \frac{\phi_{i+e_y} - 2\phi_i + \phi_{i-e_y}}{\Delta y^2}, \tag{9.7}
\]

\[
\phi_{xy} = \frac{\phi_{i+e_x+e_y} - \phi_{i+e_x-e_y} - \phi_{i-e_x+e_y} + \phi_{i-e_x-e_y}}{4\Delta x\Delta y}. \tag{9.8}
\]

The curvature term is:

\[
\kappa = \begin{cases} 
0, & \phi_x^2 + \phi_y^2 \leq 0, \\
\frac{\phi_{xx}\phi_y^2 - 2\phi_{xy}\phi_x\phi_y + \phi_{yy}\phi_x^2}{(\phi_x^2 + \phi_y^2)^{3/2}}, & \text{otherwise}.
\end{cases} \tag{9.9}
\]

We have a speed constraint:

\[
\Delta t \leq \frac{0.8\Delta x}{\text{speed}}, \tag{9.10}
\]

and a curvature constraint:

\[
\Delta t \leq \frac{0.8\Delta x^2}{4|\kappa_b|}. \tag{9.11}
\]

9.3.4 PHIUPD

Now we update \( \phi \) by \( \Delta t \) using the levelset advance equation from above:

\[
\frac{\partial \phi}{\partial t} + \mathbf{U} \cdot \nabla \phi + (\kappa_a - \kappa_b\kappa)|\nabla \phi| = 0. \tag{9.12}
\]
For each cell in \textbf{nb}and whose 8 immediate neighbors are also within \textbf{nb}and, we compute the update:

\[
\phi^{\text{new}} = \phi^{\text{old}} - \Delta t \left( \mathbf{U} \cdot \nabla \phi + \kappa_a |\nabla \phi| - \kappa_b \kappa |\nabla \phi| \right).
\]  

(9.13)

We individually examine the advection, expansion, and curvature terms below. After computing \( \phi^{\text{new}} \), we determine if the interface crossed any mines by checking if \( \text{sign}(\phi) \) changed in any mines.

### 9.3.4.1 Advection Term

We first define cell-centered velocities by averaging the face-centered, time-centered velocities:

\[
\begin{align*}
    u^{\text{avg}} &= \frac{u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}}{2}, \\
    v^{\text{avg}} &= \frac{v_{i+\frac{1}{2}} + v_{i-\frac{1}{2}}}{2}.
\end{align*}
\]  

(9.14) (9.15)

We define one-sided derivatives:

\[
\begin{align*}
    \phi_x^+ &= \frac{-\phi_{i+1} + \phi_{i}}{\Delta x}, \\
    \phi_x^- &= \frac{\phi_{i} - \phi_{i-1}}{\Delta x}, \\
    \phi_y^+ &= \frac{-\phi_{i} + \phi_{i+1}}{\Delta y}, \\
    \phi_y^- &= \frac{\phi_{i} - \phi_{i-1}}{\Delta y}.
\end{align*}
\]  

(9.16) (9.17) (9.18) (9.19)

The update is:

\[
\mathbf{U} \cdot \nabla \phi = u^{\text{avg}} \phi_x^{\pm} + v^{\text{avg}} \phi_y^{\pm},
\]  

(9.20)

where \( \pm_x = \pm \) if \( u^{\text{avg}} \gtrless 0 \) and \( \pm_y = \pm \) if \( v^{\text{avg}} \gtrless 0 \).

### 9.3.4.2 Expansion Term

The expansion term uses upwind differences:

If \( \kappa_a > 0 \) then

\[
|\nabla \phi| = \sqrt{\left[ \max(\phi_x^-, 0) + \min(\phi_x^+, 0) \right]^2 + \left[ \max(\phi_y^-, 0) + \min(\phi_y^+, 0) \right]^2}
\]  

(9.21)

else

\[
|\nabla \phi| = \sqrt{\left[ \min(\phi_x^-, 0) + \max(\phi_x^+, 0) \right]^2 + \left[ \min(\phi_y^-, 0) + \max(\phi_y^+, 0) \right]^2}
\]  

(9.22)

end if

### 9.3.4.3 Curvature Term

The curvature term is:

\[
\kappa |\nabla \phi| = \begin{cases} 
0, & \phi_x^2 + \phi_y^2 \leq 0, \\
\frac{\phi_x^2 - 2\phi_x \phi_{xx} + \phi_y^2 \phi_{xy} + \phi_{yy} \phi_{xy}}{\phi_x^2 + \phi_y^2}, & \text{otherwise}.
\end{cases}
\]  

(9.23)
9.3.5 REINIT

1. Call RETYPIFY to set type = 3 for all cells in nband.

2. Call FINDINTRFCE:
   (a) For each cell $i$ in nband, set $\phi_i^{\text{new}} = \text{sign}(\infty, \phi_i^{\text{old}})$.
   (b) For each cell $i$ in nband, call UPDATEF(i):

   
   
   if cell $i$ corresponds to the bottom-left vertex of an interface box then
   i. Construct a bicubic polynomial within the interface box.
   ii. For each cell-center that is a vertex of the interface box, compute the physical
       location of the nearest point to the interface using the bicubic polynomial.
   iii. If this point lies within the interface box, return the distance to that point, making
       sure to use the min operator since any particular cell-center might be considered
       a vertex for other interface boxes. If this point does not lie within the interface
       box, do not return a distance and do not mark this cell center as being part of the
       interface.
   iv. Compute intfacep, intfacenump, intfacen, intfacenumn, making sure not to
       double count. This is done by setting type =0 after the first pass through for any
       given cell, and then checking to see if the cell’s type has been changed from 3 to 0.

   end if

   (c) Clear nband by setting nband(1,:) = -LARGEINT.

3. For each side of the interface, call FASTMARCH.

4. Fill periodic ghost cells.

5. Compute nbandnum from nband.

6. For each side of the interface, call FASTMARCH2.

7. Loop over steps 4-6 until done.

8. Call MINE to compute mine from type and mark the end of mine.

9.3.6 FASTMARCH

Here we generically refer to intfacep/intfacen as intface.

1. For each cell $i$ in intface, call UPDATE(i).

2. For each cell $i$ in intface, increment nbandnum and add $i$ to nband.

3. If numtent $>$ 0, call RMVNODE, which returns the coordinates of the removed node, $i$.

4. if $\phi_i < \text{nbandwidth}$ then
   (a) Set type$_i = 0$, increment nbandnum, and add $i$ to nband
else
   (a) Set  $\text{type}_i = 3$, set  $\phi_i = \text{sign}(\infty, \phi_i)$, and go to step [8]
end if

5. If  $\text{mineloc} < \phi_i < \text{nb bandwidth}$, set  $\text{type}_i = 1$.

6. Call  $\text{UPDATE}(i)$.

7. Repeat steps 3 - 6 while  $\text{numtent} > 0$.

8. Mark the end of  $\text{nb band}$.

9. Remove all remaining nodes $i$ from  $\text{heap}$, set  $\text{type}_i = 3$, and set  $\phi_i = \text{sign}(\infty, \phi_i)$

9.3.7 $\text{UPDATE}(i)$

1. For each cell $i'$ directly above, below, left, and right of  $i$, if  $\text{type}_{i'} > 1$ and  $\text{sign}(\phi_{i'}) = \text{sign}(\phi_i)$, call  $\text{EVAL}(i')$.

2. For each cell  $i'$ we called  $\text{EVAL}$ for:
   
   if  $\text{type} > 2$ then
      (a) Set  $\text{type} = 2$ and call  $\text{ADDNODE}(i')$.
   
   else
      (a) Call  $\text{UPDATENODE}(i')$.
   end if

9.3.8 $\text{EVAL}(i)$

1. Solve  $|\nabla \phi_i| = 1$, which can be formulated as a quadratic equation using one-sided differences with known (type $\leq 1$) points.

9.3.9 $\text{FASTMARCH2}$

1. Set  $\text{numtent} = 0$ and clear  $\text{heaploc}$.

2. For each ghost cell  $i$ directly above, below, left, or right of a valid cell, if  $\text{type}_i \leq 1$ and  $\text{sign}(\phi_i)$ is positive (if we are currently marching over the positive side of the interface) or negative (if we are marching over the negative side of the interface), call  $\text{UPDATE2}(i)$.

3. Set  $\text{done} = \text{false}$.

4. If  $\text{numtent} > 0$, call  $\text{RMVNODE}$, which returns the coordinates of the removed node,  $i$.

5. if  $\phi_i < \text{nb bandwidth}$ then
   
   (a) Set  $\text{done} = \text{true}$.

   (b) If  $\text{type}_i \geq 2$, add  $i$ to  $\text{nb band}$ and increment  $\text{nb band num}$.
(c) Set \( \text{type}_i = 0 \) and call \text{UPDATE2}(i).

else

(a) Set \( \text{type}_i = 3 \), set \( \phi_i = \text{sign}(\infty, \phi_i) \), and go to step 8.

end if

6. If \( \text{mineloc} < \phi_i < \text{nbandwidth} \), set \( \text{type}_i = 1 \).

7. Repeat steps 4 - 6 while \( \text{numtent} > 0 \).

8. Mark the end of \text{nband}.

9. Remove all remaining nodes \( i \) from \text{heap}, set \( \text{type}_i = 3 \), and set \( \phi_i = \text{sign}(\infty, \phi_i) \).

\[9.3.10 \hspace{1em} \text{UPDATE2}(i)\]

1. For each cell \( i' \) directly above, below, left, and right of ghost cell \( i \), check whether it’s a valid cell. If it’s not, skip the rest of this function and test the next \( i' \).

2. Check the following conditions:

(a) \( \text{sign}(\phi_{i'}) = \text{sign}(\phi_i) \).

(b) \( |\phi_{i'}| > |\phi_i| \).

(c) \( |\phi_{i'}| \neq \infty \) or the sign of the four cells \( \phi_{i''} \) directly above, below, left, and right of \( \phi_{i'} \) is positive (if were are currently marching over the positive side of the interface) or negative (if we are marching over the negative side of the interface).

3. If conditions 2a - 2c are all true, call \text{EVAL2}(i') (reminder: \( i' \) is the valid cell and \( i \) is the ghost cell).

4. Another call to \text{EVAL2}(i')...

5. If \text{isnew} then

(a) Set \( \text{type}_{i'} = \min(2, \text{type}_{i'}) \).

(b) If \( \text{heaploc}_{i'} = -1 \), call \text{ADDDNODE}(i'). Otherwise, call \text{UPDATENODE}(i').

\[9.3.11 \hspace{1em} \text{EVAL2}(i)\]

1. Solve \( |\nabla \phi_i| = 1 \), which can be formulated as a quadratic equation using one-sided differences with valid points. There’s some fancy checking as to whether neighboring points are valid or not. If any \( \phi \) is overwritten, set \( \text{isnew} = \text{true} \).

\[9.4 \hspace{1em} \text{Heap Sorting}\]

The heap structure that we use is called a min-heap. It has the property that every node is smaller than its children in addition to the usual heap property, i.e., that the structure is a nearly complete binary tree with nodes being filled out from left to right. Thus, the smallest node will always be
the root of the heap. In order for the min-heap to work, this is the only property that we need to satisfy, i.e., we don’t need to completely sort the heap. Because of this it only takes at most $O(\log N)$ steps to perform heap operations as opposed to $O(N \log N)$ steps for a complete sort. Any operations that we perform on the min-heap usually just involves checking to see if the current node is either larger than its parent or smaller than its children and then swapping appropriately. Jeff got the algorithm from Cormen’s book [1]. It is also mentioned in Sethian’s Level Set book [9] in Chapter 8.4, page 90. You could probably find more explanations by googling min heap (or max heap).

9.4.1 ADDNODE

**ADDNODE** works by adding a new node to the bottom of the heap. We then enforce the min-heap property by checking to see if the parent node is smaller and if so we swap the current node with its parent. We then repeat this process until the current node is smaller than its parent.

1. Put $i$ at the bottom of the heap.
2. Restructure heap and heaploc as described above.
3. Increment numtent.

9.4.2 UPDATENODE

**UPDATENODE** works the same was as ADDNODE except that the current node starts somewhere in the middle of the heap. In our case we only have to check parent nodes since nodes added from the Fast Marching Method can only be made smaller. And, so, for a more general algorithm where a node’s value may increase, you may have to also check the value of the children nodes.

1. Restructure heap and heaploc as described above.

9.4.3 RMVNODE

**RMVNODE** works by removing and returning the node with the smallest value, which is at the root of the heap. We then fill in the vacant root with the node at the end of the heap. At this point, we see if any of the children nodes are smaller than the current root node. If so, we swap the smaller of the children nodes with the current root node. This process is then repeated at the node where the swap occurred until the current node is indeed smaller than its children.

1. Remove and return the coordinates $i$ corresponding to the root of the heap.
2. Restructure heap and heaploc as described above.
3. Decrement numtent.

9.5 Future Work

- Improved hyperbolics (PLM? PPM? BDS?)
• Boundary conditions
• Checkpoint compatibility
Our approach to adaptive refinement in CASTRO uses a nested hierarchy of logically-rectangular grids with simultaneous refinement of the grids in both space and time. The integration algorithm on the grid hierarchy is a recursive procedure in which coarse grids are advanced in time, fine grids are advanced multiple steps to reach the same time as the coarse grids and the data at different levels are then synchronized.

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

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

10.1 Synchronization Algorithm

Here is the AMR algorithm for the compressible equations with self-gravity. The gravity component of the algorithm is closely related to (but not identical to) that in Miniati and Colella, JCP, 2007 (in press).

Over a coarse grid time step we collect flux register information for the hyperbolic part of the synchronization:

$$\delta F = -\Delta t_c A^c F^c + \sum \Delta t_f A^f F^f$$  \hfill (10.1)

Analogously, at the end of a coarse grid time step we store the mismatch in normal gradients of $\phi$ at the coarse-fine interface:

$$\delta F_\phi = -A^c \frac{\partial \phi^c}{\partial n} + \sum A^f \frac{\partial \phi^f}{\partial n}$$  \hfill (10.2)
Because we want the composite \( \phi^c - f \) to satisfy the multilevel version of (6.2) at each time \( t^n \), we do not accumulate \( \frac{\partial \phi^f}{\partial t} \) over time, rather we add coarse and fine fluxes only at integer coarse times.

At the end of a coarse grid time step we can define \( \overline{U}^c - f \) and \( \overline{\phi}^c - f \) as the composite of the data from coarse and fine grids as a provisional solution at time \( n + 1 \). (Assume \( \overline{U} \) has been averaged down so that the data on coarse cells underlying fine cells is the average of the fine cell data above it.)

The synchronization consists of two parts:

- **Step 1: Hyperbolic reflux**
  
  In the hyperbolic reflux step, we update the conserved variables with the flux synchronization and adjust the gravitational terms to reflect the changes in \( \rho \) and \( u \).
  
  \[
  \overline{U}^{c,*} = \overline{U}^c + \frac{\delta F}{V},
  \]
  
  (10.3)

  where \( V \) is the volume of the cell and the correction from \( \delta F \) is supported only on coarse cells adjacent to fine grids.

- **Step 2: Gravitational synchronization**
  
  In this step we correct for the mismatch in normal derivative in \( \phi^c - f \) at the coarse-fine interface, as well as accounting for the changes in source terms for \( (\rho u) \) and \( (\rho E) \) due to the change in \( \rho \).

  On the coarse grid only, we define
  
  \[
  (\delta \rho)^c = \rho^{c,*} - \overline{\rho}^c.
  \]
  
  (10.4)

  We then form the composite residual, which is composed of two contributions. The first is the degree to which the current \( \overline{\phi}^c - f \) does not satisfy the original equation on a composite grid (since we have solved for \( \overline{\phi}^c - f \) separately on the coarse and fine levels). The second is the response of \( \phi \) to the change in \( \rho \). We define
  
  \[
  R = -4\pi G (\rho^{c,*} - \Delta^c - f \phi^c - f) - 4\pi G (\delta \rho)^c - (\nabla \cdot \delta F_\phi)|c.
  \]
  
  (10.5)

  Then we solve
  
  \[
  \Delta^c - f \delta \phi^c - f = R
  \]
  
  (10.6)

  as a two level solve at the coarse and fine levels. We define the update to gravity,

  \[
  \delta g^{c,f} = \nabla (\delta \phi^c - f).
  \]
  
  (10.7)

  Define the syncsources for momentum on the coarse and fine, \( S_{\rho u}^{\text{sync},c} \), and \( S_{\rho u}^{\text{sync},f} \), respectively as follows:

  \[
  S_{\rho u}^{\text{sync},c} = \overline{\rho}^c + (\delta \rho)^c (g_c^{c,n+1} + \delta g^c - f) - \overline{\rho}^c g_c^{c,n+1}
  \]
  
  \[
  S_{\rho u}^{\text{sync},f} = \overline{\rho}^f \delta g^f.
  \]
  
  (10.8)
10.1—Synchronization Algorithm

These momentum sources lead to the following energy sources:

\[
S_{\rho E}^{\text{sync},c} = S_{\rho u}^{\text{sync},c} \cdot (\bar{u}^{c,n+1} + S_{\rho u}^{\text{sync},c} \Delta t_c / (2 \bar{\rho}^{c,n+1})) \quad (10.10)
\]

\[
S_{\rho E}^{\text{sync},f} = S_{\rho u}^{\text{sync},f} \cdot (\bar{u}^{f} + S_{\rho u}^{\text{sync},f} \Delta t_f / (2 \bar{\rho}^{f})) \quad (10.11)
\]

The coarse and fine level state is updated using:

\[
(\rho u)^{c,n+1} = (\rho u)^{c,*} + \frac{1}{2} \Delta t_c S_{\rho u}^{\text{sync},c}, \quad (\rho u)^{f,n+1} = (\rho u)^{f} + \frac{1}{2} \Delta t_f S_{\rho u}^{\text{sync},f}, \quad (10.12)
\]

\[
(\rho E)^{c,n+1} = (\rho E)^{c,*} + \frac{1}{2} \Delta t_c S_{\rho E}^{\text{sync},c}, \quad (\rho E)^{f,n+1} = (\rho E)^{f} + \frac{1}{2} \Delta t_f S_{\rho E}^{\text{sync},f}. \quad (10.13)
\]

As the final component of this step we need to

- add \( \delta \phi^{c,f} \) directly to \( \phi^c \) and \( \phi^f \) and interpolate \( \delta \phi^{c,f} \) to any finer levels and add to the current \( \phi \) at those levels.

- if level \( c \) is not the coarsest level in the calculation, then we must transmit the effect of this change in \( \phi \) to the coarser levels by updating the flux register between level \( c \) and the next coarser level, \( cc \). In particular, we set

\[
\delta F_{\phi}^{cc-c} = \delta F_{\phi}^{cc-c} + \sum A^{c} \frac{\partial (\delta \phi)^{c-f}}{\partial n}. \quad (10.14)
\]
CHAPTER 11

Equation of State and Burning Network

11.1 Equation of State

CASTRO is written in a modular fashion so that the EOS and network burning routines can be supplied by the user. However, for the examples presented later we use several EOS and network routines that come with the CASTRO distribution.

EOS routines that come with CASTRO are (listed by directory name):

- **GammaLawEOS** directory represents a gamma law gas.

- **HelmEOS** directory contains a general, publicly available stellar equation of state based on the Helmholtz free energy, with contributions from ions, radiation, and electron degeneracy, as described in (Timmes and Arnett 1999, Times and Swesty 2000, and Fryxell et al. 2000).

- **LattimerSwestyEOS** directory contains a modified version of the LS EOS available at http://www.astro.sunysb.edu/dswesty. Full documentation is available through that web site. We use this EOS in the 1D core collapse supernova example in a later section.

Each EOS directory contains two subroutines by which it interfaces to the rest of the CASTRO code. The first,

$$\text{EosGivenRTX}(e^{\text{out}}, p^{\text{out}}, \rho^{\text{in}}, T^{\text{in}}, X^{\text{in}}, n_{\text{spec}}, Y^{\text{in}}, \text{n}_{\text{aux}})$$

is a direct interface to the EOS, in which density, species and auxiliary variables, and temperature are specified, and the necessary thermodynamical variables such as internal energy, pressure, gamma, and sound speed are returned.

The second routine,

$$\text{EosGivenReX}(\Gamma^{\text{out}}, p^{\text{out}}, e^{\text{out}}, T^{\text{out}}, \rho^{\text{in}}, e^{\text{in}}, X^{\text{in}}, n_{\text{spec}}, Y^{\text{in}}, \text{n}_{\text{aux}})$$

59
uses a Newton iteration to find the temperature given the internal energy, density, and species and auxiliary variables.

### 11.2 Burning Network

Burning network routines that come with CASTRO are (listed by directory name):

- **networks/null** directory describes a non-reacting white dwarf, with only hydrogen, helium and carbon12. There are no auxiliary variables, and no reactions are allowed.

- **networks/collapse** directory describes a pre-supernova neutron star with hydrogen, helium, oxygen and iron. There is one auxiliary variable, Ye, the electron fraction. Again no reactions are allowed.

- **networks/ignition** directory contains a single-step $^{12}\text{C}(^{12}\text{C},\gamma)^{24}\text{Mg}$ reaction. The carbon mass fraction equation appears as

\[
\frac{DX(^{12}\text{C})}{Dt} = -\frac{1}{12}\rho X(^{12}\text{C})^2 f_{\text{Coul}} [N_A \langle \sigma v \rangle] ,
\]

where $N_A \langle \sigma v \rangle$ is evaluated using the reaction rate from (Caughlan and Fowler 1988). The Coulomb screening factor, $f_{\text{Coul}}$, is evaluated using the general routine from the Kepler stellar evolution code (Weaver 1978), which implements the work of (Graboske 1973) for weak screening and the work of (Alastuey 1978 and Itoh 1979) for strong screening.

There are two primary files within each network directory. The first, castro_burner.f90, contains the burner routine, which takes $\rho^{\text{in}}, e^{\text{in}}, X_k^{\text{in}}$, and $\Delta t$ as inputs. It is possible for the internal energy, $e$, which is computed from $U$, to be negative due to roundoff error. CASTRO has an option to protect against using a negative value of $e$ by recomputing $e = e(\rho, T_{\text{small}}, X_k)$ using the equation of state, where $T_{\text{small}}$ is a user-defined temperature floor. In the event that $e$ is still negative, we abort the program. CASTRO also has an option to skip the reactions if the density is below a user-defined density floor.

Next, the burner computes $T = T(\rho^{\text{in}}, e^{\text{in}}, X_k^{\text{in}})$ using the equation of state. The burner returns $X_k^{\text{out}}$ and $e^{\text{out}}$ by solving over a time interval of $\Delta t/2$,

\[
\frac{\partial X_k}{\partial t} = \dot{\omega}_k .
\]

In particular, to evolve the species, we solve the system:

\[
\frac{dX_k}{dt} = \dot{\omega}_k(\rho, X_k, T) ,
\]

\[
\frac{dT}{dt} = \frac{1}{c_p} \left( -\sum_k \xi_k \dot{\omega}_k \right) .
\]

using the stiff ordinary differential equation integration methods provided by the VODE package. The absolute error tolerances are set to $10^{-12}$ for the species, and a relative tolerance of $10^{-5}$ is used for the temperature. The integration yields the new values of the mass fractions, $X_k^{\text{out}}$. Equation
is derived from equation (???) by assuming that the pressure is constant during the burn state. In evolving these equations, we need to evaluate $c_p$ and $\xi_k$. In theory, this means evaluating the equation of state for each right-hand side evaluation that VODE requires. In practice, we freeze $c_p$ and $\xi_k$ at the start of the integration time step and compute them using $\rho^{in}, X^{in}_k$, and $T^{in}$ as inputs to the equation of state. Note that the density remains unchanged during the burning. At the end of the routine, we compute $T^{out} = T(\rho^{out}, e^{out}, X^{out}_k)$.

The second file, “network.f90”, supply the number of species and auxiliary variables, names of each species and auxiliary variable, as well as other initializing data, such as aion, zion and the binding energy.

It is straightforward to implement additional EOS and network routines; all that is required is to create an appropriate interface to the CASTRO calls, which is easily done given the prototypes supplied with the CASTRO distribution.
Within the CASTRO distribution, there is the capability to “grow” a checkpoint file so that a calculation can be restarted in a larger domain covered by grid cells a factor of two or four coarser than the existing coarsest level. Instructions for how to do so are in the Castro/ConvertCheckpoint/README file and are included here. Upon restart the existing data in the checkpoint file will be used to fill the region of the previous computational domain, and the new regions will be filled by some other means, typically interpolation from a 1D model file.

12.1 Star in Corner (star_at_center = 0)

In this section we consider the case where the star (or feature of interest) is centered at the lower left corner of the domain, e.g. you are modeling only one quarter of the star in 2D, or an octant of the star in 3D. Then you only want to grow the domain in the “high side” directions (e.g., to the upper right).

12.1.1 Converting the Checkpoint File

Let’s say you have a checkpoint file, chk00100, say, with 5 levels of refinement and a (real) problem domain size \( P \) and (integer) domain size \( D \) at level 0.

The inputs file that created this might have contained:

- \texttt{max\_step} = 100
- \texttt{amr.max\_level} = 5
- \texttt{amr.n\_cell} = D D
- \texttt{geometry.prob\_lo} = 0 0
• geometry.prob_hi = P P
• amr.ref_ratio = 4 4 4 4

Now let’s suppose that you want to grow the domain by a factor of 8 and cover that new larger domain with a level that is a factor of 2 coarser than the existing level 0 grids.

1. First, set DIM = in the GNUmakefile, and type ”make” in the ConvertCheckpoint directory. This will make an executable from the Embiggen.cpp code.

2. Run the embiggening code as follows:

   Embiggen2d.Linux.Intel.Intel.ex checkin=chk00100 checkout=newchk00050 ref_ratio=2
grown_factor=8 star_at_center=0

   (Your executable may have a slightly different name depending on the compilers you built it with.)

   This will create a new checkpoint directory, called newchk00050, that represents a simulation with one additional level of refinement coarser than the previous level 0 grids by a factor of ref_ratio (in this case, 2). The new domain will be a factor of grown_factor (in this case, 8) larger than the previous domain.

   Note that ref_ratio must be 2 or 4, because those are the only acceptable values of ref_ratio in CASTRO.

   grown_factor can be any reasonable integer; I’ve only tested 2, 3, 4 and 8. It does not need to be a multiple of 2.

12.1.2 Restarting from a Grown Checkpoint File

You should now be able to restart your calculation using newchk00050.

Your inputs file should now contain lines like:

• max_step = 51
• amr.restart = newchk00050
• amr.max_level = 6
• amr.n_cell = 4D 4D
• geometry.prob_lo = 0 0
• geometry.prob_hi = 8P 8P
• castro.grown_factor = 8
• castro.star_at_center = 0
• amr.ref_ratio = 2 4 4 4 4

IMPORTANT:

1. Unlike earlier, you may now set amr.max_level to be at most one greater than before, but you need not set it that high. For example, you could set amr.max_level the same as before
and you would lose data at the finest refinement level. You may not set \texttt{amr.max\_level} = 0, however, because we have no data at the new level 0 until we average down from the new level 1 after the restart.

2. You must set \texttt{amr.n\_cell} = (\texttt{grown\_factor} / \texttt{ref\_ratio}) times the previous value of \texttt{amr.n\_cell}. In this case \texttt{amr.n\_cell} = (8/2)*D = 4D.

3. You must set \texttt{amr.prob\_hi} to be a factor of \texttt{grown\_factor} greater than the previous value of \texttt{amr.prob\_hi}.

4. You must insert the value of \texttt{ref\_ratio} used in the Embiggen call as the first value in the list of \texttt{amr.ref\_ratio}, since that will now be the refinement ratio between the new level 0 and the new level 1.

5. You must set \texttt{castro.grown\_factor} in your inputs file equal to the value of \texttt{grown\_factor} you used when you called Embiggen\textunderscore ex so that the CASTRO code knows how big the original domain was.

6. Note that if you have run 100 steps at the original level 0, that would be equivalent to 50 steps at the new level 0 because you coarsened by a factor of 2. Thus once you re-start from the new checkpoint directory, the next step will be 51, not 101. Make sure to keep track of your plotfiles accordingly.

7. Don’t forget to adjust \texttt{max\_denerr\_lev} and comparable variables to control the number of fine levels you now want. If you want to have 6 levels of refinement after restart, then make sure \texttt{max\_denerr\_lev}, etc, are set high enough. If you only want to have 5 levels of refinement (where the new level 5 would now be a factor of \texttt{ref\_ratio} coarser than the previous level 5), make sure to adjust \texttt{max\_denerr\_lev} accordingly as well.

### 12.2 Star at Center of Domain (\texttt{star\_at\_center} = 1)

Now let’s assume that the star (or feature of interest) is centered at the center of the domain in 2D or 3D Cartesian coordinates. We will later consider the case of 2D cylindrical (r-z) coordinates in which the star is centered at the left midpoint.

#### 12.2.1 Converting the Checkpoint File

Suppose that you want to grow the domain by a factor of 2 and cover that new larger domain with a level that is a factor of 2 coarser than the existing level 0 grids.

After you build the Embiggen executable, you type:

- \texttt{Embiggen2d.Linux.Intel.Intel.ex checkin=chk00100 checkout=newchk00050 ref\_ratio=2 grown\_factor=2 star\_at\_center=1}

Note that

- \texttt{ref\_ratio} must still be 2 or 4
- \texttt{grown\_factor} can only be 2 or 3 in this case.
12.2.2 Restarting from a Grown Checkpoint File

Your inputs file for restarting would now look like

- `max_step = 51`
- `amr.restart = newchk00050`
- `amr.max_level = 6`
- `amr.n_cell = D D`
- `geometry.prob_lo = -P/2 -P/2`
- `geometry.prob_hi = 3P/2 3P/2`
- `castro.grown_factor = 2`
- `castro.star_at_center = 1`
- `amr.ref_ratio = 2 4 4 4 4`

12.2.3 Cylindrical Coordinates

In the case of 2D cylindrical (r-z) coordinates in which the star is centered at the left edge but vertical midpoint of the domain, the embiggening procedure is the same as above (with `star_at_center = 1`) but the inputs file for restart is slightly different in that `geometry.prob_lo` is modified in the z- but not the r-direction. If we consider the original inputs file to look like:

- `max_step = 100`
- `amr.max_level = 6`
- `amr.n_cell = D 2D`
- `geometry.prob_lo = 0 0`
- `geometry.prob_hi = P 2P`
- `amr.ref_ratio = 4 4 4 4`

then an inputs file for restart would look like:

- `amr.restart = newchk00050`
- `amr.max_level = 6`
- `amr.n_cell = D 2D`
- `geometry.prob_lo = 0 -P`
- `geometry.prob_hi = 2P 3P`
- `castro.grown_factor = 2`
- `castro.star_at_center = 1`
- `amr.ref_ratio = 2 4 4 4 4`
Figure 12.1: Data from checkpoint file before and after the domain has been coarsened and grown. This case uses $\text{star\_at\_center} = 0$ and $\text{ref\_ratio}=2$. The first grown example has $\text{grown\_factor}=2$, the second has $\text{grown\_factor}=3$. In all figures the level 0 grids are shown in white, the level 1 grids in red, the level 2 grids in yellow, and in the grown figures, the level 3 grids are in pink.
Figure 12.2: Data from checkpoint file before and after the domain has been coarsened and grown. This case uses \texttt{star\_at\_center} = 0 and \texttt{ref\_ratio}=2. The first grown example has \texttt{grown\_factor}=2, the second has \texttt{grown\_factor}=3. In all figures the level 0 grids are shown in white, the level 1 grids in red, the level 2 grids in yellow, and in the grown figure, the level 3 grids are in pink.
CHAPTER 13

Initializing CASTRO with MAESTRO Data

13.1 Overview

We can now initialize a CASTRO simulation using data from a MAESTRO plotfile. This should not be thought of as a restart mode, but rather a new simulation with a special initialization. In order to use this feature, you must make sure the MAESTRO plotfile has the proper variables, add some new parameters to your inputs file, and add a few subroutines to Prob_Xd.f90. You need to build a special executable with “USE_MAESTRO_INIT=TRUE”, which will add “.MAESTRO” to the executable string. For multilevel problems, there are a few extra steps relating to the fact that you have to supply a grids file consistent with the MAESTRO grid structure.

13.2 MAESTRO Plotfile Requirements

The MAESTRO plotfile needs to have the following variables:

- “x_vel”, “y_vel”, (and “z_vel”, depending on dimensionality of the problem)
- “density” (castro.MAESTRO_init_type = 1 and 2 only)
- Optional species (such as “X(C12)” - there is an option to not read any species from the MAESTRO plotfile. In this case, you must make sure your code manually defines the species cell-by-cell in the initial CASTRO data
- “tfromp”
- “pi” (castro.MAESTRO_init_type = 2, 3, and 4 only)
- “entropy” (castro.MAESTRO_init_type = 4 only)
Also, model_cc_XXXX needs to list variables in the following order, which is the default order found in MAESTRO/Source/base_io.f90: r, base_r, rho0, p0, gamma1bar, rhoh0, div_coeff, psi, tempbar, etarho_cc, tempbar_init.

### 13.3 List of Parameters

Here are the additional parameters you must add to your inputs file.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>castro.MAESTRO_plotfile</td>
<td>name of the MAESTRO plotfile</td>
<td>std::string</td>
<td>must be set</td>
</tr>
<tr>
<td>castro.MAESTRO_modelfile</td>
<td>name of the MAESTRO “model_cc” file</td>
<td>std::string</td>
<td>must be set</td>
</tr>
<tr>
<td>castro.MAESTRO_npts_model</td>
<td>number of points in the MAESTRO model_cc file</td>
<td>int</td>
<td>must be set</td>
</tr>
<tr>
<td>castro.MAESTRO_first_species</td>
<td>name of the first species</td>
<td>std::string</td>
<td>must be set or else nothing will be read in</td>
</tr>
<tr>
<td>castro.MAESTRO_nspec</td>
<td>number of species in the MAESTRO plotfile</td>
<td>std::string</td>
<td>NumSpec in CASTRO</td>
</tr>
<tr>
<td>castro.MAESTRO_init_type</td>
<td>determines how we initialize the CASTRO state</td>
<td>int</td>
<td>must be set</td>
</tr>
<tr>
<td>castro.MAESTRO_spherical</td>
<td>specifies planar or spherical problem</td>
<td>int</td>
<td>must be set</td>
</tr>
</tbody>
</table>

#### 13.3.1 Examples of Usage

- `castro.MAESTRO_plotfile = "wd_384_6.25e8K_norotate_plt120578"

- `castro.MAESTRO_modelfile = "/wd_384_6.25e8K_norotate_plt120578/model_cc_120578"

- `castro.MAESTRO_npts_model = 1663`
  This is the number of points in `castro.MAESTRO_modelfile`. Note that this is not the same thing as “npts_model”, which is the number of points in the initial model file used for standard simulations where we do not initialize from a MAESTRO plotfile.

- `castro.MAESTRO_first_species = “X(C12)”` If you do not specify this, no species will be read in. You can always manually specify or overwrite the species cell-by-cell later.

- `castro.MAESTRO_nspec = 3`
  If you do not specify this, it will default to the number of species in the CASTRO network, “NumSpec”. We have this here because sometimes MAESTRO and CASTRO will use different networks with different number of species.

- `castro.MAESTRO_cutoff_density = 1.66`
  The code will use this density to figure out the radial coordinate, r_model_start, which is the last radial coordinate before rho0 falls below `castro.MAESTRO_cutoff_density`. It is possible to set `castro.MAESTRO_cutoff_density` to a tiny value, such that rho0 never falls below this value, in which case we set r_model_start to \( \infty \). In INITDATA_MAKEMODEL, we create a new 1D model integrating outward starting from \( r_{\mathrm{model}\_\text{start}} \). Then, in INITDATA_OVERWRITE, we overwrite newly initialized CASTRO data in any cell that maps into a radial coordinate greater than \( r_{\mathrm{model}\_\text{start}} \) by interpolating from the new 1D model.

- `castro.MAESTRO_init_type = 2`
  CASTRO will read in data from the MAESTRO plotfile, and then call the EOS to make sure that \( \rho, e, T, \) and \( X_k \) are consistent. The inputs to the EOS are based on the value of `castro.MAESTRO_init_type`:

  1. \( e = e(\rho, T, X_k) \)
2. \( e, T = e, T(\rho, p_0 + \pi, X_k) \)
3. \( \rho, e = \rho, e(\rho_0 + \pi, T, X_k) \)
4. \( \rho, T, e = \rho, T, e(\rho_0 + \pi, s, X_k) \)

- castro.MAESTRO_spherical = 1
0 = planar; 1 = spherical.

13.4 New Subroutines in Prob_Xd.f90

There are three routines that need to be added to your local copy of Prob_Xd.f90. See Castro/Exec/wdconvect/Prob_3d.f90 for a standard spherical MAESTRO initialization.

1. INITDATA_MAEOSTRO
   This fills in the CASTRO state by taking the MAESTRO data, calling the EOS, and making the proper variables conserved quantities. Specifically, we need a thermodynamically consistent \( \rho, T, e, \) and \( X_k \), and then algebraically compute \( \rho u, \rho e, \rho E, \) and \( \rho X_k \).

2. INITDATA_MAKEMODEL
   This creates a user-defined 1D initial model starting from \( r_{\text{model start}} \).

3. INITDATA_OVERWRITE
   This overwrites the initialized CASTRO data using the new 1D initial model for all cells that map into radial coordinates greater than \( r_{\text{model start}} \).

13.5 Additional Notes

Note that for both single-level and multilevel MAESTRO to CASTRO initialization, the CASTRO base grid structure does not have to match the MAESTRO base grid structure, as long as the problem domain is the same. For example, if the coarsest level in a MAESTRO plotfile contains \( 64^3 \) cells divided into 8-32 \( 3 \) grids, it is ok to use a CASTRO base grid structure with 1-64 \( 3 \) grid, 64-16 \( 3 \) grids, or anything else you can imagine - the grids don’t even have to be the same size. As is normally the case, the CASTRO base grid structure is created based on the parameters in the CASTRO inputs file, such as \texttt{amr.max_grid_size}, \texttt{amr.blocking_factor}, etc.

13.5.1 Multilevel Restart

When initializing from a multilevel MAESTRO plotfile, there are some extra steps. First, you need to create a CASTRO-compatible grids file from the MAESTRO plotfile. This can be done with the AmrPostprocessing/F_Src/fboxinfo.f90 utility. Compile and run this using the “--castro” option, e.g., “fboxinfo.Linux.gfortran.exe --castro pltxxxxx 1 tee gr0.maestro”, to generate the CASTRO-compatible grids file. Note that the base grid structure is still controlled by \texttt{amr.max_grid_size}, \texttt{amr.blocking_factor}, etc., since in C++ BoxLib, the grids file only indicates the refined grid structure, whereas in Fortran BoxLib the grids file contains the base grid and refined grid structures.

Now, when you initialize the CASTRO simulation, you need to specify the grid file using \texttt{amr.regrid_file = ”gr0_3d.128_2levels”}, for example. You can happily run this now, but
note that the regridding algorithm will never be called (since CASTRO thinks it’s started a new simulation from scratch with a grids file, thus disabling the regridding). If you wish for the grid structure to be changed, you must do a traditional CASTRO restart from the CASTRO-generated checkpoint file (you can still use the same “.MAESTRO” executable or an executable built with USE_MAESTRO_INIT=FALSE), making sure that you do not specify \texttt{amr.regrid\_file} (or else the grids will stay fixed). You are free to specify \texttt{amr.regrid\_on\_restart}, \texttt{amr.compute\_new\_dt\_on\_regrid}, and \texttt{amr.plotfile\_on\_restart}.

Sometimes a MAESTRO plotfile will only have 1 or 2 total levels, but you ultimately want to run a CASTRO simulation with many more levels of refinement. My recommended strategy is the following:

1. Initialize a CASTRO simulation from the MAESTRO plotfile while preserving the exact same grid structure and run for 10 time steps.

2. Do a traditional CASTRO restart from chk00010, but do not increase \texttt{amr.max\_level}, and run for 10 more time steps. This allows a new grid structure with the same effective resolution as before settle in using the C++ BoxLib regridding algorithm.

3. Do a traditional CASTRO restart from chk00020, but increase \texttt{amr.max\_level} by 1, and run for 10 time steps.

4. Repeat the procedure from the previous step (using the most updated checkpoint of course) as many times as desired.
14.1 2D and 3D

14.1.1 amrvis

Our favorite visualization tool is amrvis. We heartily encourage you to build the 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 – 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.

If you like to have amrvis display a certain variable, at a certain scale, when you first bring up each plotfile (you can always change it once the amrvis window is open), you can modify the amrvis.defaults file in your directory to have amrvis default to these settings every time you run it. The directories CoreCollapse, HSE_test, Sod and Sedov have amrvis.defaults files in them. If you are working in a new run directory, simply copy one of these and modify it.

14.1.2 VisIt

VisIt is also a great visualization tool, and it directly handles our plotfile format (which it calls Boxlib). For more information check out visit.llnl.gov.

[Useful tip:] To use the Boxlib3D plugin, select it from File → Open file → Open file as type Boxlib, and then the key is to read the Header file, plt00000/Header, for example, rather than telling to to read plt00000.
14.2 Controlling What’s in the PlotFile

```
amr.plot.vars =

and

amr.derive.plot.vars =
```

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

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

```
amr.derive.plot.vars = entropy pressure
```

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

```
amr.plot.vars = density

amr.derive.plot.vars = pressure
```

14.3 1D

amrvis doesn’t like 1-d plotfiles, and for those we use a 1-d plotting capability installed by Mike Singer Castro/Util/plot1d. If you want to make xmgrace-compatible files, for example, add the following to your inputs file:

```
xgraph.xmgrace_file = 1
xgraph.format = xmg
xgraph.use_xmgrace_legend = 1
xgraph.use_xmgrace_title = 1
xgraph.graph = xvel x_velocity 100 -1
```

This tells is to write a file called `xvel_0000.xmgr`, for example, every 100 time steps, including all levels of data. (The last variable, -1, specifies the maximum level; if it is -1 then all levels are used.)
If you want to write more than one variable into a single file, then instead of setting each variable on a separate line as in the xvel example above, you can do the following:

\[ \text{xgraph.graph} = \text{file.name ALL 100 -1} \]

If you specify “ALL” then

\[ \text{amr.plot.vars} = \]

and

\[ \text{amr.derive.plot.vars} = \]

are used to control which variables are included. The default for \text{amr.plot.vars} is all of the state variables. The default for \text{amr.derive.plot.vars} is none of the derived variables. So if you include neither of these lines then the file file.name.xmg will contain all of the state variables and none of the derived variables.

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

\[ \text{amr.derive.plot.vars} = \text{entropy pressure} \]

\[ \text{xgraph.graph} = \text{file.name ALL 100 -1} \]

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

\[ \text{amr.plot.vars} = \text{density} \]

\[ \text{amr.derive.plot.vars} = \text{pressure} \]

\[ \text{xgraph.graph} = \text{file.name ALL 100 -1} \]

Feel free to read the routines in Castro/Util/plot1d.
CHAPTER 15

Software Framework

15.1 Code structure

The code structure in the Castro directory is as follows:

- **constants**: contains a file of useful constants in CGS units
- **ConvertCheckpoint**: a tool to convert a checkpoint file to a larger domain
- **EOS**: contains directories for different EOS routines
- **Exec**: various examples
  - **Sedov**: run directory for the Sedov problem
  - **Sod**: run directory for the Sod problem
  - **KH**: run directory for the Kelvin-Helmholz problem
- **Networks**: contains directories for different reaction networks
- **Source**: source code
- **UsersGuide**: you’re reading this now!
- **Util**: a catch-all for additional things you may need
15.2 Castro Data Structures

15.2.1 State Data

CASTRO relies on the class structure defined by BoxLib to manage the data.

In Castro.H, the enum StateType defines the different descriptors for the state data that Castro recognizes. The main descriptors are:

- **State_Type**: the state variables for the hydrodynamics solver.
- **Rad_Type**: the radiation quantities (only enabled if RADIATION is defined).
- **Gravity_Type**: the data required for the gravity solve (only enabled if GRAVITY is defined).
- **Reactions_Type**: what is this for?

The state data is registered with BoxLib in Castro_setup.cpp. We access the multifabs that carry the data of interest by interacting with this BoxLib data-structure. Each state quantity always has both an old and new timestate and the BoxLib class knows how to interpolate in both space and time. We interact with the data by getting pointers to multifabs. For instance:

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

gets a pointer to the multifab containing the hydrodynamics state data at the new time (here State_Type is the enum defined in Castro.H).

We iterate over the multifabs using an iterator MFIter. This iterator knows about the locality of the data—only the boxes on the processor will be looped over. An example loop (for the initialization, from Castro_setup.cpp would be):

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

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

here BL_TO_FORTRAN is a special BoxLib macro that converts the C++ multifab into a Fortran array, and BL_FORT_PROC_CALL is a BoxLib macro that is used to interface with Fortran routines.
15.2.2 Other Quantities

The following is a list of variables, routines, etc used in CASTRO. It may not be complete or even entirely accurate; it’s mostly intended for my own use.

`lo, hi`: index extent of the "grid" of data currently being handled by a CASTRO routine

`domlo, domhi`: index extent of the problem domain. This changes according to refinement level: 0th refinement level will have 0, `castro.max_grid_size`, and nth level will go from 0 to `castro.max_grid_size*(multiplying equivalent of sum)castro.ref_ratio(n)`. 

`dx`: cell spacing, presumably in cm, since CASTRO uses cgs units

`xlo`: physical location of the lower left-hand corner of the "grid" of data currently being handled by a CASTRO routine

`bc`: array that holds boundary condition of and array. Sometimes it appears of the form `bc(:,;:)` and sometimes `bc(:,;,:)`. The last index of the latter holds the variable index, i.e. density, pressure, species, etc.

`EXT_DIR`: from BoxLib/Src/C_AMRLib/BC_TYPES.H:EXT_DIR : data specified on EDGE (FACE) of bndry

`FOEXTRAP`: from BoxLib/Src/C_AMRLib/BC_TYPES.H:FOEXTRAP : first order extrapolation from last cell in interior CASTRO

15.3 Setting Up Your Own Problem

To define a new problem, we create a new directory under Exec/, and place in it a `Prob_2d.f90` file (or 1d/3d, depending on the dimensionality of the problem), a `probdatal.f90` file, the `inputs` and `probin` files, and a `Make.package` file that tells the build system what problem-specific routines exist. The simplest way to get started is to copy these files from an existing problem. Here we describe how to customize your problem.

A typical `Prob_?d.f90` routine consists of the following subroutines:

- `PROBINIT`
- `ca_initdata`
- the `*fill` routines: The following routines handle how CASTRO fills ghostcells for specific data. The idea is that these routines are registered in Castro_setup.cpp, and called as needed. By default, they just pass the arguments through to `filcc`, which handles all of the generic boundary conditions (like reflecting, extrapolation, etc.). The specific 'fill' routines can then supply the
problem-specific boundary conditions, which are typically just Dirichlet boundary conditions. The code implementing these specific conditions should follow the filcc call.

- **ca_hypfill**: This handles the boundary filling for the hyperbolic system.

- **ca_denfill**: At times, we need to fill just the density (always assumed to be the first element in the hyperbolic state) instead of the entire state. When the fill patch routine is called with `first_comp = Density` and `num_comp = 1`, then we use `ca_denfill` instead of `ca_hypfill`.

- **ca_gravfill**: These routines will the ghostcells with the gravitational acceleration. By default, they will just do something like a first-order extrapolation. These are needed for the hydro routines to have the gravitational acceleration needed for the source terms to the interface states.

- **ca_reactfill**

### 15.4 Boundaries

#### 15.4.1 Boundaries Between Grids

Boundaries between grids are of two types. The first we call ”fine-fine”, which is two grids at the same level. Filling ghost cells at the same level is also part of the fillpatch operation – it’s just a straight copy from ”valid regions” to ghost cells. The second type is ”coarse-fine”, which needs interpolation from the coarse grid to fill the fine grid ghost cells. This also happens as part of the FillPatch operation, which is why arrays aren’t just arrays, they’re ”State Data”, which means that the data knows how to interpolate itself (in an anthropomorphical sense). The type of interpolation to use is defined in Castro\_setup.cpp as well – search for `cell_cons_interp`, for example – that’s ”cell conservative interpolation”, i.e the data is cell-based (as opposed to node-based or edge-based) and the interpolation is such that the average of the fine values created is equal to the coarse value from which they came. (This wouldn’t be the case with straight linear interpolation, for example.)

A **FillPatchIterator** is used to loop over the grids and fill ghostcells. One should never assume that ghostcells are valid. A key thing to keep in mind about the FillPatchIterator is that you operate on a copy of the data—the data is disconnected from the original source. If you want to update the data in the source, you need to explicitly copy it back. Also note: FillPatchIterator takes a multifab, but this is not filled—this is only used to get the grid layout.

**simple example**

#### 15.4.2 Physical Boundaries

<table>
<thead>
<tr>
<th>Physical BC</th>
<th>Velocity</th>
<th>Temperature</th>
<th>Scalars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outflow</td>
<td>FOEXTRAP</td>
<td>FOEXTRAP</td>
<td>FOEXTRAP</td>
</tr>
<tr>
<td>No Slip Wall with Adiabatic Temp</td>
<td>EXT_DIR ( u = v = 0 )</td>
<td>REFLECT_EVEN (dT/dt = 0 )</td>
<td>HOEXTRAP</td>
</tr>
<tr>
<td>No Slip Wall with Fixed Temp</td>
<td>EXT_DIR ( u = v = 0 )</td>
<td>EXT_DIR</td>
<td>HOEXTRAP</td>
</tr>
<tr>
<td>Slip Wall with Adiabatic Temp</td>
<td>EXT_DIR ( u_n = 0 ), HOEXTRAP ( u_t )</td>
<td>REFLECT_EVEN (dT/dn = 0 )</td>
<td>HOEXTRAP</td>
</tr>
<tr>
<td>Slip Wall with Fixed Temp</td>
<td>EXT_DIR ( u_n = 0 )</td>
<td>EXT_DIR</td>
<td>HOEXTRAP</td>
</tr>
</tbody>
</table>

**Table 15.1**: Conversions from physical to mathematical BCs
The boundary conditions in Table 15.1 have already been implemented in CASTRO. The table looks cruddy—it’s copied from BoxLib/Src/C_AMRLib/amrlib/BC_TYPES.H. Some of that makes more sense if there are linebreaks within the table, but I’m not sure how to do it. Here’s definitions of some of the funnier-sounding all-caps words from above:

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

Basically, boundary conditions are imposed on ”state variables” every time that they’re ”fill-patched”, as part of the fillpatch operation.

For example, the loop that calls CA_UMDRV (all the integration stuff) starts with

```c
for (FillPatchIterator fpi(*this, S_new, NUM_GROW, time, State_Type, strtComp, NUM_STATE);
    fpi.isValid(); ++fpi)
```

Here the FillPatchIterator is the thing that distributes the grids over processors and makes parallel ”just work”. This fills the single patch ”fpi”, which has NUM_GROW ghost cells, with data of type ”State_Type” at time ”time”, starting with component strtComp and including a total of NUM_STATE components.

The way that you tell the code what kind of physical boundary condition to use is given in Castro_setup.cpp. At the top we define arrays such as ”scalar_bc”, ”norm_vel_bc”, etc, which say which kind of bc to use on which kind of physical boundary. Boundary conditions are set in functions like ”set_scalar_bc”, which uses the scalar_bc pre-defined arrays.

If you want to specify a value at a function (like at an inflow boundary), there are routines in Prob_1d.f90, for example, which do that. Which routine is called for which variable is again defined in Castro_setup.cpp

## 15.5 Parallel I/O

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

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

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

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

16.1.1 Sod’s Problem (and Other Shock Tube Problems)

The Exec/Sod problem directory sets up a general one-dimensional shock tube. The left and right primitive-variable states are specified and the solution evolves until a user-specified end time. For a simple discontinuity, the exact solution can be found from an exact Riemann solver. For this problem, the exact solutions were computed with the exact Riemann solver from Toro [12], Chapter 4.

16.1.1.1 Sod’s Problem

The Sod problem [10] is a simple shock tube problem that exhibits a shock, contact discontinuity, and a rarefaction wave. The initial conditions are:

\[
\begin{align*}
\rho_L &= 1 & \rho_R &= 0.125 \\
u_L &= 0 & u_R &= 0 \\
p_L &= 1 & p_R &= 0.1
\end{align*}
\]  

(16.1)

The gamma law equation of state is used with \( \gamma = 1.4 \). The system is evolved until \( t = 0.2 \) s. Setups for 1-, 2-, and 3-d are provided. The following inputs files and probin files setup the Sod’s problem:

For multi-dimensional runs, the directions transverse to the jump are kept constant. We use a CFL number of 0.9, an initial timestep shrink (castro.init.shrink) of 0.1, and the maximum factor by which the timestep can increase (castro.change_max) of 1.05.
Chapter 16. Verification Test Problems

<table>
<thead>
<tr>
<th>inputs file</th>
<th>probin file</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs-sod-x</td>
<td>probin-sod-x</td>
<td>Sod’s problem along $x$-direction</td>
</tr>
<tr>
<td>inputs-sod-y</td>
<td>probin-sod-y</td>
<td>Sod’s problem along $y$-direction</td>
</tr>
<tr>
<td>inputs-sod-z</td>
<td>probin-sod-z</td>
<td>Sod’s problem along $z$-direction</td>
</tr>
</tbody>
</table>

Figure 16.1: Castro solution for Sod’s problem run in 3-d, with the newest ppm limiters, along the $x$, $y$, and $z$ axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line.

Figure [16.1] shows the Castro solution using the newest PPM limiters compared to the analytic solution, showing the density, velocity, pressure, and internal energy. Figure [16.2] is the same as Figure [16.1] but with the piecewise-linear Godunov method with limiters, shown for comparison.

The Verification subdirectory includes the analytic solution for the Sod problem sod-exact.out, with $\gamma = 1.4$. 1-d slices can be extracted from the Castro plotfile using the $\text{fextract}$ tool from AmrPostprocessing/F_Src/. The steps to generate this verification plot with Castro are:

1. in Exec/Sod, build the Castro executable in 3-d

2. run the Sod problem with Castro in the $x$, $y$, and $z$ directions:
   ```
   ./Castro3d.Linux.Intel.Intel.ex inputs-sod-x
   ./Castro3d.Linux.Intel.Intel.ex inputs-sod-y
   ./Castro3d.Linux.Intel.Intel.ex inputs-sod-z
   ```

3. build the $\text{fextract}$ tool in AmrPostprocessing/F_Src/.

4. run $\text{fextract}$ on the Castro output to generate 1-d slices through the output:
   ```
   fextract3d.Linux.Intel.exe -d 1 -s sodx.out -p sod_x_plt00034
   fextract3d.Linux.Intel.exe -d 2 -s sody.out -p sod_y_plt00034
   fextract3d.Linux.Intel.exe -d 3 -s sodz.out -p sod_z_plt00034
   ```

5. copy the sodx/y/z.out files into the Verification directory.
Figure 16.2: Castro solution for Sod’s problem run in 3-d, with the piecewise-linear Godunov method with limiters, along the $x$, $y$, and $z$ axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line.

6. in Verification run the gnuplot script `sod_3d.gp` as:
   
   ```
   gnuplot sod_3d.gp
   ```
   This will produce the figure `sod_3d.eps`.

### 16.1.1.2 Double Rarefaction

The double rarefaction is the “Test 2” problem described by Toro [12], Chapter 6. In this test, the center of the domain is evacuated as two rarefaction waves propagate in each direction, outward from the center. It is difficult to get the internal energy to behave at the center of the domain because we are creating a vacuum. The initial conditions are:

\[
\begin{align*}
\rho_L &= 1 & \rho_R &= 1 \\
\rho_L &= 0.4 & \rho_R &= 0.4 \\
\end{align*}
\]

(16.2)

The `gamma_law` equation of state is used with $\gamma = 1.4$. The system is evolved until $t = 0.15$ s. Setups for 1-, 2-, and 3-d are provided. The following inputs files and probin files setup the Sod’s problem:

<table>
<thead>
<tr>
<th>inputs file</th>
<th>probin file</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs-test2-x</td>
<td>probin-test2-x</td>
<td>Double rarefaction problem along $x$-direction</td>
</tr>
<tr>
<td>inputs-test2-y</td>
<td>probin-test2-y</td>
<td>Double rarefaction problem along $y$-direction</td>
</tr>
<tr>
<td>inputs-test2-z</td>
<td>probin-test2-z</td>
<td>Double rarefaction problem along $z$-direction</td>
</tr>
</tbody>
</table>

We use a CFL number of 0.8, an initial timestep shrink (`castro.init_shrink`) of 0.1, and the
maximum factor by which the timestep can increase \((\text{castro} . \text{change} . \text{max})\) of 1.05. The PPM solver with the new limiters are used.

![Figure 16.3: Castro solution for the double rarefaction problem run in 3-d, along the \(x\), \(y\), and \(z\) axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line.](image)

Figure 16.3 shows the Castro output, run along all 3 coordinate axes in 3-d, compared to the analytic solution.

The comparison to the analytic solution follows the same procedure as described for the Sod’s problem above. The gnuplot script \texttt{test2.3d.gp} will generate the figure, from the 1-d slices created by \texttt{fextract} named \texttt{test2x.out}, \texttt{test2y.out}, and \texttt{test2z.out}.

### 16.1.1.3 Strong Shock

The strong shock test is the “Test 3” problem described by Toro [12], Chapter 6. In this test, a large pressure jump at the initial interface creates a very strong rightward moving shock, followed very closely by a contact discontinuity. The initial conditions are:

\[
\begin{align*}
\rho_L &= 1 & \rho_R &= 1 \\
u_L &= 0 & u_R &= 0 \\
p_L &= 1000 & p_R &= 0.01
\end{align*}
\] (16.3)

The \texttt{gamma} law equation of state is used with \(\gamma = 1.4\). The system is evolved until \(t = 0.012\) s. Setups for 1-, 2-, and 3-d are provided. The following inputs files and probin files setup the Sod’s problem:
16.1—Hydrodynamics Test Problems

<table>
<thead>
<tr>
<th>inputs file</th>
<th>probin file</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs-test3-x</td>
<td>probin-test3-x</td>
<td>Strong shock problem along x-direction</td>
</tr>
<tr>
<td>inputs-test3-y</td>
<td>probin-test3-y</td>
<td>Strong shock problem along y-direction</td>
</tr>
<tr>
<td>inputs-test3-z</td>
<td>probin-test3-z</td>
<td>Strong shock problem along z-direction</td>
</tr>
</tbody>
</table>

Figure 16.4: Castro solution for the strong shock problem run in 3-d, along the x, y, and z axes. A coarse grid of 32 zones in the direction of propagation, with 2 levels of refinement was used. The analytic solution appears as the red line.

We use a CFL number of 0.9, an initial timestep shrink (castro.init.shrink) of 0.1, and the maximum factor by which the timestep can increase (castro.change.max) of 1.05. The PPM solver with the new limiters are used.

Figure 16.4 shows the Castro output, run along all 3 coordinate axes in 3-d, compared to the analytic solution.

The comparison to the analytic solution follows the same procedure as described for the Sod’s problem above. The gnuplot script test3_3d.gp will generate the figure, from the 1-d slices created by fextract named test3x.out, test3y.out, and test3z.out.

16.1.2 Sedov Problem

The Sedov (or Sedov-Taylor) blast wave is a standard hydrodynamics test problem. A large amount of energy is placed into a very small volume, driving a spherical (or cylindrical in 2-d Cartesian coordinates) blast wave. Analytic solutions were found by Sedov [8].

A cylindrical blast wave (e.g. a point explosion in a 2-d plane) can be modeled in 2-d Cartesian coordinates. A spherical blast wave can be modeled in 1-d spherical, 2-d axisymmetric (cylindrical
Chapter 16. Verification Test Problems

The Castro implementation of the Sedov problem is in Exec/Sedov. A number of different inputs/probin files are provided, corresponding to different Sedov/Castro geometries. The main ones are:

<table>
<thead>
<tr>
<th>inputs file</th>
<th>probin file</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs.1d.sph</td>
<td>probin.1d.sph</td>
<td>Spherical Sedov explosion modeled in 1-d spherical coordinates</td>
</tr>
<tr>
<td>inputs.2d.sph_in_cylcoords</td>
<td>probin.2d.sph_in_cylcoords</td>
<td>Spherical Sedov explosion modeled in 2-d cylindrical (axisymmetric) coordinates</td>
</tr>
<tr>
<td>inputs.2d.cyl_in_cartcoords</td>
<td>probin.2d.cyl_in_cartcoords</td>
<td>Cylindrical Sedov explosion modeled in 2-d Cartesian coordinates</td>
</tr>
<tr>
<td>inputs.3d.sph</td>
<td>probin.3d.sph</td>
<td>Spherical Sedov explosion modeled in 3-d Cartesian coordinates</td>
</tr>
</tbody>
</table>

Table 16.1: Sedov inputs files

In the Sedov problem, the explosion energy, \( E_{\text{exp}} \) (in units of energy, not energy/mass or energy/volume) is to be deposited into a single point, in a medium of uniform ambient density, \( \rho_{\text{ambient}} \), and pressure, \( p_{\text{ambient}} \). Initializing the problem can be difficult because the small volume is typically only a cell in extent. This can lead to grid imprinting in the solution. A standard solution (see for example [6] and the references therein) is to convert the explosion energy into a pressure contained within a certain volume, \( V_{\text{init}} \), of radius \( r_{\text{init}} \) as

\[
p_{\text{init}} = \frac{(\gamma - 1) E_{\text{exp}}}{V_{\text{init}}}. \tag{16.4}
\]

This pressure is then deposited in all of the cells where \( r < r_{\text{init}} \).

To further minimize any grid effects, we do subsampling in each zone: each zone is divided it into \( N_{\text{sub}} \) subzones in each coordinate direction, each subzone is initialized independently, and then the subzones are averaged together (using a volume weighting for spherical or cylindrical/axisymmetric Castro grids) to determine the initial state of the full zone.

For these runs, we use \( \rho_{\text{ambient}} = 1 \), \( p_{\text{ambient}} = 10^{-5} \), \( E_{\text{exp}} = 1 \), \( r_{\text{init}} = 0.01 \), and \( N_{\text{sub}} = 10 \). A base grid with 32 zones in each coordinate direction plus 3 levels of refinement is used (the finest mesh would correspond to 256 zones in a coordinate direction). The domain runs from 0 to 1 in each coordinate direction.

Analysis routines for the Sedov problem are provided in AmrPostprocessing/F_Src/Castro_hydro. These routines will average the Castro solution over angles, using the proper geometric weighting, to produce an average profile as a function of radius. The following routines correspond to the inputs files described above:
16.1—Hydrodynamics Test Problems

<table>
<thead>
<tr>
<th>inputs file</th>
<th>analysis routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs.1d.sph</td>
<td>fsedov1d.f90</td>
</tr>
<tr>
<td>inputs.2d.sph_in_cylcoords</td>
<td>fsedov2d_sph_in_cylcoords.f90</td>
</tr>
<tr>
<td>inputs.2d_cyl_in_cartcoords</td>
<td>fsedov2d_cyl_in_cartcoords.f90</td>
</tr>
<tr>
<td>inputs.3d.sph</td>
<td>fsedov3d_sph.f90</td>
</tr>
</tbody>
</table>

Table 16.2: Analysis routines for Sedov

16.1.2.1 Spherical Blast Wave

A spherical Sedov explosion can be modeled in 1-d spherical, 2-d cylindrical (axisymmetric), or 3-d Cartesian coordinates, using the inputs files described in Table [16.1]. A 1-d radial profile can be extracted using the appropriate `fsedov` routine, as listed in Table [16.2]. For example, to run and process the 2-d cylindrical Sedov explosion, one would do:

1. in Exec/Sedov, build the Castro executable in 2-d
2. run the spherical Sedov problem with Castro in 2-d cylindrical coordinates:
   ```bash
   ./Castro2d.Linux.Intel.Intel.exe inputs.2d.sph_in_cylcoords
   ```
3. build the `fsedov2d_sph_in_cylcoords` tool in AmrPostprocessing/F_Src/Castro_hydro.
4. run `fsedov2d_sph_in_cylcoords` on the Castro output to generate 1-d radial profiles:
   ```bash
   fsedov2d_sph_in_cylcoords.Linux.Intel.exe -s sedov_2d_sph_in_cyl.out -p sedov_2d_sph_in_cyl=plt00246
   ```

A similar procedure can be used for the 1-d and 3-d spherical Sedov explosions (with the output named `sedov_1d_sph.out` and `sedov_3d_sph.out` respectively). Once this is done, the `sedov_sph.gp` gnuplot script can be used to make a plot comparing the 3 solutions to the analytic solution, `spherical_sedov.dat`.

Figure [16.5] shows the comparison of the 3 Castro spherical Sedov explosion simulations to the analytic solution.

16.1.2.2 Cylindrical Blast Wave

16.1.3 Rayleigh-Taylor

2D. Domain size 0.5 by 1.0. 256 by 512 cells, single level calculation. Periodic in x, solid walls on top and bottom in y. Gamma law gas with $\gamma = 1.4$, no reactions. Zero initial velocity. Constant $|\mathbf{g}| = 1$. The density profile is essentially $\rho = 1$ on bottom, $\rho = 2$ on top, but with a perturbation. A single-mode perturbation is constructed as:

$$\tilde{y}(x) = 0.5 + 0.01 \frac{\cos(4\pi x) + \cos(4\pi (L_x - x))}{2}$$  \hspace{1cm} (16.5)

We note that the symmetric form of the cosine is done to ensure that roundoff error does not introduce a left-right asymmetry in the problem. Without this construction, the R-T instability will lose its symmetry as it evolves. This then applied to the interface with a tanh profile to smooth the transition between the high and low density material:

$$\rho(x, y) = 1 + 0.5 \left[ 1 + \tanh \left( \frac{y - \tilde{y}(x)}{0.005} \right) \right]$$  \hspace{1cm} (16.6)
Figure 16.5: Castro solution for the Sedov blast wave problem run in 1-d spherical, 2-d axisymmetric, and 3-d Cartesian coordinates. Each of these geometries produces a spherical Sedov explosion.

Figure 16.6: Castro solution for the Sedov blast wave problem run in 2-d Cartesian coordinates. This corresponds to a cylindrical Sedov explosion.

Hydrostatic pressure with $p = 5.0$ at bottom of domain, assuming $\rho = 1$ on the lower half of the domain, and $\rho = 2$ on the upper half and no density perturbation. We run to $t = 2.5$ with piecewise
16.2—Gravity Test Problems

16.3 Radiation Test Problems

There are two photon radiation solvers in Castro—a gray solver and a multigroup solver. The gray solver follows the algorithm outlined in [3]. We use the notation described in that paper. In particular, the radiation energy equation takes the form of:

\[
\frac{\partial E_R}{\partial t} = \nabla \cdot \left( \frac{c \lambda(E_R)}{\kappa_R} \nabla E_R \right) + \kappa_P (4\sigma T^4 - cE_R) \tag{16.7}
\]

Here, $E_R$ is the radiation energy density, $\kappa_R$ is the Roseland-mean opacity, $\kappa_P$ is the Planck-mean opacity, and $\lambda$ is a quantity $\leq 1/3$ that is subjected to limiting to keep the radiation field causal. Castro allows for $\kappa_R$ and $\kappa_P$ to be set independently as power-laws.
16.3.1 Light Front

The light front problem tests the ability of the radiation solver to operate in the free-streaming limit. A radiation front is established by initializing one end of the computational domain with a finite radiation field, and zero radiation field everywhere else. The speed of propagation of the radiation front is kept in check by the flux-limiters, to prevent it from exceeding $c$.

16.3.2 Diffusion of a Gaussian Pulse

The diffusion of a Gaussian pulse problem tests the diffusion term in the radiation energy equation. The radiation energy density is initialized at time $t = t_0$ to a Gaussian distribution:

$$E_R = (E_R)_0 \exp \left\{-\frac{1}{4Dt_0} |r - r_0|^2 \right\}.$$ (16.8)

As the radiation diffuses, the overall distribution will remain Gaussian, with the time-dependent solution of:

$$E_R = (E_R)_0 \frac{t_0}{t_0 + t} \exp \left\{-\frac{1}{4D(t_0 + t)} |r - r_0|^2 \right\}.$$ (16.9)

16.3.3 Radiation Source Problem

The radiation source problem tests the coupling between the radiation field and the gas energy through the radiation source term. The problem begins with the radiation field and gas temperature out of equilibrium. If the gas is too cool, then the radiation field will heat it. If the gas is too hot, then it will radiate and cool. In each case, the gas energy and radiation field will evolve until thermal equilibrium is achieved.

Our implementation of this problem follows that of [11].

16.3.4 Radiating Sphere

The radiating sphere is a multigroup radiation test problem. A hot sphere is centered at the origin in a spherical geometry. The spectrum from this sphere follows a Planck distribution. The ambient medium is at a much lower temperature. A frequency-dependent opacity makes the domain optically thin for high frequencies and optically thick for low frequency. At long times, the solution will be a combination of the blackbody radiation from the ambient medium plus the radiation that propagated from the hot sphere. An analytic solution exists [2] which gives the radiation energy as a function of energy group at a specified time and distance from the radiating sphere.

Our implementation of this problem is in Exec/RadSphere and follows that of [11]. The routine that computes the analytic solution is provided as analytic.f90.

16.4 Regression Testing

An automated regression test suite for Castro (or any BoxLib-based code) written in Python exists in Parallel/util/regtests/ as test.py. The test suite consists of a set of problem definitions.
Figure 16.8: Castro solution for radiating source test problem. Heating and cooling solutions are shown as a function of time, compared to the analytic solution. The gray photon solver was used.

Figure 16.9: Castro solution for radiating sphere problem, showing the radiation energy density as a function of energy group. This test was run with 64 photon energy groups.

(the Castro problem + their inputs/probin files, etc.). When the suite is run the first time, the plotfiles created at the end of each problem’s execution is stored as a benchmark. After this
Chapter 16. Verification Test Problems

Figure 16.10: Main test suite results page. Each row indicates a single test suite run, arranged by date, and each column indicates a different test problem. Note: this page is from the Maestro code, but a Castro test suite run will produce similar output.

Initialization, each subsequent run of the test suite compares the current output of the code, level-by-level and zone-by-zone to the stored benchmarks (using the fcompare.f90 routine in from AmrPostprocessing/F_Src/). Any differences are flagged as errors. A web page report is generated by the test suite and provides a history of the regression testing. Single-processor and parallel test problems, compilation tests, and testing restarting from a checkpoint file are supported.

16.4.1 Test Suite Inputs File

The inputs file for the test suite separates the problems into blocks. The header of a problem block has the form [problem-name]. Beneath each problem block, there are a number of options set for each problem. A separate heading, [main], is used for the suite-wide options.
An example of the `main` block from `Castro-tests.ini` is:

```ini
[main]
sourceDir = /work/zingale/test/Castro/
testTopDir = /work/zingale/test/Castro/
compareToolDir = /work/zingale/test/AmrPostprocessing/F\_Src/
helmeosDir = /work/zingale/test/Castro/EOS/helmeos/

sourceTree = Parallel

COMP = gcc
FCOMP = gfortran

suiteName = Castro

MPIcommand = mpiexec -host @host@ -n @nprocs@ @command@
MPIhost = node1
```

The first group of options define the necessary paths. Here, `sourceDir` points to the top-level directory, which is expected to contain the `Parallel` and `fParallel` subdirectories. `testTopDir` refers to the directory that the suite should use as its root directory for output—usually this is the same as `sourceDir`. The `fcompare.f90` comparison tool is expected to be found in `compareToolDir`. Finally, `helmeosDir` lists the path to the `helm_table.dat` file used by the general stellar equation of state.

Next, we set `sourceTree` to `Parallel`, indicating that Castro is built using the C++ BoxLib framework. (Maestro uses the Fortran 95 BoxLib framework, and therefore would set `sourceTree` to `fParallel`). This option tells the test suite what build system to use.

`COMP` and `FCOMP` tell the test suite which C++ and Fortran compilers to use. These override what is listed in any `GNUnmakefile` to ensure that the compiler stays consistent in the tests.

The `suiteName` option simply tells the test suite what name to prefix to the output directories. It does not need to match the program name.

Finally, `MPIcommand` lists the generic manner in which to run an MPI program on the target system. The string `@host@` in the `MPIcommand` will be substituted by the `MPIhost` string by the test suite. Similarly the `@nprocs@` string will be substituted by the number of processors, which is set on a problem-by-problem basis. Finally, the `MPIcommand` should include the string `@command@`, which is where the Castro executable and inputs file will be substituted. For single processor runs, these options are ignored.

Each problem to be run by the test suite gets its own block. For example, a Sod’s problem test might look like:

```ini
[Sod-x]
buildDir = Castro/Exec/Sod/
inputFile = inputs-sod-x
probinFile = probin-sod-x
needs_helmeos = 0
dim = 3
restartTest = 0
```
useMPI = 0
compileTest = 0
doVis = 0

Here Sod-x contained inside the [] is the name of the problem, as the test suite will refer to it. buildDir is the path beneath sourceDir where the make command should be executed. The inputs file and probin file are given by inputFile and probinFile, which should be relative to the buildDir. The dimensionality is specified by dim.

A number of options are available:

- If the general stellar equation of state is used, then needs_helmeos should be set to 1 to ensure that the EOS table is copied into the run directory.
- If the test is to be run in parallel, the useMPI should be 1 and numprocs should give the number of processors.
- To test the compilation of the problem only (and skip running), set compileTest to 1.
- To test the ability of the code to restart, set restartTest to 1. Also set restartFileNum to the number of the checkpoint file to restart from. The suite will run the problem as usual and then restart from the specified checkpoint and run to completion again. The output from the initial run will then be compared to the output from the restart. In a restart test, there is no stored benchmark.
- To add a simple visualization to the test suite webpage, set doVis to 1, and set visVar to the name of the plotfile variable to visualize. An image of that field from the last plotfile will be appended to the problem’s test webpage.

16.4.2 Initializing the Test Suite

The first time you run the test suite there are no benchmark files to compare to. Once you generate an inputs file, as described above, you would simply run the suite as:

```
./test.py --make_benchmarks "initial run" ./Castro_tests.ini
```

The string following --make_benchmarks is simply a comment that will be added to the web report. This command creates three output directories, using the suiteName as the prefix.

- suiteName-tests is where the tests are run. Each time the test suite is run, a subdirectory, based on the date, is created, with a subdirectory for each test. All the files necessary to run the test are copied into the test subdirectory.
- suiteName-web is where the web-based reports for the test are generated. The master webpage is suiteName-web/index.html.
- suiteName-benchmarks is where the test benchmark files are stored. This are used for comparison to the current output.

16.4.3 Regular Use

Once the initial benchmarks are created, you can compare the current version of the code to the stored results by simply doing:
Figure 16.11: The test suite output for a single day’s run. Each row indicates a separate test, showing whether they passed or failed. Clicking on the test name will give more information about that particular test on that day. Note: this page is from the Maestro code, but a Castro test suite run will produce similar output.

```
./test.py ./Castro_tests.ini
```

This will do a CVS update, generate ChangeLog files listing all of the CVS comments for the code, build the test comparison tools, and then loop over each test, building and running the executable and comparing the output to the benchmarks.

Upon completion of all the runs, a web page for this invocation of the test suite will be generated (see figure 16.11), as well as pages showing the details for each of the problems run. Test failures indicate that the current output does not match the stored benchmarks.
16.4.4 Updating Benchmarks

A test failure means that the current version of the code gives a different answer than the stored benchmark. A test can fail either because a bug was introduced into the code or a bug was fixed or new feature introduced.

If a bug was introduced into the code recently, then by examining the test history you can determine the time period in which the bug was introduced. The ChangeLog.BoxLib and ChangeLog.Castro files linked to on each test date’s webpage will list all the changes committed to git up to that point, which is useful for tracking down the bug. Once the bug is fixed, rerunning the suite should generate a ‘pass’.

If a bug was fixed or a new feature was introduced, and you are confident that the latest output is correct, then you can tell the test suite to update the benchmarks. If you want to do this for all the test problems, you would do:

```
./test.py --make_benchmarks "X bug fixed" ./Castro_tests.ini
```

where the string after “--make_benchmarks” is a note that is listed on the regression suite web page describing the reason for the benchmark update. Subsequent runs of the test suite will use the new benchmarks. If you only want to update the benchmarks of a single test, then you can use the “--single_test test” flag on the commandline, where test is the name of the test to update.
17.1 Automatic Restarting and Archiving of Data

The submission script `jaguar.run` and shell script `process.jaguar` in `Exec/job_scripts/` are designed to allow you to run CASTRO with minimal interaction, while being assured that the data is archived to HPSS on the NCCS machines.

To use the scripts, first create a directory in HPSS that has the same name as the directory on lustre you are running in (just the directory name, not the full path). E.g. if you are running in a directory called `Castro_run`, then do:

```
hsi
mkdir Castro_run
```

The script `process.jaguar` is called from `jaguar.run` and will run in the background and continually wait until checkpoint or plotfiles are created (actually, it always leaves the most recent one alone, since data may still be written to it, so it waits until there are more than one in the directory).

Then the script will use `htar` to archive the plotfiles and checkpoints to HPSS. If the `htar` command was successful, then the plotfiles are copied into a `plotfile/` subdirectory. This is actually important, since you don’t want to try archiving the data a second time and overwriting the stored copy, especially if a purge took place.

Additionally, if you have the path to the `ftime` executable set in the script (`ftime.f90` lives in `AmrPostprocessing/F.Src/`), then the script will create a file called `ftime.out` that lists the name of the plotfile and the corresponding simulation time.

Finally, right when the job is submitted, the script will tar up all of the diagnostic files created by `diag.f90` and archive them on HPSS. The `.tar` file is given a name that contains the date-string
to allow multiple archives to co-exist.

This really allows you to run the job and have all of the data stored long term automatically. This way you don’t need to worry about filesystem purges. It seems to work well.

Also, the `jaguar.run` submission script has code in it that will look at the most recently generated checkpoint files, make sure that they were written out correctly (it looks to see if there is a Header file, since that is the last thing written), and automatically set the `--restart` flag on the run line to restart from the most recent checkpoint file.

This allows you to job-chain a bunch of submission and have them wait until the previous job finished and then automatically queue up:

```
qsub -W depend=afterany:<JOB-ID> <QSUB SCRIPT>
```

where `<JOB-ID>` is the id number of the job that must complete before the new submission runs and `<QSUB SCRIPT>` is the submission script (e.g. `jaguar.run`). This way you can queue up a bunch of runs and literally leave things alone and it will restart from the right place automatically and store the data as it is generated.

When `process.jaguar` is running, it creates a lockfile (called `process.pid`) that ensures that only one instance of the script is running at any one time. Sometimes if the machine crashes, the `process.pid` file will be left behind, in which case, the script aborts. Just delete that if you know the script is not running.

Analogous scripts exist for running on Atlas, with the same general procedure. The command to chain a job on atlas is:

```
msub -l depend=<JOB-ID> <MSUB SCRIPT>
```

where, again, `<JOB-ID>` is the id number of the job that must complete before the current submission runs and `<MSUB SCRIPT>` is the job submission script (e.g. `atlas.run`).
With the new flurry of interest in the scaling behavior of CASTRO, we’ve run some tests on Franklin.

18.1 Sod Problem in 3D

We ran the Sod problem (inputs-test2-x) for 10 time steps with max_level = 0 on 4, 32, 256, 2048 and 16384 processors. This is a gamma-law gas with no reactions and one species. On 4 processors the base grid was 512x128x128; for each factor of 8 increase in processors we doubled the number of cells in each direction. We set max_grid_size = 64 for all cases, resulting in 256 grids per processor. (Thus for perfect scaling we would expect this plot to be flat.)

18.2 White Dwarf in 3D

This test was performed on 6/1/09 using ScalingTest/inputs.nog, inputs.g3, inputs.g5, inputs.g3.2levels, and inputs.g5.2levels on jaguarpf. Here we interpolated a 1D initial model of a white dwarf onto a 3D Cartesian grid and ran 5 time steps. This problem used the Helmholtz EOS with no reactions, no thermal diffusion, and three species.

We ran this problem on 8, 64, 512, 4096, 13824, 32768, and 64000 processors. The base grid size, respectively, was $128^3$, $256^3$, $512^3$, $1024^3$, $1536^3$, $2048^3$, and $2560^3$. We ran with no gravity (castro.do_grav = 0), gravity type 3 (Poisson solver using multigrid) and gravity type 5 (1D integration of the averaged radial density to define a monopole approximation for gravity), each with max_level=0. We also ran with gravity types 3 and 5 with max_level=1. The fine grid structure had the exact same grid structure as the coarse level and was centered in the domain. For all cases
max_grid_size = 64, so there was 1 processor per grid per level. Thus for perfect scaling we would expect this plot to be flat.

Figure 18.1: Scaling behavior of Sod problem on franklin.nersc.gov

Figure 18.2: Scaling behavior of ScalingTest problem on jaguarpf.ccs.ornl.gov
CHAPTER 19

Suggestions, Warnings, and Gotchas

1. 2/12/09: If running on more than 64 processors on pleiades.ucsc.edu, you must set

   amr.plot.nfiles = NUMBER
   amr.checkpoint.nfiles = NUMBER

   where NUMBER is a number greater than or equal to the number of processors (putting a large integer here is fine). The default value for plot.nfiles and checkpoint.nfiles is 64, so if you are running on 64 or fewer processors you are ok.

   There is a problem with writing to the file system on pleiades. We have not yet been able to come up with a fix but this appears to be a good solution for now.

2. 2/2/09: Intel version 10.1.015 for Fortran has a known bug when run in non-DEBUG mode. We’ve tried the new version 11 as well and parts of the code no longer compile. So beware the Intel compilers. Best suggestion at this point is to return to version 9.

3. 1/25/09: on franklin.nersc.gov, the PathScale compiler with full optimization appears to have bug. Please use PGI for now, or reduce the optimization from -O to -O1 with PathScale.

4. 1/25/09: Very rough estimate on franklin-- for a code with 1 species, one can run a $256^3$ calculation on 16 processors but not 8. Keep this in mind as you choose the number of processors for a given job. Suggested settings for large 3-d runs: blocking.factor = 16, max.grid.size = 32.
19.1 Compilers

This is a brief collection of our experiences with compilers on various machines. This is not an exhaustive list.

19.1.1 Those that compile...

<table>
<thead>
<tr>
<th>Machine</th>
<th>Compiler</th>
<th>Modules</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>hopper</td>
<td>PathScale</td>
<td>PrgEnv-pathscale/3.1.61(default)</td>
<td>Date: 1/12/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pathscale/4.0.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PGI</td>
<td>PrgEnv-pgi/3.1.61(default)</td>
<td>Date: 1/12/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgi/11.7.0(default)</td>
<td></td>
</tr>
<tr>
<td>titan &amp; blue waters</td>
<td>Cray</td>
<td>PrgEnv-cray/4.1.20(default)</td>
<td>Date: 1/23/13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cce/8.1.2</td>
<td></td>
</tr>
</tbody>
</table>

*Note: Cray compilers version 8.1.2 need the `-hnopgas_runtime` flag to be set.*

19.1.2 Those that don't compile...

<table>
<thead>
<tr>
<th>Machine</th>
<th>Compiler</th>
<th>Modules</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>hopper</td>
<td>PathScale</td>
<td>PrgEnv-pathscale/3.1.61(default)</td>
<td>Date: 1/12/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pathscale/3.2.99(default)</td>
<td></td>
</tr>
<tr>
<td>titan &amp; blue waters</td>
<td>Cray</td>
<td>PrgEnv-cray/4.1.20(default)</td>
<td>Date: 1/23/13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cce/8.1.1(default)</td>
<td></td>
</tr>
</tbody>
</table>
References


