FLASH User’s Guide

Version 4.2
February 2014 (last updated February 7, 2014)

Flash Center for Computational Science
University of Chicago
License

0.1 Acknowledgments in Publication

All publications resulting from the use of the FLASH Code must acknowledge the Flash Center. Addition of the following text to the paper acknowledgments will be sufficient.

"The software used in this work was in part developed by the DOE NNSA-ASC OASCR Flash Center at the University of Chicago."

The users should visit the bibliography hosted at flash.uchicago.edu/site/publications/flash_pubs.shtml to find the relevant papers to cite in their work.

This is a summary of the rules governing the dissemination of the "Flash Code" by the Flash Center for Computational Science to users outside the Center, and constitutes the License Agreement for users of the Flash Code. Users are responsible for following all of the applicable rules described below.

0.2 Full License Agreement

Below is a summary of the rules governing the dissemination of the "FLASH Code" by the Flash Center for Computational Science to users outside the Center, and constitutes the License Agreement for users of the FLASH Code. Users are responsible for following all of the applicable rules described below.

• Public Release. Publicly released versions of the FLASH Code are available via the Center’s website. We expect to include any external contributions to the Code in public releases that occur after the end of a negotiated time.

• Decision Process. At present, release of the FLASH Code to users not located at the University of Chicago or at Argonne National Laboratory is governed solely by the Center’s Director and Management Committee; decisions related to public release of the FLASH Code will be made in the same manner.

• License and Distribution Rights. The University of Chicago owns the copyright to all Code developed by the members of the Flash Center at the University of Chicago. External contributors may choose to be included in the Copyright Assertion. The FLASH Code, or any part of the code, can only be released and distributed by the Flash Center; individual users of the FLASH Code are not free to re-distribute the FLASH Code, or any of its components, outside the Center. All users of the FLASH Code must sign a hardcopy version of this License Agreement and send it to the Center. Distribution of the FLASH Code can only occur once we receive a signed License Agreement.

• Modifications and Acknowledgments. Users may make modifications to the FLASH Code, and they are encouraged to send such modifications to the Center. Users are not free to distribute the FLASH Code to others, as noted in Section 3 above. As resources permit, we will incorporate such modifications in subsequent releases of the FLASH Code, and we will acknowledge these external contributions. Note that modifications that do not make it into an officially-released version of the FLASH Code will not be supported by us.
If a user modifies a copy or copies of the FLASH Code or any portion of it, thus forming a work based on the FLASH Code, to be included in a FLASH release it must meet the following conditions:

- a) The software must carry prominent notices stating that the user changed specified portions of the FLASH Code. This will also assist us in clearly identifying the portions of the FLASH Code that the user has contributed.
- b) The software must display the following acknowledgement: "This product includes software developed by and/or derived from the Flash Center for Computational Science (http://flash.uchicago.edu) to which the U.S. Government retains certain rights."
- c) The FLASH Code header section, which describes the origins of the FLASH Code and of its components, must remain intact, and should be included in all modified versions of the code. Furthermore, all publications resulting from the use of the FLASH Code, or any modified version or portion of the FLASH Code, must acknowledge the Flash Center for Computational Science; addition of the following text to the paper acknowledgments will be sufficient: "The software used in this work was developed in part by the DOE NNSA ASC- and DOE Office of Science ASCR-supported Flash Center for Computational Science at the University of Chicago." The Code header provides information on software that has been utilized as part of the FLASH development effort (such as the AMR). The Center website includes a list of key scientific journal references for the FLASH Code. We request that such references be included in the reference section of any papers based on the FLASH Code.

- Commercial Use. All users interested in commercial use of the FLASH Code must obtain prior written approval from the Director of the Center. Use of the FLASH Code, or any modification thereof, for commercial purposes is not permitted otherwise.

- Bug Fixes and New Releases. As part of the FLASH Code dissemination process, the Center has set up and will maintain as part of its website mechanisms for announcing new FLASH Code releases, collecting requests for FLASH Code use, and collecting and disseminating relevant documentation. We support the user community through mailing lists and by providing a bug report facility.

- User Feedback. The Center requests that all users of the FLASH Code notify the Center about all publications that incorporate results based on the use of the code, or modified versions of the code or its components. All such information can be sent to infoflash.uchicago.edu.

- Disclaimer. The FLASH Code was prepared, in part, as an account of work sponsored by an agency of the United States Government. THE FLASH CODE IS PROVIDED "AS IS" AND NEITHER THE UNITED STATES, NOR THE UNIVERSITY OF CHICAGO, NOR ANY CONTRIBUTORS TO THE FLASH CODE, NOR ANY OF THEIR EMPLOYEES OR CONTRACTORS, MAKES ANY WARRANTY, EXPRESS OR IMPLIED (INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE), OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

IN NO EVENT WILL THE UNITED STATES, THE UNIVERSITY OF CHICAGO OR ANY CONTRIBUTORS TO THE FLASH CODE BE LIABLE FOR ANY DAMAGES, INCLUDING DIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM EXERCISE OF THIS LICENSE AGREEMENT OR THE USE OF THE SOFTWARE.
Acknowledgments

The Flash Center for Computational Science at the University of Chicago is supported by the DOE NNSA-ASC and NSF. Some of the test calculations described here were performed on machines at LLNL, LANL, San Diego Supercomputing Center, and ANL. The current contributors to the code from the Center include:

Sean Couch, Norbert Flocke, Dongwook Lee, Petros Tzeferacos, and Klaus Weide.
Considerable external and past contributors include:


PARAMESH was developed under NASA Contracts/Grants NAG5-2652 with George Mason University; NAS5-32350 with Raytheon/STX; NAG5-6029 and NAG5-10026 with Drexel University; NAG5-9016 with the University of Chicago; and NCC5-494 with the GEST Institute. For information on PARAMESH please contact its main developers, Peter MacNeice (macneice@alfven.gsfc.nasa.gov) and Kevin Olson (olson@physics.drexel.edu) and see the website http://www.physics.drexel.edu/~olson/paramesh-doc/Users_manual/amr.html.
## Contents

<table>
<thead>
<tr>
<th>License</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 Acknowledgments in Publication</td>
<td>i</td>
</tr>
<tr>
<td>0.2 Full License Agreement</td>
<td>i</td>
</tr>
</tbody>
</table>

### 1 Introduction

| 1.1 What’s New in FLASH4                    | 1 |
| 1.2 External Contributions                 | 4 |
| 1.3 Known Issues in This Release           | 4 |
| 1.4 About the User's Guide                 | 5 |

### I Getting Started

<table>
<thead>
<tr>
<th>2 Quick Start</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 System requirements</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Unpacking and configuring FLASH for quick start</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Running FLASH</td>
<td>12</td>
</tr>
</tbody>
</table>

### 3 Setting Up New Problems

| 3.1 Creating a Config file                  | 17 |
| 3.2 Creating a Makefile                     | 18 |
| 3.3 Creating a Simulation_data.F90         | 19 |
| 3.4 Creating a Simulation_init.F90         | 19 |
| 3.5 Creating a Simulation_initBlock.F90    | 21 |
| 3.6 The runtime parameter file (flash.par) | 25 |
| 3.7 Running your simulation                | 27 |

### II The FLASH Software System

<table>
<thead>
<tr>
<th>4 Overview of FLASH architecture</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 FLASH Inheritance</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Unit Architecture</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1 Stub Implementations</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2 Subunits</td>
<td>35</td>
</tr>
<tr>
<td>4.2.3 Unit Data Modules, _init, and _finalize routines</td>
<td>36</td>
</tr>
<tr>
<td>4.2.4 Private Routines: kernels and helpers</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Unit Test Framework</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5 The FLASH configuration script (setup)</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Setup Arguments</td>
<td>40</td>
</tr>
<tr>
<td>5.2 Comprehensive List of Setup Arguments</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Using Shortcuts</td>
<td>54</td>
</tr>
</tbody>
</table>
CONTENTS

5.4 Setup Variables and Preprocessing Config Files .................................................. 54
5.5 Config Files ........................................................................................................... 56
  5.5.1 Configuration file syntax ................................................................................. 56
  5.5.2 Configuration directives ................................................................................. 57
5.6 Creating a Site-specific Makefile ........................................................................... 65
5.7 Files Created During the setup Process ................................................................. 66
  5.7.1 Informational files ........................................................................................... 66
  5.7.2 Code generated by the setup call ..................................................................... 67
  5.7.3 Makefiles generated by setup .......................................................................... 67
5.8 Setup a hybrid MPI+OpenMP FLASH application .................................................. 68
5.9 Setup a FLASH+Chombo application ..................................................................... 70
  5.9.1 Overview ......................................................................................................... 70
  5.9.2 Build procedure ............................................................................................... 70

6 The Flash.h file  ........................................................................................................ 75
  6.1 UNK, FACE(XYZ) Dimensions ............................................................................ 75
  6.2 Property Variables, Species and Mass Scalars .................................................... 76
  6.3 Fluxes .................................................................................................................... 77
  6.4 Scratch Vars ......................................................................................................... 77
  6.5 Fluid Variables Example ....................................................................................... 78
  6.6 Particles ................................................................................................................ 79
    6.6.1 Particles Types ............................................................................................... 79
    6.6.2 Particles Properties ....................................................................................... 79
  6.7 Non-Replicated Variable Arrays .......................................................................... 80
    6.7.1 Per-Array Macros .......................................................................................... 80
    6.7.2 Array Partitioning Macros ............................................................................. 80
    6.7.3 Example .......................................................................................................... 81
  6.8 Other Preprocessor Symbols ................................................................................. 82

III Driver Unit ............................................................................................................. 83

7 Driver Unit ................................................................................................................ 85
  7.1 Driver Routines ..................................................................................................... 85
    7.1.1 Driver_initFlash ............................................................................................ 86
    7.1.2 Driver_evolveFlash ....................................................................................... 86
    7.1.3 Driver_finalizeFlash ..................................................................................... 88
    7.1.4 Driver accessor functions .............................................................................. 88

IV Infrastructure Units ............................................................................................... 91

8 Grid Unit .................................................................................................................. 93
  8.1 Overview ............................................................................................................... 95
  8.2 GridMain Data Structures ................................................................................... 97
  8.3 Computational Domain ....................................................................................... 98
  8.4 Boundary Conditions ......................................................................................... 99
    8.4.1 Boundary Condition Types ............................................................................ 99
    8.4.2 Boundary Conditions at Obstacles ............................................................... 100
    8.4.3 Implementing Boundary Conditions ............................................................. 101
  8.5 Uniform Grid ....................................................................................................... 103
    8.5.1 FIXEDBLOCKSIZE Mode ............................................................................. 103
    8.5.2 NONFIXEDBLOCKSIZE mode ................................................................. 104
  8.6 Adaptive Mesh Refinement (AMR) Grid with Paramesh ...................................... 104
    8.6.1 Additional Data Structures .......................................................................... 106
## CONTENTS

8.6.2 Grid Interpolation ............................................ 107
8.6.3 Refinement .................................................. 108
8.7 Chombo .......................................................... 111
  8.7.1 Using Chombo in a UG configuration .................... 111
  8.7.2 Using Chombo in a AMR configuration .................. 111
8.8 GridMain Usage .................................................. 113
8.9 GridParticles .................................................. 114
  8.9.1 GridParticlesMove ......................................... 115
  8.9.2 GridParticlesMapToMesh .................................. 117
8.10 GridSolvers ................................................... 121
  8.10.1 Pfft ....................................................... 121
  8.10.2 Poisson equation ........................................ 124
  8.10.3 Using the Poisson solvers .............................. 137
  8.10.4 HYPRE .................................................... 142
8.11 Grid Geometry .................................................. 145
  8.11.1 Understanding Curvilinear .............................. 145
  8.11.2 Choosing a Geometry .................................... 147
  8.11.3 Geometry Information in Code ......................... 147
  8.11.4 Available Geometries .................................... 147
  8.11.5 Conservative Prolongation/Restriction on Non-Cartesian Grids ................. 150
8.12 Unit Test ....................................................... 151
9 IO Unit .................................................................. 153
  9.1 IO Implementations ............................................ 155
  9.2 Output Files ................................................... 157
    9.2.1 Checkpoint files - Restarting a Simulation ............ 157
    9.2.2 Plotfiles ................................................ 159
    9.2.3 Particle files ........................................... 160
    9.2.4 Integrated Grid Quantities – flash.dat ................. 161
    9.2.5 General Runtime Parameters ............................ 162
  9.3 Restarts and Runtime Parameters .......................... 163
  9.4 Output Scalars ................................................ 163
  9.5 Output User-defined Arrays ................................ 163
  9.6 Output Scratch Variables ................................... 164
  9.7 Face-Centered Data ........................................... 164
  9.8 Output Filenames ............................................. 164
  9.9 Output Formats ................................................ 165
    9.9.1 HDF5 .................................................... 165
    9.9.2 Parallel-NetCDF ......................................... 171
    9.9.3 Direct IO ................................................ 171
    9.9.4 Output Side Effects ..................................... 171
  9.10 Working with Output Files ................................ 172
  9.11 Unit Test ....................................................... 172
  9.12 Chombo ........................................................ 173
  9.13 Derived data type I/O ....................................... 173

10 Runtime Parameters Unit ....................................... 175
  10.1 Defining Runtime Parameters .............................. 175
  10.2 Identifying Valid Runtime Parameters ................. 175
  10.3 Routine Descriptions ........................................ 176
  10.4 Example Usage ................................................ 177
## 11 Multispecies Unit

- **11.1 Defining Species** ............................................ 179
- **11.2 Initializing Species Information in `Simulation_initSpecies`** ........ 180
- **11.3 Specifying Constituent Elements of a Species** ..................... 182
- **11.4 Alternative Method for Defining Species** .......................... 182
- **11.5 Routine Descriptions** ........................................ 183
- **11.6 Example Usage** ............................................... 185
- **11.7 Unit Test** .................................................... 185

## 12 Physical Constants Unit

- **12.1 Available Constants and Units** ................................ 188
- **12.2 Applicable Runtime Parameters** ................................ 188
- **12.3 Routine Descriptions** ........................................ 188
- **12.4 Unit Test** .................................................... 189

## V Physics Units

### 13 3T Capabilities for Simulation of HEDP Experiments

- **13.1 Introduction** .................................................. 221
- **13.2 Gamma Law and Multigamma** ................................ 222
- **13.3 Helmholtz** .................................................... 223
- **13.4 Multitemperature extension for Eos** ................................ 225
- **13.5 Non-ideal MHD** ................................................ 225
- **13.6 Usage** .......................................................... 228
- **13.7 Usage Information** ............................................ 228
- **13.8 Runtime Parameters** .......................................... 229
- **13.9 Algorithm: The Unsplit Staggered Mesh Solver** .................... 235
- **13.10 Algorithm: The Eight-wave Solver** ................................ 235
- **13.11 Non-ideal MHD** ................................................ 236
- **13.12 Unit Test** ..................................................... 236

## 14 Hydrodynamics Units

- **14.1 Gas hydrodynamics** ........................................... 199
  - **14.1.1 Usage** .................................................... 199
  - **14.1.2 The piecewise-parabolic method (PPM)** ...................... 199
  - **14.1.3 The unsplit hydro solver** ................................ 200
  - **14.1.4 Multitemperature extension for Hydro** ...................... 205
  - **14.1.5 Chombo compatible Hydro** ................................ 208
- **14.2 Relativistic hydrodynamics (RHD)** ................................ 209
  - **14.2.1 Overview** ............................................... 209
  - **14.2.2 Equations** ............................................... 209
  - **14.2.3 Relativistic Equation of State** ............................ 210
  - **14.2.4 Additional Runtime Parameter** ............................ 210
- **14.3 Magnetohydrodynamics (MHD)** ................................ 210
  - **14.3.1 Description** ............................................. 210
  - **14.3.2 Usage** .................................................... 211
  - **14.3.3 Algorithm: The Unsplit Staggered Mesh Solver** ............ 212
  - **14.3.4 Algorithm: The Eight-wave Solver** ....................... 217

## 15 Equation of State Unit

- **15.1 Introduction** .................................................. 221
- **15.2 Gamma Law and Multigamma** ................................ 222
  - **15.2.1 Ideal Gamma Law for Relativistic Hydrodynamics** ........... 223
- **15.3 Helmholtz** .................................................... 223
- **15.4 Multitemperature extension for Eos** ................................ 225
  - **15.4.1 Gamma** ................................................... 225
  - **15.4.2 Multigamma** ............................................. 227
  - **15.4.3 Tabulated** ................................................. 227
  - **15.4.4 Multitype** ................................................ 227
- **15.5 Usage** .......................................................... 228
  - **15.5.1 Initialization** ............................................ 228
  - **15.5.2 Runtime Parameters** ..................................... 228
  - **15.5.3 Direct and Wrapped Calls** ................................ 228
- **15.6 Unit Test** ..................................................... 229
## 16 Local Source Terms

16.1 Burn Unit .................................................. 232
  16.1.1 Algorithms ........................................ 232
  16.1.2 Reaction networks ............................ 233
  16.1.3 Detecting shocks .............................. 237
  16.1.4 Energy generation rates and reaction rates 237
  16.1.5 Temperature-based timestep limiting ..... 238
16.2 Ionization Unit ........................................... 238
  16.2.1 Algorithms ........................................ 239
  16.2.2 Usage ............................................. 239
16.3 Stir Unit .................................................. 239
  16.3.1 Stir Unit: Generate Implementation .... 240
  16.3.2 Stir Unit: FromFile Implementation .. 240
  16.3.3 Using the StirFromFile Unit ............ 241
  16.3.4 Stirring Unit Test ............................ 244
16.4 Energy Deposition Unit ................................. 244
  16.4.1 Ray Tracing in the Geometric Optics Limit 244
  16.4.2 Laser Power Deposition ..................... 245
  16.4.3 Algorithmic Implementation of the Ray Tracing 246
  16.4.4 Setting up the Laser Pulse ................. 251
  16.4.5 Setting up the Laser Beam ................. 252
  16.4.6 Setting up the Rays ........................... 255
  16.4.7 3D Laser Ray Tracing in 2D Cylindrical Symmetry 259
  16.4.8 Synchronous and Asynchronous Ray Tracing 265
  16.4.9 Usage ............................................. 269
  16.4.10 Unit Tests .................................... 273
16.5 Heatexchange ............................................. 279
  16.5.1 Spitzer Heat Exchange ....................... 280
16.6 Flame ................................................... 282
  16.6.1 Reaction-Diffusion Forms ..................... 282
  16.6.2 Unit Structure ................................ 283
16.7 Turbulence Measurement .............................. 284

## 17 Diffusive Terms

17.1 Diffuse Unit ............................................. 285
  17.1.1 Diffuse Flux-Based Implementations ...... 286
  17.1.2 General Implicit Diffusion Solver ...... 286
  17.1.3 Flux Limiters .................................. 289
  17.1.4 Stand-Alone Electron Thermal Conduction 291

## 18 Gravity Unit

18.1 Introduction ............................................. 293
18.2 Externally Applied Fields ........................... 294
  18.2.1 Constant Gravitational Field ....... 294
  18.2.2 Plane-parallel Gravitational field ...... 294
  18.2.3 Gravitational Field of a Point Mass ...... 294
18.3 Self-gravity ............................................. 294
  18.3.1 Coupling Gravity with Hydrodynamics 295
  18.3.2 Tree Gravity ................................ 295
18.4 Usage .................................................. 297
  18.4.1 Tree Gravity Unit Usage ..................... 297
18.5 Unit Tests ............................................... 298
### 19 Particles Unit

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Time Integration</td>
<td>301</td>
</tr>
<tr>
<td>19.1.1 Active Particles (Massive)</td>
<td>301</td>
</tr>
<tr>
<td>19.1.2 Charged Particles - Hybrid PIC</td>
<td>303</td>
</tr>
<tr>
<td>19.1.3 Passive Particles</td>
<td>305</td>
</tr>
<tr>
<td>19.2 Mesh/Particle Mapping</td>
<td>309</td>
</tr>
<tr>
<td>19.2.1 Quadratic Mesh Mapping</td>
<td>310</td>
</tr>
<tr>
<td>19.2.2 Cloud in Cell Mapping</td>
<td>312</td>
</tr>
<tr>
<td>19.3 Using the Particles Unit</td>
<td>312</td>
</tr>
<tr>
<td>19.3.1 Particles Runtime Parameters</td>
<td>314</td>
</tr>
<tr>
<td>19.3.2 Particle Attributes</td>
<td>314</td>
</tr>
<tr>
<td>19.3.3 Particle I/O</td>
<td>315</td>
</tr>
<tr>
<td>19.3.4 Unit Tests</td>
<td>315</td>
</tr>
<tr>
<td>19.4 Sink Particles</td>
<td>316</td>
</tr>
<tr>
<td>19.4.1 Basics of Sink Particles</td>
<td>316</td>
</tr>
<tr>
<td>19.4.2 Using the Sink Particle Unit</td>
<td>316</td>
</tr>
<tr>
<td>19.4.3 The Sink Particle Method</td>
<td>317</td>
</tr>
<tr>
<td>19.4.4 Sink Particle Unit Test</td>
<td>318</td>
</tr>
</tbody>
</table>

### 20 Cosmology Unit

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Algorithms and Equations</td>
<td>321</td>
</tr>
<tr>
<td>20.2 Using the Cosmology unit</td>
<td>323</td>
</tr>
<tr>
<td>20.3 Unit Test</td>
<td>324</td>
</tr>
</tbody>
</table>

### 21 Material Properties Units

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Thermal Conductivity</td>
<td>326</td>
</tr>
<tr>
<td>21.2 Magnetic Resistivity</td>
<td>326</td>
</tr>
<tr>
<td>21.2.1 Constant resistivity</td>
<td>327</td>
</tr>
<tr>
<td>21.2.2 Spitzer HighZ resistivity</td>
<td>327</td>
</tr>
<tr>
<td>21.3 Viscosity</td>
<td>327</td>
</tr>
<tr>
<td>21.4 Opacity</td>
<td>327</td>
</tr>
<tr>
<td>21.4.1 Constant Implementation</td>
<td>328</td>
</tr>
<tr>
<td>21.4.2 Multispecies Implementation</td>
<td>328</td>
</tr>
<tr>
<td>21.4.3 The IONMIX EOS/Opacity Format</td>
<td>330</td>
</tr>
<tr>
<td>21.5 Mass Diffusivity</td>
<td>333</td>
</tr>
</tbody>
</table>

### 22 Radiative Transfer Unit

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1 Multigroup Diffusion</td>
<td>335</td>
</tr>
<tr>
<td>22.1.1 Using Multigroup Radiation Diffusion</td>
<td>336</td>
</tr>
<tr>
<td>22.1.2 Using Mesh Replication with MGD</td>
<td>337</td>
</tr>
<tr>
<td>22.1.3 Specifying Initial Conditions</td>
<td>338</td>
</tr>
<tr>
<td>22.1.4 Altering the Radiation Spectrum</td>
<td>340</td>
</tr>
</tbody>
</table>

### VI Monitor Units

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 Logfile Unit</td>
<td>345</td>
</tr>
<tr>
<td>23.1 Meta Data</td>
<td>346</td>
</tr>
<tr>
<td>23.2 Runtime Parameters, Physical Constants, and Multispecies Data</td>
<td>346</td>
</tr>
<tr>
<td>23.3 Accessor Functions and Timestep Data</td>
<td>348</td>
</tr>
<tr>
<td>23.4 Performance Data</td>
<td>349</td>
</tr>
<tr>
<td>23.5 Example Usage</td>
<td>350</td>
</tr>
</tbody>
</table>
## 24 Timer and Profiler Units

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.1</td>
<td>Timers</td>
<td>351</td>
</tr>
<tr>
<td>24.1.1</td>
<td>MPINative</td>
<td>351</td>
</tr>
<tr>
<td>24.1.2</td>
<td>Tau</td>
<td>352</td>
</tr>
<tr>
<td>24.2</td>
<td>Profiler</td>
<td>353</td>
</tr>
</tbody>
</table>

## VII The Supplied Test Problems

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.1</td>
<td>Hydrodynamics Test Problems</td>
<td>357</td>
</tr>
<tr>
<td>25.1.1</td>
<td>Sod Shock-Tube</td>
<td>358</td>
</tr>
<tr>
<td>25.1.2</td>
<td>Variants of the Sod Problem in Curvilinear Geometries</td>
<td>361</td>
</tr>
<tr>
<td>25.1.3</td>
<td>Interacting Blast-Wave Blast2</td>
<td>362</td>
</tr>
<tr>
<td>25.1.4</td>
<td>Sedov Explosion</td>
<td>366</td>
</tr>
<tr>
<td>25.1.5</td>
<td>Isentropic Vortex</td>
<td>371</td>
</tr>
<tr>
<td>25.1.6</td>
<td>The double Mach reflection problem</td>
<td>372</td>
</tr>
<tr>
<td>25.1.7</td>
<td>Wind Tunnel With a Step</td>
<td>376</td>
</tr>
<tr>
<td>25.1.8</td>
<td>The Shu-Osher problem</td>
<td>379</td>
</tr>
<tr>
<td>25.1.9</td>
<td>Driven Turbulence StirTurb</td>
<td>382</td>
</tr>
<tr>
<td>25.1.10</td>
<td>Relativistic Sod Shock-Tube</td>
<td>382</td>
</tr>
<tr>
<td>25.1.11</td>
<td>Relativistic Two-dimensional Riemann</td>
<td>389</td>
</tr>
<tr>
<td>25.1.12</td>
<td>Flow Interactions with Stationary Rigid Body</td>
<td>390</td>
</tr>
<tr>
<td>25.2</td>
<td>Magnetohydrodynamics Test Problems</td>
<td>392</td>
</tr>
<tr>
<td>25.2.1</td>
<td>Brio-Wu MHD Shock Tube</td>
<td>392</td>
</tr>
<tr>
<td>25.2.2</td>
<td>Orszag-Tang MHD Vortex</td>
<td>394</td>
</tr>
<tr>
<td>25.2.3</td>
<td>Magnetized Accretion Torus</td>
<td>401</td>
</tr>
<tr>
<td>25.2.4</td>
<td>Magnetized Noh Z-pinch</td>
<td>402</td>
</tr>
<tr>
<td>25.2.5</td>
<td>MHD Rotor</td>
<td>402</td>
</tr>
<tr>
<td>25.2.6</td>
<td>MHD Current Sheet</td>
<td>403</td>
</tr>
<tr>
<td>25.2.7</td>
<td>Field Loop</td>
<td>408</td>
</tr>
<tr>
<td>25.2.8</td>
<td>3D MHD Blast</td>
<td>408</td>
</tr>
<tr>
<td>25.3</td>
<td>Gravity Test Problems</td>
<td>410</td>
</tr>
<tr>
<td>25.3.1</td>
<td>Jeans Instability</td>
<td>410</td>
</tr>
<tr>
<td>25.3.2</td>
<td>Homologous Dust Collapse</td>
<td>414</td>
</tr>
<tr>
<td>25.3.3</td>
<td>Huang-Greengard Poisson Test</td>
<td>416</td>
</tr>
<tr>
<td>25.3.4</td>
<td>MacLaurin</td>
<td>417</td>
</tr>
<tr>
<td>25.4</td>
<td>Particles Test Problems</td>
<td>422</td>
</tr>
<tr>
<td>25.4.1</td>
<td>Two-particle Orbit</td>
<td>422</td>
</tr>
<tr>
<td>25.4.2</td>
<td>Zel’dovich Pancake</td>
<td>423</td>
</tr>
<tr>
<td>25.4.3</td>
<td>Modified Huang-Greengard Poisson Test</td>
<td>426</td>
</tr>
<tr>
<td>25.5</td>
<td>Burn Test Problem</td>
<td>426</td>
</tr>
<tr>
<td>25.5.1</td>
<td>Cellular Nuclear Burning</td>
<td>426</td>
</tr>
<tr>
<td>25.6</td>
<td>RadTrans Test Problems</td>
<td>430</td>
</tr>
<tr>
<td>25.6.1</td>
<td>Infinite Medium, Energy Equilibration</td>
<td>430</td>
</tr>
<tr>
<td>25.6.2</td>
<td>Radiation Step Test</td>
<td>430</td>
</tr>
<tr>
<td>25.7</td>
<td>Other Test Problems</td>
<td>432</td>
</tr>
<tr>
<td>25.7.1</td>
<td>The non-equilibrium ionization test problem</td>
<td>432</td>
</tr>
<tr>
<td>25.7.2</td>
<td>The Delta-Function Heat Conduction Problem</td>
<td>437</td>
</tr>
<tr>
<td>25.7.3</td>
<td>The HydroStatic Test Problem</td>
<td>437</td>
</tr>
<tr>
<td>25.7.4</td>
<td>Hybrid-PIC Test Problems</td>
<td>437</td>
</tr>
<tr>
<td>25.7.5</td>
<td>Full-physics Laser Driven Simulation</td>
<td>439</td>
</tr>
<tr>
<td>25.8</td>
<td>3T Shock Simulations</td>
<td>448</td>
</tr>
<tr>
<td>25.8.1</td>
<td>Shafranov Shock</td>
<td>448</td>
</tr>
</tbody>
</table>
25.8.2 Non-Equilibrium Radiative Shock ........................................ 449
25.8.3 Blast Wave with Thermal Conduction .................................... 451

VIII Tools .................................................................................. 455

26 VisIt ..................................................................................... 459

27 Serial FLASH Output Comparison Utility (sfocu) ......................... 461
  27.1 Building sfocu ................................................................. 461
  27.2 Using sfocu ................................................................. 461

28 Drift ...................................................................................... 463
  28.1 Introduction .................................................................... 463
  28.2 Enabling drift ............................................................... 464
  28.3 Typical workflow ........................................................... 464
  28.4 Caveats and Annoyances .................................................. 465

29 FLASH IDL Routines (fidlr3.0) .............................................. 467
  29.1 Installing and Running fidlr3.0 ........................................... 467
    29.1.1 Setting Up fidlr3.0Environment Variables ...................... 467
    29.1.2 Running IDL .............................................................. 468
  29.2 xflash3: A Widget Interface to Plotting FLASH Datasets ......... 468
    29.2.1 File Menu ................................................................. 468
    29.2.2 Defaults Menu .......................................................... 469
    29.2.3 Colormap Menu ........................................................ 470
    29.2.4 X/Y plot count Menu .................................................. 470
    29.2.5 Plotting options available from the GUI ....................... 470
    29.2.6 Plotting buttons ........................................................ 472
  29.3 Comparing two datasets .................................................... 474

30 convertspec3d ....................................................................... 477
  30.1 Installation ................................................................. 477
  30.2 Usage ................................................................. 478

IX Going Further with FLASH .................................................. 479

31 Adding new solvers .............................................................. 481

32 Porting FLASH to other machines ......................................... 483
  32.1 Writing a Makefile.h ....................................................... 483

33 Multithreaded FLASH ......................................................... 487
  33.1 Overview ................................................................... 487
  33.2 Threading strategies ...................................................... 487
  33.3 Running multithreaded FLASH ........................................ 488
    33.3.1 OpenMP variables .................................................... 488
    33.3.2 FLASH variables ...................................................... 488
    33.3.3 FLASH constants ..................................................... 489
  33.4 Verifying correctness ...................................................... 489
  33.5 Performance results ....................................................... 489
    33.5.1 Multipole solver ....................................................... 489
    33.5.2 Helmholtz EOS ......................................................... 490
    33.5.3 Sedov ................................................................. 491
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33.5.4 LaserSlab</td>
<td>492</td>
</tr>
<tr>
<td>33.6 Conclusion</td>
<td>493</td>
</tr>
<tr>
<td>References</td>
<td>495</td>
</tr>
<tr>
<td>Runtime Parameters</td>
<td>501</td>
</tr>
<tr>
<td>API Index</td>
<td>505</td>
</tr>
<tr>
<td>Index</td>
<td>507</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The FLASH code is a modular, parallel multiphysics simulation code capable of handling general compressible flow problems found in many astrophysical environments. It is a set of independent code units put together with a Python language setup tool to form various applications. The code is written in FORTRAN90 and C. It uses the Message-Passing Interface (MPI) library for inter-processor communication and the HDF5 or Parallel-NetCDF library for parallel I/O to achieve portability and scalability on a variety of different parallel computers. FLASH4 has three interchangeable discretization grids: a Uniform Grid, and a block-structured oct-tree based adaptive grid using the PARAMESH library, and a block-structured patch based adaptive grid using Chombo. Both PARAMESH and Chombo place resolution elements only where they are needed most. Chombo is new to code, has had limited testing, and its full capability is not yet deployed. At this time the use of Chombo for production must exercise great care. The code’s architecture is designed to be flexible and easily extensible. Users can configure initial and boundary conditions, change algorithms, and add new physics units with minimal effort.

The Flash Center was founded at the University of Chicago in 1997 under contract to the United States Department of Energy as part of its Accelerated Strategic Computing Initiative (ASCI) (now the Advanced Simulation and Computing (ASC) Program). The scientific goal of the Center then was to address several problems related to thermonuclear flashes on the surface of compact stars (neutron stars and white dwarfs), in particular Type Ia supernovae, and novae. The software goals of the center were to develop new simulation tools capable of handling the extreme resolution and physical requirements imposed by conditions in these explosions and to make them available to the community through the public release of the FLASH code. Since 2009 the several new scienc and computational code development projects have been added to the Center, the notable one among them are: Supernova Models, High-Energy Density Physics (HEDP), Fluid-Structure Interaction, and Implicit Solvers for stiff parabolic and hyperbolic systems with AMR.

The FLASH code has become a key hydrodynamics application used to test and debug new machine architectures because of its modular structure, portability, scalability and dependence on parallel I/O libraries. It has a growing user base and has rapidly become a shared code for the astrophysics community and beyond, with hundreds of active users who customize the code for their own research.

1.1 What’s New in FLASH4

This Guide describes the release version 4.2 of FLASH4. FLASH4 includes all the well tested capabilities of FLASH3. There were a few modules in the official releases of FLASH2 which were added and tested by local users, but did not have standardized setups that could be used to test them after the migration to FLASH3. Those modules are not included in the official release of FLASH3 or FLASH4, however, they are being made available to download ”as is” from the Flash Center’s website. We have ensured that they have been imported into FLASH4 to the extent that they conform to the architecture and compile. We cannot guarantee that they work correctly; they are meant to be useful starting points for users who need their functionality. We also welcome setups contributed by the users that can meaningfully test these units. If such setups become available to us, the units will be released in future.
In terms of the code architecture, FLASH4 closely follows FLASH3. The major changes from FLASH3 are several new capabilities in both physics solvers and infrastructure. Major effort went into the design of the FLASH3 architecture to ensure that the code can be easily modified and extended by internal as well as external developers. Each code unit in FLASH4, like in FLASH3 has a well defined interface and follows the rules for inheritance and encapsulation defined in FLASH3. One of the largest achievements of FLASH3 was the separation of the discretized ‘grid’ architecture from the actual physics. This untangling required changes in the deepest levels of the code but has now demonstrated its worth by allowing us to import a new AMR package \texttt{Chombo} into the code. Additionally, because of the increasing importance of software verification and validation, the Flash Code group released a test-suite application for FLASH3. The application is called FlashTest and can be used to setup, compile, execute and test a series of FLASH code simulations on a regular basis. FlashTest is available without a license and can be downloaded from the Code Support Web Page. There is also a more general open-source version of FlashTest which can be used to test any software in which an application is configured and then executed under a variety of different conditions. The results of the tests can then be visualized in a browser with FlashTestView, a companion to FlashTest that is also open-source.

Many but not all parts of FLASH4 are backwards compatible with FLASH2, and they are all compatible with FLASH3. The Flash code group has written extensive documentation detailing how to make the transition from FLASH2 to FLASH3 as smooth as possible. The user should follow the "Name changes from FLASH2 to FLASH3" link on the Code Support Web Page for help on transitioning to FLASH4 from FLASH2. The transition from FLASH3 to FLASH4 does not require much effort from the users except in any custom implementation they may have.

The new capabilities in FLASH4 that were not included in FLASH3 include

- 3T capabilities in the split and unsplit Hydro solvers. There is support for non-cartesian geometry and the unsplit solver also supports stationary rigid body.
- Upwind biased constrained transport (CT) scheme in the unsplit staggered mesh MHD solver
- Full corner transport upwind (CTU) algorithm in the unsplit hydro/MHD solver
- Cylindrical geometry support in the unsplit staggered mesh MHD solver on UG and AMR. A couple of MHD simulation setups using cylindrical geometry.
- Units for radiation diffusion, conduction, and heat exchange.
- Equation-of state unit includes table based multi-material multi-temperature implementation.
- The Opacities unit with the ability to use hot and cold opacities.
- The laser drive with threading for performance
- Ability to replicate mesh for multigroup diffusion or other similar applications.
- Several important solvers have been threaded at both coarse-grain (one block per thread) and fine-grain (threads within a block) levels.
- Several new HEDP simulation setups.
- A new multipole solver
- Ability to add particles during evolution
- A new EXPERIMENTAL setup tool “setup_alt” intended to be a compatible replacement with a cleaner structure.

The enhancements and bug fixes to the existing capabilities since FLASH4-beta release are:

- The HLLD Riemann solver has been improved to handle MHD degeneracy.
1.1. WHAT’S NEW IN FLASH4

- PARAMESH’s handling for face-centered variables in order to ensure divergence-free magnetic fields evolution on AMR now uses `gr_pmrpDivergenceFree=.true.` and `gr_pmrpForceConsistency=.true.` by default.

- The HEDP capabilities of the code have been exercised and are therefore more robust.

- Laser 3D in 2D ray tracing has been added. The code traces rays in a real 3D cylindrical domain using a computational 2D cylindrical domain and is based on a polygon approximation to the angular part.

- In non-fixedblocksize mode, restart with particles did not work when starting with a different processor count. This bug has now been fixed.

- All I/O implementations now support reading/writing 0 blocks and 0 particles.

- There is support for particles and face variables in PnetCDF

- Initialization of of the computation domain has been optimized by eliminating redundant calls to orrery at simulation startup. It is possible to run the orrery in a reduced communicator in order to speed up FLASH initialization.

- The custom region code and corresponding Grid API routines have been removed.

- PARAMESH4DEV is now the default PARAMESH implementation.

The new capabilities since FLASH4.0.1 include:

- New Core-Collapse Super Nova (CCSN) physics, with complete nuclear EOS routines, local neutrino heating/cooling and multispecies neutrino leakage.

- New, highly optimized for performance Hydro and MHD unsplit modules. These implementations are now the default option. We have retained the old implementations for compatibility reasons.

- New support for 3T magnetohydrodynamics, designed for HEDP problems.

- A new magnetic resistivity implementation, Spitzer HighZ, for HEDP problems. We have also extended the support for resistivity in cylindrical geometry in the unsplit solver.

- New threading capabilities for unsplit MHD, compatible with all threading strategies followed by the code.


- Reorganization of the Energy Deposition module. A new feature has been included that allows EnergyDeposition to be called once every n time steps.

The following features are provided on an EXPERIMENTAL basis. They may only work in limited circumstances and/or have not yet been tested to our satisfaction.

- New Laser - Async communication (experimental).

- Electron-Entropy Advection in Hydro for non-ideal Eos.

- New cubic and quartic equation solvers have been added and are ready to be used. They return only real cubic and quartic roots. The routines can be found in the flashUtilites/general section.
1.2 External Contributions

Beginning with this release, we list the contributions from outside the Flash Center that are included in the distribution. For completeness, we also list such contributions in FLASH3 which have long been included in the release.

- Direct solvers for Uniform Grid contributed by Marcos Vanella. The solvers have been in the release since FLASH3. Reference: [http://dx.doi.org/10.1002/cpe.2821](http://dx.doi.org/10.1002/cpe.2821)
- Hybrid-PIC code, contributed by Mats Holmström. The contribution has been in the distribution since FLASH4-alpha. Reference: [http://adsabs.harvard.edu/abs/2011arXiv1104.1440H](http://adsabs.harvard.edu/abs/2011arXiv1104.1440H)
- Barnes Hut tree gravity solver contributed by Richard Wunsch. This contribution has been further extended in the current release and has been developed in collaboration with Frantisek Dimbier (responsible for periodic and mixed boundary conditions) and Stefanie Walch.
- Sink Particles contributed by Christoph Federrath et al. This contribution has received significant updates in the current release. Please refer to [http://iopscience.iop.org/0004-637X/713/1/269/](http://iopscience.iop.org/0004-637X/713/1/269/) for details. In this release there is also a new ‘FromFile’ implementation of the Stir unit, now sitting beside the older ‘Generate’ implementation.
- New Flame and Turb units contributed by Dean Townsley. More information can be found in Chapter 16.

1.3 Known Issues in This Release

- The outflow boundary condition for face-centered variables in the use of solenoidal magnetic field evolution on AMR fails to ensure the solenoidal constraint at the physical outflow boundaries. However, numerical solutions with respect to this error are still physically correct away from the outflow boundaries. This issue will be resolved in future releases.
- The upwind-biased electric field implementation (i.e., `E_upwind=.true.`) for the unsplit staggered mesh solver in some cases fails to satisfy divergence-free magnetic field evolutions at restart. Users can still use (i.e., `E_upwind=.false.`) in most applications.
- The new multipole solver is missing the ability to treat a non-zero minimal radius for spherical geometries, and the ability to specify a point mass contribution to the potential.
- The “Split” implementation of the diffusion solver is essentially meant for testing purposes. Moreover, it has not been exercised with PARAMESH.
- Some configurations of hydrodynamic test problems with Chombo grid show worse than expected mass and total energy conservation. Please see the Chombo section in the Hydro chapter of this Guide for details.
- We have experienced the following abort when running IsentropicVortex problem with Chombo Grid: "MayDay: TreeIntVectSet.cpp:1995: Assertion ‘bxNumPts != 0’ failed. !!!!" We have been in contact with the Chombo team to resolve this issue.
- The unsplit MHD solver doesn’t support the mode "useGravPotUpdate=.true." for 1D when self-gravity is utilized. The solver will still work if it is set to be .false. In this case the usual reconstruction schemes will be used in computing the gravitational accelerations at the predictor step (i.e., at the n+1/2 step) rather than calling the Poisson solver to compute them.
• The -index-reorder setup flag does not work in all the configurations. If you wish to use it please contact the FLASH team.

• Time limiting due to burning, even though has an implementation, is turned off in most simulations by keeping the value of parameter enucDtFactor very high. The implementation is therefore not well tested and should be used with care.

• Mesh replication is only supported for parallel HDF5 and the experimental derived data type Parallel-NetCDF (+pnetTypeIO) I/O implementations. Flash will fail at runtime if multiple meshes are in use without using one of these I/O implementations.

• The unsplit staggered mesh MHD solver shows slight differences (of relative magnitudes of order $10^{-12}$) in restart comparisons (e.g., sfocu comparison), when there are non-zero values of face-centered magnetic fields. With PARAMESH4DEV, the new default Grid implementation, we have observed this problem only with the 'force_consistency' flag (see Runtime Parameter gr_pmrpForceConsistency) turned on.

• In some cases with the default refinement criteria implementation, the refinement pattern at a given point in time of a PARAMESH AMR simulation may be slightly different depending on how often plotfiles and checkpoints are written; with resulting small changes in simulation results. The effect is expected to also be present in previous FLASH versions. This is a side effect of Grid restrictAllLevels calls that happen in IO to prepare all grid blocks for being dumped to file. We have determined that this can only impact how quickly coarser blocks next to a refinement boundary are allowed to further derefine when their better resolved neighbors also derefine, in cases where second-derivative criteria applied to the block itself would allow such derefinement. Users who are concerned with this effect may want to replace the call to amr_restrict in Grid_updateRefinement with a call to Grid_restrictAllLevels, at the cost of a slight increase in runtime.

• The PG compiler fails to compile source files which contain OpenMP parallel regions that reference threadprivate data. This happens in the threaded versions of the Multipole solver and the within block threaded version of split hydro. A workaround is to remove “default(none)” from the OpenMP parallel region.

• Absoft compiler (gcc 4.4.4/Absoft Pro fortran 11.1.x86_64 with mpich2 1.4.1p1) generates incorrectly behaving code with some files when used with any optimization. More specifically, we have seen this behavior with gr_markRefineDerefine.F90, but other files may be vulnerable too. An older version (Absoft Fortran 95 9.0 EP/gcc 4.5.1 with mpich-1.2.7p1) works.

• The index reorder does not work in all the configurations. If you wish to use it please contact the FLASH team.

• The -noclobber setup option will not force a rebuild of all necessary source files in a FLASH application with derived data type I/O (+hdf5TypeIO or +pnetTypeIO). Do not use -noclobber with derived data type I/O.

1.4 About the User’s Guide

This User’s Guide is designed to enable individuals unfamiliar with the FLASH code to quickly get acquainted with its structure and to move beyond the simple test problems distributed with FLASH, customizing it to suit their own needs. Code users and developers are encouraged to visit the FLASH code documentation page for other references to complement the User’s Guide.

Part I provides a rapid introduction to working with FLASH. Chapter 2 (Quick Start) discusses how to get started quickly with FLASH, describing how to setup, build, and run the code with one of the included test problems and then to examine the resulting output. Users unfamiliar with the capabilities of FLASH, who wish to quickly ‘get their feet wet’ with the code, should begin with this section. Users who want to get started immediately using FLASH to create new problems of their own will want to refer to Chapter 3 (Setting Up New Problems) and Chapter 5 (The FLASH Configuration Script).
CHAPTER 1. INTRODUCTION

Part II begins with an overview of both the FLASH code architecture and a brief overview of the
units included with FLASH. It then describes in detail each of the units included with the code, along
with their subunits, runtime parameters, and the equations and algorithms of the implemented solvers.
**Important note:** We assume that the reader has some familiarity both with the basic physics
involved and with numerical methods for solving partial differential equations. This familiarity
is absolutely essential in using FLASH (or any other simulation code) to arrive at meaningful solutions to
physical problems. The novice reader is directed to an introductory text, examples of which include

Fletcher, C. A. J. *Computational Techniques for Fluid Dynamics* (Springer-Verlag, 1991)


Roache, P. *Fundamentals of Computational Fluid Dynamics* (Hermosa, 1998)


The advanced reader who wishes to know more specific information about a given unit’s algorithm is directed
to the literature referenced in the algorithm section of the chapter in question.

Part VII describes the different test problems distributed with FLASH. Part VIII describes in more detail
the analysis tools distributed with FLASH, including **fidlr** and **sfocu**.
Part I

Getting Started
Chapter 2

Quick Start

This chapter describes how to get up-and-running quickly with FLASH with an example simulation, the Sedov explosion. We explain how to configure a problem, build it, run it, and examine the output using IDL.

2.1 System requirements

You should verify that you have the following:

- A copy of the FLASH source code distribution (as a Unix tar file). To request a copy of the distribution, click on the “Code Request” link on the FLASH Center web site. You will be asked to fill out a short form before receiving download instructions. Please remember the username and password you use to download the code; you will need these to get bug fixes and updates to FLASH.

- A F90 (Fortran 90) compiler and a C compiler. Most of FLASH is written in F90. Information available at the Fortran Company web site can help you select an F90 compiler for your system. FLASH has been tested with many Fortran compilers. For details of compilers and libraries, see the RELEASE-NOTES available in the FLASH home directory.

- An installed copy of the Message-Passing Interface (MPI) library. A freely available implementation of MPI called MPICH is available from Argonne National Laboratory.

- To use the Hierarchical Data Format (HDF) for output files, you will need an installed copy of the freely available HDF library. The serial version of HDF5 is the current default FLASH format. HDF5 is available from the HDF Group (http://www.hdfgroup.org/) of the National Center for Supercomputing Applications (NCSA) at http://www.ncsa.illinois.edu. The contents of HDF5 output files produced by the FLASH units are described in detail in Section 9.1.

- To use the Parallel NetCDF format for output files, you will need an installed copy of the freely available PnetCDF library. PnetCDF is available from Argonne National Lab at http://www.mcs.anl.gov/parallel-netcdf/. For details of this format, see Section 9.1.

- To use Chombo as the (Adaptive Mesh Refinement) AMR option, you will need an installed copy of the library, available freely from Lawrence Berkeley National Lab at https://seesar.lbl.gov/anag/chombo. The use of Chombo is described in Section 8.7.

- To use the Diffuse unit with HYPRE solvers, you will need to have an installed copy of HYPRE, available for free from Lawrence Livermore National Lab at https://computation.llnl.gov/casc/hypre/software.html. HYPRE is required for using several critical HEDP capabilities including multigroup radiation diffusion and thermal conduction. Please make sure you have HYPRE installed if you want these capabilities.
To use the output analysis tools described in this section, you will need a copy of the IDL language from ITT Visual Information Solutions. IDL is a commercial product. It is not required for the analysis of FLASH output, but the fidlr3.0 tools described in this section require it. (FLASH output formats are described in Section 9.9. If IDL is not available, another visual analysis option is ViSit, described in Chapter 26.) The newest IDL routines, those contained in fidlr3.0, were written and tested with IDL 6.1 and above. You are encouraged to upgrade if you are using an earlier version. Also, to use the HDF5 version 1.6.2, analysis tools included in IDL require IDL 6.1 and above. New versions of IDL come out frequently, and sometimes break backwards compatibility, but every effort will be made to support them.

- The GNU make utility, gmake. This utility is freely available and has been ported to a wide variety of different systems. For more information, see the entry for make in the development software listing at http://www.gnu.org/. On some systems make is an alias for gmake. GNU make is required because FLASH uses macro concatenation when constructing Makefiles.

- A copy of the Python language, version 2.2 or later is required to run the setup script. Python can be downloaded from http://www.python.org.

### 2.2 Unpacking and configuring FLASH for quick start

To begin, unpack the FLASH source code distribution.

```
tar -xvf FLASHX.Y.tar
```

where X.Y is the FLASH version number (for example, use FLASH4-alpha.tar for FLASH version 4-alpha, or FLASH3.1.tar for FLASH version 3.1). This will create a directory called FLASHX/. Type ‘cd FLASHX/’ to enter this directory. Next, configure the FLASH source tree for the Sedov explosion problem using the setup script. Type

```
./setup Sedov -auto
```

This configures FLASH for the 2d Sedov problem using the default hydrodynamic solver, equation of state, Grid unit, and I/O format defined for this problem, linking all necessary files into a new directory, called ‘object/’. For the purpose of this example, we will use the default I/O format, serial HDF5. In order to compile a problem on a given machine FLASH allows the user to create a file called Makefile.h which sets the paths to compilers and libraries specific to a given platform. This file is located in the directory sites/mymachine.myninstitution.mydomain/. The setup script will attempt to see if your machine/platform has a Makefile.h already created, and if so, this will be linked into the object/ directory. If one is not created the setup script will use a prototype Makefile.h with guesses as to the locations of libraries on your machine. The current distribution includes prototypes for AIX, IRIX64, Linux, Darwin, and TFLOPS operating systems. In any case, it is advisable to create a Makefile.h specific to your machine. See Section 5.6 for details.

Type the command cd object to enter the object directory which was created when you setup the Sedov problem, and then execute make. This will compile the FLASH code.

```
cd object
make
```

If you have problems and need to recompile, ‘make clean’ will remove all object files from the object/ directory, leaving the source configuration intact; ‘make realclean’ will remove all files and links from object/. After ‘make realclean’, a new invocation of setup is required before the code can be built. The building can take a long time on some machines; doing a parallel build (make -j for example) can significantly increase compilation speed, even on single processor systems.

Assuming compilation and linking were successful, you should now find an executable named flashX in the object/ directory, where X is the major version number (e.g., 4 for X.Y = 4.0). You may wish to check that this is the case.

If compilation and linking were not successful, here are a few common suggestions to diagnose the problem:
2.2. UNPACKING AND CONFIGURING FLASH FOR QUICK START

- Make sure the correct compilers are in your path, and that they produce a valid executable.
- The default Sedov problem uses HDF5 in serial. Make sure you have HDF5 installed. If you do not have HDF5, you can still set up and compile FLASH, but you will not be able to generate either a checkpoint or a plot file. You can set up FLASH without I/O by typing

  ```bash
  ./setup Sedov -auto +noio
  ```

- Make sure the paths to the MPI and HDF libraries are correctly set in the Makefile.h in the object/ directory.
- Make sure your version of MPI creates a valid executable that can run in parallel.

These are just a few suggestions; you might also check for further information in this guide or at the FLASH web page.

FLASH by default expects to find a text file named `flash.par` in the directory from which it is run. This file sets the values of various runtime parameters that determine the behavior of FLASH. If it is not present, FLASH will abort; `flash.par` must be created in order for the program to run (note: all of the distributed setups already come with a `flash.par` which is copied into the object/ directory at setup time). There is a command-line option to use a different name for this file, described in the next section. Here we will create a simple `flash.par` that sets a few parameters and allows the rest to take on default values. With your text editor, edit the `flash.par` in the `object` directory so it looks like Figure 2.1.

```
# runtime parameters
lrefine_max = 5
basenm = "sedov_"
restart = .false.
checkpointFileIntervalTime = 0.01
nend = 10000
tmax = 0.05
gamma = 1.4
xl_boundary_type = "outflow"
xr_boundary_type = "outflow"
yl_boundary_type = "outflow"
yr_boundary_type = "outflow"
plot_var_1 = "dens"
plot_var_2 = "temp"
plot_var_3 = "pres"
```

Figure 2.1: FLASH parameter file contents for the quick start example.

This example instructs FLASH to use up to five levels of adaptive mesh refinement (AMR) (through the `lrefine_max` parameter) and to name the output files appropriately (`basenm`). We will not be starting from a checkpoint file ("`restart = .false.`" — this is the default, but here it is explicitly set for clarity). Output files are to be written every 0.01 time units (`checkpointFileIntervalTime`) and will be created until `t = 0.05` or `10000` timesteps have been taken (`tmax` and `nend` respectively), whichever comes first. The ratio
of specific heats for the gas ($\gamma$) is taken to be 1.4, and all four boundaries of the two-dimensional grid have outflow (zero-gradient or Neumann) boundary conditions (set via the \texttt{[xy][lr]_boundary_type} parameters).

Note the format of the file – each line is of the form \texttt{variable=value}, a comment (denoted by a hash mark, #), or a blank. String values are enclosed in double quotes ("). Boolean values are indicated in the FORTRAN style, \texttt{.true.} or \texttt{.false.}. Be sure to insert a carriage return after the last line of text. A full list of the parameters available for your current setup is contained in the file \texttt{setup_params} located in the \texttt{object/} directory, which also includes brief comments for each parameter. If you wish to skip the creation of a \texttt{flash.par}, a complete example is provided in the \texttt{source/Simulation/SimulationMain/Sedov/} directory.

## 2.3 Running FLASH

We are now ready to run FLASH. To run FLASH on $N$ processors, type

```
mpirun -np \textit{N} flash\textit{X}
```

remembering to replace \textit{N} and \textit{X} with the appropriate values. Some systems may require you to start MPI programs with a different command; use whichever command is appropriate for your system. The FLASH4 executable accepts an optional command-line argument for the runtime parameters file. If \texttt{"-par_file file-name"} is present, FLASH reads the file specified on command line for runtime parameters, otherwise it reads \texttt{flash.par}.

You should see a number of lines of output indicating that FLASH is initializing the Sedov problem, listing the initial parameters, and giving the timestep chosen at each step. After the run is finished, you should find several files in the current directory:

- \texttt{sedov.log} echoes the runtime parameter settings and indicates the run time, the build time, and the build machine. During the run, a line is written for each timestep, along with any warning messages. If the run terminates normally, a performance summary is written to this file.

- \texttt{sedov.dat} contains a number of integral quantities as functions of time: total mass, total energy, total momentum, etc. This file can be used directly by plotting programs such as \texttt{gnuplot}; note that the first line begins with a hash (#) and is thus ignored by \texttt{gnuplot}.

- \texttt{sedov.hdf5_chk_000*} are the different checkpoint files. These are complete dumps of the entire simulation state at intervals of \texttt{checkpointFileIntervalTime} and are suitable for use in restarting the simulation.

- \texttt{sedov.hdf5_plt_cnt_000*} are plot files. In this example, these files contain density, temperature, and pressure in single precision. If needed, more variables can be dumped in the plotfiles by specifying them in \texttt{flash.par}. They are usually written more frequently than checkpoint files, since they are the primary output of FLASH for analyzing the results of the simulation. They are also used for making simulation movies. Checkpoint files can also be used for analysis and sometimes it is necessary to use them since they have comprehensive information about the state of the simulation at a given time. However, in general, plotfiles are preferred since they have more frequent snapshots of the time evolution. Please see Chapter 9 for more information about IO outputs.

We will use the \texttt{xflash3} routine under IDL to examine the output. Before doing so, we need to set the values of three environment variables, \texttt{IDL_DIR}, \texttt{IDL_PATH} and \texttt{XFLASH3_DIR}. You should usually have \texttt{IDL_DIR} already set during the IDL installation. Under \texttt{csh} the two additional variables can be set using the commands

```
setenv XFLASH3_DIR "<flash home dir>/tools/fidlr3.0"
setenv IDL_PATH "$\{XFLASH3_DIR\}:$IDL_PATH"
```

where \texttt{<flash home dir>} is the location of the \texttt{FLASH.X.Y} directory. If you get a message indicating that \texttt{IDL_PATH} is not defined, enter
setenv IDL_PATH "${XFLASH3_DIR}}:${IDL_DIR}:${IDL_DIR}/lib"

where ${IDL_DIR} points to the directory where IDL is installed. Fidlr assumes that you have a version of IDL with native HDF5 support.

FLASH Transition

Please note! The environment variable used with FLASH3 onwards is XFLASH3_DIR. The main routine name for interactive plotting is xflash3.

Now run IDL (idl or idl start_linux) and enter xflash3 at the IDL> prompt. You should see the main widget as shown in Figure 2.2.

Select any of the checkpoint or plot files through the File/Open Prototype... dialog box. This will define a prototype file for the dataset, which is used by fidlr to set its own data structures. With the prototype defined, enter the suffixes '0000', '0005' and '1' in the three suffix boxes. This tells xflash3 which files to plot. xflash3 can generate output for a number of consecutive files, but if you fill in only the beginning suffix, only one file is read. Click the auto box next to the data range to automatically scale the plot to the data. Select the desired plotting variable and colormap. Under ‘Options,’ select whether to plot the logarithm of the desired quantity and select whether to plot the outlines of the computational blocks. For very highly refined grids, the block outlines can obscure the data, but they are useful for verifying that FLASH is putting resolution elements where they are needed.

When the control panel settings are to your satisfaction, click the ‘Plot’ button to generate the plot. For Postscript or PNG output, a file is created in the current directory. The result should look something like Figure 2.3, although this figure was generated from a run with eight levels of refinement rather than the five used in the quick start example run. With fewer levels of refinement, the Cartesian grid causes the explosion to appear somewhat diamond-shaped. Please see Chapter 29 for more information about visualizing FLASH output with IDL routines.

FLASH is intended to be customized by the user to work with interesting initial and boundary conditions. In the following sections, we will cover in more detail the algorithms and structure of FLASH and the sample problems and tools distributed with it.
Figure 2.2: The main xflash3 widget.
2.3. RUNNING FLASH

Figure 2.3: Example of xflash output for the Sedov problem with eight levels of refinement.
Chapter 3

Setting Up New Problems

A new FLASH problem is created by making a directory for it under FLASH4/source/Simulation/SimulationMain. This location is where the FLASH setup script looks to find the problem-specific files. The FLASH distribution includes a number of pre-written simulations; however, most FLASH users will want to simulate their own problems, so it is important to understand the techniques for adding a customized problem simulation.

Every simulation directory contains routines to initialize the FLASH grid. The directory also includes a Config file which contains information about infrastructure and physics units, and the runtime parameters required by the simulation (see Chapter 5). The files that are usually included in the Simulation directory for a problem are:

- **Config**: Lists the units and variables required for the problem, defines runtime parameters and initializes them with default values.
- **Makefile**: The make include file for the Simulation.
- **Simulation_data.F90**: Fortran module which stores data and parameters specific to the Simulation.
- **Simulation_init.F90**: Fortran routine which reads the runtime parameters, and performs other necessary initializations.
- **Simulation_initBlock.F90**: Fortran routine for setting initial conditions in a single block.
- **Simulation_initSpecies.F90**: Optional Fortran routine for initializing species properties if multiple species are being used.
- **flash.par**: A text file that specifies values for the runtime parameters. The values in flash.par override the defaults from Config files.

In addition to these basic files, a particular simulation may include some files of its own. These files could provide either new functionality not available in FLASH, or they may include customized versions of any of the FLASH routines. For example, a problem might require a custom refinement criterion instead of the one provided with FLASH. If a customized implementation of Grid_markRefineDerefine is placed in the Simulation directory, it will replace FLASH’s own implementation when the problem is setup. In general, users are encouraged to put any modifications of core FLASH files in the SimulationMain directory in which they are working rather than by altering the default FLASH routines. This encapsulation of personal changes will make it easier to integrate Flash Center patches, and to upgrade to more recent versions of the code. The user might also wish to include data files in the SimulationMain necessary for initial conditions. Please see the LINKIF and DATAFILES keywords in Section 5.5.1 for more information on linking in datafiles or conditionally linking customized implementations of FLASH routines.

The next few paragraphs are devoted to the detailed examination of the basic files for an example setup. The example we describe here is a hydrodynamical simulation of the Sod shock tube problem, which has a one-dimensional flow discontinuity. We construct the initial conditions for this problem by establishing a planar interface at some angle to the x and y axes, across which the density and pressure values are discontinuous. The fluid is initially at rest on either side of the interface. To create a new simulation, we first create a new directory Sod in Simulation/SimulationMain and then add the Config, Makefile, flash.par, Simulation_initBlock.F90, Simulation_init.F90 and Simulation_data.F90 files. Since this is a single fluid simulation, there is no need for a Simulation_initSpecies file. The easiest way to construct
these files is to use files from another setup as templates.

### 3.1 Creating a Config file

The Config file for this example serves two principal purposes; (1) to specify the required units and (2) to register runtime parameters.

```plaintext
# configuration file for our example problem
REQUIRES Driver
REQUIRES physics/Eos/EosMain/Gamma
REQUIRES physics/Hydro
```

The lines above define the FLASH units required by the Sod problem. Note that we do not ask for particular implementations of the Hydro unit, since for this problem many implementations will satisfy the requirements. However, we do ask for the gamma-law equation of state (physics/Eos/EosMain/Gamma) specifically, since that implementation is the only valid option for this problem. In FLASH4-alpha, the PARAMESH 4 Grid implementation is passed to Driver by default. As such, there is no need to specify a Grid unit explicitly, unless a simulation requires an alternative Grid implementation. Also important to note is that we have not explicitly required IO, which is included by default. In constructing the list of requirements for a problem, it is important to keep them as general as the problem allows. We recommend asking for specific implementations of units as command line options or in the Units file when the problem is being setup, to avoid the necessity of modifying the Config files. For example, if there was more than one implementation of Hydro that could handle the shocks, any of them could be picked at setup time without having to modify the Config file. However, to change the Eos to an implementation other than Gamma, the Config file would have to be modified. For command-line options of the setup script and the description of the Units file see Chapter 5.

After specifying the units, the Config file lists the runtime parameters specific to this problem. The names of runtime parameters are case-insensitive. Note that no unit is constrained to use only the parameters defined in its own Config file. It can legitimately access any runtime parameter registered by any unit included in the simulation.

```plaintext
PARAMETER sim_rhoLeft REAL 1. [0 to ]
PARAMETER sim_rhoRight REAL 0.125 [0 to ]
PARAMETER sim_pLeft REAL 1. [0 to ]
PARAMETER sim_pRight REAL 0.1 [0 to ]
PARAMETER sim_uLeft REAL 0.
PARAMETER sim_uRight REAL 0.
PARAMETER sim_xangle REAL 0. [0 to 360]
PARAMETER sim_yangle REAL 90. [0 to 360]
PARAMETER sim_posn REAL 0.5
```

Here we define (sim_rhoLeft), (sim_pLeft) and (sim_uLeft) as density, pressure and velocity to the left of the discontinuity, and (sim_rhoRight), (sim_pRight) and (sim_uRight) as density, pressure and velocity to the right of the discontinuity. The parameters (sim_xangle) and (sim_yangle) give the angles with respect to the x and y axes, and (sim_posn) specifies the intersection between the shock plane and x axis. The quantities in square brackets define the permissible range of values for the parameters. The default value of any parameter (like sim_xangle) can be overridden at runtime by including a line (i.e. sim_xangle = 45.0) defining a different value for it in the flash.par file.

### 3.2 Creating a Makefile

The file Makefile included in the Simulation directory does not have the standard Makefile format for make/gmake. Instead, the setup script generates a complete compilation Makefile from the machine/system specific one (see Section 5.6) and the unit Makefiles (see Section 5.7.3).
In general, standard module and routine dependencies are figured out by the setup script or are inherited from the directory structure. The Makefile for this example is very simple, it only adds the object file for Simulation_data to the Simulation unit Makefile. The additional object files such as Simulation_init.o are already added in the directory above SimulationMain.

### 3.3 Creating a Simulation_data.F90

The Fortran module Simulation_data is used to store data specific to the Simulation unit. In FLASH4-alpha there is no central ‘database’, instead, each unit stores its own data in its Unit_data Fortran module. Data needed from other units is accessed through that unit’s interface. The basic structure of the Simulation_data module is shown below:

```fortran
module Simulation_data
  implicit none
  !! Runtime Parameters
  real, save :: sim_rhoLeft, sim_rhoRight, sim_pLeft, sim_pRight
  real, save :: sim_uLeft, sim_uRight, sim_xAngle, sim_yAngle, sim_posn
  real, save :: sim_gamma, sim_smallP, sim_smallX

  !! Other unit variables
  real, save :: sim_xCos, sim_yCos, sim_zCos
  end module Simulation_data
```

Note that all the variables in this data module have the save attribute. Without this attribute the storage for the variable is not guaranteed outside of the scope of Simulation_data module with many compilers. Also notice that there are many more variables in the data module than in the Config file. Some of them, such as ‘sim_smallX’ etc, are runtime parameters from other units, while others such as ‘sim_xCos’ are simulation specific variables that are available to all routines in the Simulation unit. The FLASH4-alpha naming convention is that variables that begin with sim are used or “belong” to the Simulation unit.

### 3.4 Creating a Simulation_init.F90

The routine Simulation_init is called by the routine Driver_initFlash at the beginning of the simulation. Driver_initFlash calls Unit_init.F90 routines of every unit to initialize them. In this particular case, the Simulation_init routine will get the necessary runtime parameters and store them in the Simulation_data Fortran module, and also initialize other variables in the module. More generally, all one-time initialization required by the simulation are implemented in the Simulation_init routine.

**FLASH Transition**

In FLASH2, the contents of the if (.firstcall.) clause are now in the Simulation_init routine in FLASH4.

The basic structure of the routine Simulation_init should consist of
CHAPTER 3. SETTING UP NEW PROBLEMS

1. Fortran module use statement for the Simulation_data
2. Fortran module use statement for the Unit_interfaces to access the interface of the RuntimeParameters unit, and any other units being used.
3. Variable typing implicit none statement
4. Necessary #include header files
5. Declaration of arguments and local variables.
6. Calls to the RuntimeParameters unit interface to obtain the values of runtime parameters.
7. Calls to the PhysicalConstants unit interface to initialize any necessary physical constants.
8. Calls to the Multispecies unit interface to initialize the species’ properties, if there multiple species are in use
9. Initialize other unit scope variables, packages and functions
10. Any other calculations that are needed only once at the beginning of the run.

In this example after the implicit none statement we include two files, "constants.h", and "Flash.h". The "constants.h" file holds global constants defined in the FLASH code such as MDIM, MASTER_PE, and MAX_STRING_LENGTH. It also stores constants that make reading the code easier, such as IAXIS, JAXIS, and KAXIS, which are defined as 1,2, and 3, respectively. More information is available in comments in the distributed constants.h. A complete list of defined constants is available on the Code Support Web Page.

The "Flash.h" file contains all of the definitions specific to a given problem. This file is generated by the setup script and defines the indices for the variables in various data structures. For example, the index for density in the cell centered grid data structure is defined as DENS_VAR. The "Flash.h" file also defines the number of species, number of dimensions, maximum number of blocks, and many more values specific to a given run. Please see Chapter 6 for complete description of the Flash.h file.

FLASH Transition

The defined constants in "Flash.h" file allows the user direct access to the variable index in ‘unk.’ This direct access is unlike FLASH2, where the user would first have to get the integer index of the variable by calling a data base function and then use the integer variable idens as the variable index. Previously:

\texttt{idens=dBaseKeyNumber(’dens’)}
\texttt{ucons(1,i) = solnData(idens,i,j,k)}

Now, the syntax is simpler:

\texttt{ucons(1,i) = solnData(DENS_VAR,i,j,k)}

This new syntax also allows discovery of specification errors at compile time.

subroutine Simulation_init()

use Simulation_data
use RuntimeParameters_interface, ONLY : RuntimeParameters_get
implicit none
! get the runtime parameters relevant for this problem

```fortran
#include "Flash.h"
#include "constants.h"

call RuntimeParameters_get('smallp', sim_smallP)
call RuntimeParameters_get('smallx', sim_smallX)
call RuntimeParameters_get('gamma', sim_gamma)
call RuntimeParameters_get('sim_rhoLeft', sim_rhoLeft)
call RuntimeParameters_get('sim_rhoRight', sim_rhoRight)
call RuntimeParameters_get('sim_pLeft', sim_pLeft)
call RuntimeParameters_get('sim_pRight', sim_pRight)
call RuntimeParameters_get('sim_uLeft', sim_uLeft)
call RuntimeParameters_get('sim_uRight', sim_uRight)
call RuntimeParameters_get('sim_xAngle', sim_xAngle)
call RuntimeParameters_get('sim_yAngle', sim_yAngle)
call RuntimeParameters_get('sim_posn', sim_posn)

! Do other initializations
! convert the shock angle parameters

sim_xAngle = sim_xAngle * 0.0174532925 ! Convert to radians.
sim_yAngle = sim_yAngle * 0.0174532925

sim_xCos = cos(sim_xAngle)

if (NDIM == 1) then
    sim_xCos = 1.
sim_yCos = 0.
sim_zCos = 0.
elseif (NDIM == 2) then
    sim_yCos = sqrt(1. - sim_xCos*sim_xCos)
sim_zCos = 0.
elseif (NDIM == 3) then
    sim_yCos = cos(sim_yAngle)
    sim_zCos = sqrt( max(0., 1. - sim_xCos*sim_xCos - sim_yCos*sim_yCos) )
endif
```

end subroutine Simulation_init

### 3.5 Creating a Simulation_initBlock.F90

The routine `Simulation_initBlock` is called by the `Grid` unit to apply initial conditions to the physical domain. If the AMR grid `PARAMESH` is being used, the formation of the physical domain starts at the lowest level of refinement. Initial conditions are applied to each block at this level by calling `Simulation_initBlock`. The `Grid` unit then checks the refinement criteria in the blocks it has created and refines the blocks if the criteria are met. It then calls `Simulation_initBlock` to initialize the newly created blocks. This process repeats until the grid reaches the required refinement level in the areas marked for refinement. The Uniform Grid has only one level, with same resolution everywhere. Therefore, only one block per processor is created and `Simulation_initBlock` is called to initialize this single block. It is important to note that a problem's
Simulation_initBlock routine is the same regardless of whether PARAMESH or Uniform Grid is being used. The Grid unit handles these differences, not the Simulation unit.

The basic structure of the routine Simulation_initBlock should be as follows:

1. A use statement for the Simulation_data
2. One of more use statement to access other unit interfaces being used, for example
   use Grid_interface, ONLY: Grid_putPointData
3. Variable typing implicit none statement
4. Necessary #include header files
5. Declaration of arguments and local variables.
6. Generation of initial conditions either from a file, or directly calculated in the routine
7. Calls to the various Grid_putData routines to store the values of solution variables.

We continue to look at the Sod setup and describe its Simulation_initBlock in detail. The first part of the routine contains all the declarations as shown below. The first statement in routine is the use statement, which provides access to the runtime parameters and other unit scope variables initialized in the Simulation_init routine. The include files bring in the needed constants, and then the arguments are defined. The declaration of the local variables is next, with allocatable arrays for each block.

```fortran
subroutine Simulation_initBlock(blockID)
  ! get the needed unit scope data
  use Simulation_data, ONLY: sim_posn, sim_xCos, sim_yCos, sim_zCos, &
    sim_rhoLeft, sim_pLeft, sim_uLeft, &
    sim_rhoRight, sim_pRight, sim_uRight, &
    sim_smallX, sim_gamma, sim_smallP
  use Grid_interfaces, ONLY : Grid_getBlkIndexLimits, Grid_getCellCoords, &
    Grid_putPointData
  implicit none

  ! get all the constants
  #include "constants.h"
  #include "Flash.h"

  ! define arguments and indicate whether they are input or output
  integer, intent(in) :: blockID

  ! declare all local variables.
  integer :: i, j, k, n
  integer :: iMax, jMax, kMax
  real :: xx, yy, zz, xxL, xxR
  real :: lPosn0, lPosn

  ! arrays to hold coordinate information for the block
  real, allocatable, dimension(:) :: xCenter, xLeft, xRight, yCoord, zCoord

  ! array to get integer indices defining the beginning and the end
  ! of a block.
  integer, dimension(2,MDIM) :: blkLimits, blkLimitsGC
```
! the number of grid points along each dimension
integer :: sizeX, sizeY, sizeZ

integer, dimension(MDIM) :: axis
integer :: dataSize
logical :: gcell = .true.

! these variables store the calculated initial values of physical
! variables a grid point at a time.
real :: rhoZone, velxZone, velyZone, velzZone, presZone, &
        enerZone, ekinZone

Note that FLASH promotes all floating point variables to double precision at compile time for maximum
portability. We therefore declare all floating point variables with \texttt{real} in the source code. In the next part
of the code we allocate the arrays that will hold the coordinates.

\begin{quote}
\textbf{FLASH Transition}
\end{quote}

FLASH4-alpha supports blocks that are not sized at compile time to generalize the Uniform
Grid, and to be able to support different AMR packages in future. For this reason, the
arrays are not sized with the static NXB etc. as was the case in FLASH2. Instead they are
allocated on a block by block basis in \texttt{Simulation\_initBlock}. Performance is compromised
by the use of allocatable arrays, however, since this part of the code is executed only at
the beginning of the simulation, it has negligible impact on the overall execution time in
production runs.

! get the integer endpoints of the block in all dimensions
! the array blkLimits returns the interior end points
! whereas array blkLimitsGC returns endpoints including guardcells
call Grid\_getBlkIndexLimits(blockId,blkLimits,blkLimitsGC)

! get the size along each dimension for allocation and then allocate
sizeX = blkLimitsGC(HIGH,IAXIS)
sizeY = blkLimitsGC(HIGH,JAXIS)
sizeZ = blkLimitsGC(HIGH,KAXIS)
allocate(xLeft(sizeX))
allocate(xRight(sizeX))
allocate(xCenter(sizeX))
allocate(yCoord(sizeY))
allocate(zCoord(sizeZ))

The next part of the routine involves setting up the initial conditions. This section could be code for
interpolating a given set of initial conditions, constructing some analytic model, or reading in a table of initial
values. In the present example, we begin by getting the coordinates for the cells in the current block. This
is done by a set of calls to \texttt{Grid\_getCellCoords}. Next we create loops that compute appropriate values for
each grid point, since we are constructing initial conditions from a model. Note that we use the \texttt{blkLimits}
array from \texttt{Grid\_getBlkIndexLimits} in looping over the spatial indices to initialize only the interior cells
in the block. To initialize the entire block, including the guardcells, the \texttt{blkLimitsGC} array should be used.

xCoord(:) = 0.0
yCoord(:) = 0.0
zCoord(:) = 0.0
CHAPTER 3. SETTING UP NEW PROBLEMS

call Grid_getCellCoords(IAXIS, blockID, LEFT_EDGE, gcell, xLeft, sizeX)
call Grid_getCellCoords(IAXIS, blockID, CENTER, gcell, xCenter, sizeX)
call Grid_getCellCoords(IAXIS, blockID, RIGHT_EDGE, gcell, xRight, sizeX)
call Grid_getCellCoords(JAXIS, blockID, CENTER, gcell, yCoord, sizeY)
call Grid_getCellCoords(KAXIS, blockID, CENTER, gcell, zCoord, sizeZ)

!-----------------------------------------------------------------------------
! loop over all of the zones in the current block and set the variables.
!-----------------------------------------------------------------------------
do k = blkLimits(LOW,KAXIS),blkLimits(HIGH,KAXIS)
   zz = zCoord(k) ! coordinates of the cell center in the z-direction
   lPosn0 = sim_posn - zz*sim_zCos/sim_xCos ! Where along the x-axis
   ! the shock intersects
   ! the xz-plane at the current z.

do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
   yy = yCoord(j) ! center coordinates in the y-direction
   lPosn = lPosn0 - yy*sim_yCos/sim_xCos ! The position of the
   ! shock in the current yz-row.
   dataSize = 1 ! for Grid put data function, we are
   ! initializing a single cell at a time and
   ! sending it to Grid

do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)
   xx = xCenter(i) ! center coordinate along x
   xxL = xLeft(i) ! left coordinate along y
   xxR = xRight(i) ! right coordinate along z

For the present problem, we create a discontinuity along the shock plane. We do this by initializing
the grid points to the left of the shock plane with one value, and the grid points to the right of the shock
plane with another value. Recall that the runtime parameters which provide these values are available to
us through the Simulation_data module. At this point we can initialize all independent physical variables
at each grid point. The following code shows the contents of the loops. Don’t forget to store the calculated
values in the Grid data structure!

if (xxR <= lPosn) then
   rhoZone = sim_rhoLeft
   presZone = sim_pLeft
   velxZone = sim_uLeft * sim_xCos
   velyZone = sim_uLeft * sim_yCos
   velzZone = sim_uLeft * sim_zCos

   ! initialize cells which straddle the shock. Treat them as though 1/2 of
   ! the cell lay to the left and 1/2 lay to the right.
elseif ((xxL < lPosn) .and. (xxR > lPosn)) then
   rhoZone = 0.5 * (sim_rhoLeft+sim_rhoRight)
   presZone = 0.5 * (sim_pLeft+sim_pRight)
   velxZone = 0.5 *(sim_uLeft+sim_uRight) * sim_xCos
   velyZone = 0.5 *(sim_uLeft+sim_uRight) * sim_yCos
   velzZone = 0.5 *(sim_uLeft+sim_uRight) * sim_zCos
When \texttt{Simulation\_initBlock} returns, the Grid data structures for physical variables have the values of the initial model for the current block. As mentioned before, \texttt{Simulation\_initBlock} is called for every block that is created as the code refines the initial model.

3.6 The runtime parameter file (\texttt{flash.par})

The \texttt{FLASH} executable expects a \texttt{flash.par} file to be present in the \texttt{run} directory, unless another name for the runtime input file is given as a command-line option. This file contains runtime parameters, and thus provides a mechanism for partially controlling the runtime environment. The names of runtime parameters are case-insensitive. Copies of \texttt{flash.par} are kept in their respective Simulation directories for easy distribution.

The \texttt{flash.par} file for the example setup is:

\begin{verbatim}
# Density, pressure, and velocity on either side of interface
sim_rhoeLeft = 1.
sim_rhoeRight = 0.125
sim_pLeft = 1.
\end{verbatim}
sim_pRight   = 0.1
sim_uLeft    = 0.
sim_uRight   = 0.

# Angle and position of interface relative to x and y axes
sim_xangle  = 0
sim_yangle  = 90.
sim_posn    = 0.5

# Gas ratio of specific heats
gamma      = 1.4

gamma      = 1.4

geometry   = cartesian

# Size of computational volume
xmin       = 0.
xmax       = 1.
ymin       = 0.
ymax       = 1.

# Boundary conditions
xl_boundary_type = "outflow"
xr_boundary_type = "outflow"

yl_boundary_type = "outflow"
yr_boundary_type = "outflow"

# Simulation (grid, time, I/O) parameters
cfl         = 0.8
basenm      = "sod_"
restart     = .false.

# checkpoint file output parameters
checkpointFileIntervalTime = 0.2
checkpointFileIntervalStep = 0
checkpointFileNumber = 0

# plotfile output parameters
plotfileIntervalTime = 0.
plotfileIntervalStep = 0
plotfileNumber = 0

nend        = 1000
tmax        = .2

run_comment = "Sod problem, parallel to x-axis"
log_file    = "sod.log"
eint_switch = 1.e-4

plot_var_1  = "dens"
plot_var_2  = "pres"
plot_var_3  = "temp"
3.7 Running your simulation

You can run your simulation either in the object directory or in a separate run directory.

Running in the object directory is especially convenient for development and testing. The command for starting a FLASH simulation may be system dependent, but typically you would type something like

    mpirun -np N flash4

or

    mpiexec -n N flash4

To start a run on N processing entities. On many systems, you can also simply use

    ./flash4
Figure 3.1: Image of the initial distribution of density in example setup.
3.7. RUNNING YOUR SIMULATION

to start a run on 1 processing entity, i.e., without process-level parallelisation.

If you want to invoke FLASH in a separate run directory, the best way is to copy the `flash4` binary into the run directory, and then proceed as above. However, before starting FLASH you also need to do the following:

- Copy a FLASH parfile, normally `flash.par`, into the run directory.

- If FLASH was configured with `Paramesh4dev` (e.g., via the setup shortcut `+pm4dev`) or with `Paramesh4.0` in LIBRARY mode: copy the file `amr_runtime_parameters`, into the run directory. This file should have been generated in the object directory by running `./setup`.

- Some code units use additional data or table files, which are also copied into the object directory by `./setup`. (These files typically match one of the patterns `*.dat` or `*_table*`) copy or move those files into the run directory, too, if your simulation needs them.
Part II

The FLASH Software System
Chapter 4

Overview of FLASH architecture

The files that make up the FLASH source are organized in the directory structure according to their functionality and grouped into components called units. Throughout this manual, we use the word ‘unit’ to refer to a group of related files that control a single aspect of a simulation, and that provide the user with an interface of publicly available functions. FLASH can be viewed as a collection of units, which are selectively grouped to form one application.

A typical FLASH simulation requires only a subset of all of the units in the FLASH code. When the user gives the name of the simulation to the setup tool, the tool locates and brings together the units required by that simulation, using the FLASH Config files (described in Chapter 5) as a guide. Thus, it is important to distinguish between the entire FLASH source code and a given FLASH application. the FLASH units can be broadly classified into five functionally distinct categories: infrastructure, physics, monitor, driver, and simulation.

The infrastructure category encompasses the units responsible for FLASH housekeeping tasks such as the management of runtime parameters, the handling of input and output to and from the code, and the administration of the grid, which describes the simulation’s physical domain.

Units in the physics category such as Hydro (hydrodynamics), Eos (equation of state), and Gravity implement algorithms to solve the equations describing specific physical phenomena.

The monitoring units Logfile, Profiler, and Timers track the progress of an application, while the Driver unit implements the time advancement methods and manages the interaction between the included units.

The simulation unit is of particular significance because it defines how a FLASH application will be built and executed. When the setup script is invoked, it begins by examining the simulation’s Config file, which specifies the units required for the application, and the simulation-specific runtime parameters. Initial conditions for the problem are provided in the routines Simulation_init and Simulation_initBlock. As mentioned in Chapter 3, the Simulation unit allows the user to overwrite any of FLASH’s default function implementations by writing a function with the same name in the application-specific directory. Additionally, runtime parameters declared in the simulation’s Config file override definitions of same-named parameters in other FLASH units. These helpful features enable users to customize their applications, and are described in more detail below in Section 4.1 and online in Architecture Tips. The simulation unit also provides some useful interfaces for modifying the behaviour of the application while it is running. For example there is an interface Simulation_adjustEvolution which is called at every time step. Most applications would use the null implementation, but its implementation can be placed in the Simulation directory of the application to customize behavior. The API functions of the Simulation unit are unique in that except Simulation_initSpecies, none of them have any default general implementations. At the API level there are the null implementations, actual implementations exist only for specific applications. The general implementations of Simulation_initSpecies exist for different classes of applications, such as those utilizing nuclear burning or ionization.
Why the name change from “modules” in FLASH2 to “units” in FLASH3? The term “module” caused confusion among users and developers because it could refer both to a FORTRAN90 module and to the FLASH-specific code entity. In order to avoid this problem, FLASH3 started using the word “module” to refer exclusively to an F90 module, and the word “unit” for the basic FLASH code component. Also, FLASH no longer uses F90 modules to implement units. Fortran’s limitation of one file per module is too restrictive for some of FLASH4’s units, which are too complex to be described by a single file. Instead, FLASH4 uses interface blocks, which enable the code to take advantage of some of the advanced features of FORTRAN90, such as pointer arguments and optional arguments. Interface blocks are used throughout the code, even when such advanced features are not called for. For a given unit, the interface block will be supplied in the file "Unit_interface.F90". Please note that files containing calls to API-level functions must include the line use Unit, ONLY: function-name1, function-name2, etc. at the top of the file.

4.1 FLASH Inheritance

FORTRAN90 is not an object-oriented language like Java or C++, and as such does not implement those languages’ characteristic properties of inheritance. But FLASH takes advantage of the Unix directory structure to implement an inheritance hierarchy of its own. Every child directory in a unit’s hierarchy inherits all the source code of its parent, thus eliminating duplication of common code. During setup, source files in child directories override same-named files in the parent or ancestor directories.

Similarly, when the setup tool parses the source tree, it treats each child or subdirectory as inheriting all of the Config and Makefile files in its parent’s directory. While source files at a given level of the directory hierarchy override files with the same name at higher levels, Makefiles and configuration files are cumulative. Since functions can have multiple implementations, selection for a specific application follows a few simple rules applied in order described in Architecture Tips.

However, we must take care that this special use of the directory structure for inheritance does not interfere with its traditional use for organization. We avoid any problems by means of a careful naming convention that allows clear distinction between organizational and namespace directories.

To briefly summarize the convention, which is described in detail online in Architecture Tips, the top level directory of a unit shares its name with that of the unit, and as such always begins with a capital letter. Note, however, that the unit directory may not always exist at the top level of the source tree. A class of units may also be grouped together and placed under an organizational directory for ease of navigation; organizational directories are given in lower case letters. For example the grid management unit, called Grid, is the only one in its class, and therefore its path is source/Grid, whereas the hydrodynamics unit, Hydro, is one of several physics units, and its top level path is source/physics/Hydro. This method for distinguishing between organizational directories and namespace directories is applied throughout the entire source tree.

4.2 Unit Architecture

A FLASH unit defines its own Application Programming Interface (API), which is a collection of routines the unit exposes to other units in the code. A unit API is usually a mix of accessor functions and routines which modify the state of the simulation.

A good example to examine is the Grid unit API. Some of the accessor functions in this unit are GridgetCellCoords, GridGetBlkData, and GridPutBlkData, while GridFillGuardCells and GridUpdateRefinement are examples of API routines which modify data in the Grid unit.

A unit can have more than one implementation of its API. The Grid Unit, for example, has both an Adaptive Grid and a Uniform Grid implementation. Although the implementations are different, they
both conform to the Grid API, and therefore appear the same to the outside units. This feature allows users to easily swap various unit implementations in and out of a simulation without affecting the way other units communicate. Code does not have to be rewritten if the users decides to implement the uniform grid instead of the adaptive grid.

4.2.1 Stub Implementations

Since routines can have multiple implementations, the setup script must select the appropriate implementation for an application. The selection follows a few simple rules described in Architecture Tips. The top directory of every unit contains a stub or null implementation of each routine in the Unit’s API. The stub functions essentially do nothing. They are coded with just the declarations to provide the same interface to callers as a corresponding “real” implementation. They act as function prototypes for the unit. Unlike true prototypes, however, the stub functions assign default values to the output-only arguments, while leaving the other arguments unaltered. The following snippet shows a simplified example of a stub implementation for the routine Grid_getListOfBlocks.

```fortran
subroutine Grid_getListOfBlocks(blockType, listOfBlocks, count)
    implicit none

    integer, intent(in) :: blockType
    integer,dimension(*),intent(out) :: listOfBlocks
    integer, intent(out) :: count

    count=0
    listOfBlocks(1)=0

    return
end subroutine Grid_getListOfBlocks
```

While a set of null implementation routines at the top level of a unit may seem like an unnecessary added layer, this arrangement allows FLASH to include or exclude units without the need to modify any existing code. If a unit is not included in a simulation, the application will be built with its stub functions. Similarly, if a specific implementation of the unit finds some of the API functions irrelevant, it need not provide any implementations for them. In those situations, the applications include stubs for the unimplemented functions, and full implementations of all the other ones. Since the stub functions do return valid values when called, unexpected crashes from un-initialized output arguments are avoided.

The Grid_updateRefinement routine is a good example of how stub functions can be useful. In the case of a simulation using an adaptive grid, such as PARAMESH, the routine Driver_evolveFlash calls Grid_updateRefinement to update the grid’s spacing. The Uniform Grid however, needs no such routine because its grid is fixed. There is no error, however, when Driver_evolveFlash calls Grid_updateRefinement during a Uniform Grid simulation, because the stub routine steps in and simply returns without doing anything. Thus the stub layer allows the same Driver_evolveFlash routine to work with both the Adaptive Grid and Uniform Grid implementations.

FLASH Transition

While the concept of “null” or “stub” functions existed in FLASH2, FLASH3 formalized it by requiring all units to publish their API (the complete Public Interface) at the top level of a unit’s directory. Similarly, the inheritance through Unix directory structure in FLASH4 is essentially the same as that of FLASH2, but the introduction of a formal naming convention has clarified it and made it easier to follow. The complete API can be found online at http://flash.uchicago.edu/site/flashcode/user_support/.
4.2.2 Subunits

One or more subunits sit under the top level of a unit. Among them the unit’s complete API is implemented. The subunits are considered peers to one another. Each subunit must implement at least one API function, and no two subunits can implement the same API function. The division of a unit into subunits is based upon identifying self-contained subsets of its API. In some instances, a subunit may be completely excluded from a simulation, thereby saving computational resources. For example, the Grid unit API includes a few functions that are specific to Lagrangian tracer particles, and are therefore unnecessary to simulations that do not utilize particles. By placing these routines in the GridParticles subunit, it is possible to easily exclude them from a simulation. The subunits have composite names; the first part is the unit name, and the second part represents the functionality that the subunit implements. The primary subunit is named UnitMain, which every unit must have. For example, the main subunit of Hydro unit is HydroMain and that of the Eos unit is EosMain.

In addition to the subunits, the top level unit directory may contain a subdirectory called localAPI. This subdirectory allows a subunit to create a public interface to other subunits within its own unit; all stub implementations of the subunit public interfaces are placed in localAPI. External units should not call routines listed in the localAPI; for this reason these local interfaces are not shown in the general source API tree.

A subunit can have a hierarchy of its own. It may have more than one unit implementation directories with alternative implementations of some of its functions while other functions may be common between them. FLASH exploits the inheritance rules described in Architecture Tips. For example, the Grid unit has three implementations for GridMain: the Uniform Grid (UG), PARAMESH 2, and PARAMESH 4. The procedures to apply boundary conditions are common to all three implementations, and are therefore placed directly in GridMain. In addition, GridMain contains two subdirectories. One is UG, which has all the remaining implementations of the API specific to the Uniform Grid. The other directory is organized as paramesh, which in turn contains two directories for the package of PARAMESH 2 and another organizational directory paramesh4. Finally, paramesh4 has two subdirectories with alternative implementations of the PARAMESH 4 package. The directory paramesh also contains all the function implementations that are common between PARAMESH 2 and PARAMESH 4. Following the naming convention described in Architecture Tips, paramesh is all lowercase, since it has child directories that have some API implementation. The namespace directories Paramesh2, Paramesh4.0 and Paramesh4dev contain functions unique to each implementation. An example of a unit hierarchy is shown in Figure 4.1. The kernels are described below in Section 4.2.4.

4.2.3 Unit Data Modules, _init, and _finalize routines

Each unit must have a F90 data module to store its unit-scope local data and an _init file to initialize it. The _init routines are called by the Driver unit once by the routine Driver_initFlash at the start of a simulation. They get unit specific runtime parameters from the RuntimeParameters unit and store them in the unit data module.

Every unit implementation directory of UnitMain, must either inherit a Unit_data module, or have its own. There is no restriction on additional unit scope data modules, and individual Units determine how best to manage their data. Other subunits and the underlying computational kernels can have their own data modules, but the developers are encouraged to keep these data modules local to their subunits and kernels for clarity and maintainability of the code. It is strongly recommended that only the data modules in the Main subunit be accessible everywhere in the unit. However, no data module of a unit may be known to any other unit. This restriction is imposed to keep the units encapsulated and their data private. If another part of the code needs access to any of the unit data, it must do so through accessor functions.

Additionally, when routines use data from the unit’s data module the convention is to indicate what particular data is being used with the ONLY keyword, as in use Unit_data, ONLY: un_someData. See the snippet of code below for the correct convention for using data from a unit’s FORTRAN Data Module.

```fortran
subroutine Driver_evolveFlash()
    use Driver_data, ONLY: dr_myPE, dr_numProcs, dr_nbegin, &
        dr_nend, dr_dt, dr_wallClockTimeLimit, &
```
dr_tmax, dr_simTime, dr_redshift, &
    dr_nstep, dr_dtOld, dr_dtNew, dr_restart, dr_elapsedWCTime

implicit none

integer :: localNumBlocks

Each unit must also have a Unit_finalize routine to clean up the unit at the termination of a FLASH run. The finalization routines might deallocate space or write out completion messages.

### 4.2.4 Private Routines: kernels and helpers

All routines in a unit that do not implement the API are classified as private routines. They are divided into two broad categories: the *kernel* is the collection of routines that implement the unit’s core functionality and solvers, and *helper* routines are supplemental to the unit’s API and sometimes act as a conduit to its kernel. A helper function is allowed to know the other unit’s APIs but is itself known only locally within the unit. The concept of helper functions allows minimization of the unit APIs, which assists in code maintenance. The helper functions follow the convention of starting with an “un” in their name, where “un” is in some way derived from the unit name. For example, the helper functions of the Grid unit start with gr_, and those of Hydro unit start with hy_. The helper functions have access to the unit’s data module, and they are also allowed to query other units for the information needed by the kernel, by using their accessor functions. If the kernel has very specific data structures, the helper functions can also populate them with the collected information. An example of a helper function is gr_expandDomain, which refines an AMR block. After refinement, equations of state usually need to be called, so the routine accesses the EOS routines via Eos_wrapped.

The concept of kernels, on the other hand, facilitates easy import of third party solvers and software into FLASH. The kernels are not required to follow either the naming convention or the inheritance rules of the FLASH architecture. They can have their own hierarchy and data modules, and the top level of
the kernel typically resides at leaf level of the FLASH unit hierarchy. This arrangement allows FLASH to import a solver without having to modify its internal code, since API and helper functions hide the higher level details from it, and hide its details from other units. However, developers are encouraged to follow the helper function naming convention in the kernel where possible to ease code maintenance.

The Grid unit and the Hydro unit both provide very good examples of private routines that are clearly distinguishable between helper functions and kernel. The AMR version of the Grid unit imports the PARAMESH version 2 library as a vendor supplied branch in our repository. It sits under the lowest namespace directory Paramesh2 in Grid hierarchy and maintains the library’s original structure. All other private functions in the paramesh branch of Grid are helper functions and their names start with gr. In the Hydro unit the entire hydrodynamic solver resides under the directory PPM, which was imported from the PROMETHEUS code (see Section 14.1.2). PPM is a directional solver and requires that data be passed to it in vector form. Routines like hy_sweep and hy_block are helper functions that collect data from the Grid unit, and put it in the format required by PPM. These routines also make sure that data stay in thermodynamic equilibrium through calls to the Eos unit. Neither PARAMESH 2, nor PPM has any knowledge of units outside their own.

4.3 Unit Test Framework

In keeping with good software practice, FLASH4 incorporates a unit test framework that allows for rigorous testing and easy isolation of errors. The components of the unit test show up in two different places in the FLASH source tree. One is a dedicated path in the Simulation unit, Simulation/SimulationMain/unitTest/UnitTestName, where UnitTestName is the name of a specific unit test. The other place is a subdirectory called unitTest, somewhere in the hierarchy of the corresponding unit which implements a function Unit_unitTest and any helper functions it may need. The primary reason for organizing unit tests in this somewhat confusing way is that unit tests are special cases of simulation setups that also need extensive access to internal data of the unit being tested. By splitting the unit test into two places, it is possible to meet both requirements without violating unit encapsulation. We illustrate the functioning of the unit test framework with the unit test of the Eos unit. For more details please see Section 15.6. The Eos unit test needs its own version of the routine Driver_evolveFlash which makes a call to its Eos_unitTest routine. The initial conditions specification and unit test specific Driver_evolveFlash are placed in Simulation/SimulationMain/unitTest/Eos, since the Simulation unit allows any substitute FLASH function to be placed in the specific simulation directory. The function Eos_unitTest resides in physics/Eos/unitTest, and therefore has access to all internal Eos data structures and helper functions.
Chapter 5

The FLASH configuration script (setup)

The setup script, found in the FLASH root directory, provides the primary command-line interface to configuring the FLASH source code. It is important to remember that the FLASH code is not a single application, but a set of independent code units which can be put together in various combinations to create a multitude of different simulations. It is through the setup script that the user controls how the various units are assembled.

The primary job of the setup script is to

- traverse the FLASH source tree and link necessary files for a given application to the object/ directory
- find the target Makefile.h for a given machine.
- generate the Makefile that will build the FLASH executable.
- generate the files needed to add runtime parameters to a given simulation.
- generate the files needed to parse the runtime parameter file.

More description of how setup and the FLASH4 architecture interact may be found in Chapter 4. Here we describe its usage.

The setup script determines site-dependent configuration information by looking for a directory sites/<hostname> where <hostname> is the hostname of the machine on which FLASH is running.\(^1\) Failing this, it looks in sites/Prototypes/ for a directory with the same name as the output of the uname command. The site and operating system type can be overridden with the -site and -ostype command-line options to the setup command. Only one of these options can be used at one time. The directory for each site and operating system type contains a makefile fragment Makefile.h that sets command names, compiler flags, library paths, and any replacement or additional source files needed to compile FLASH for that specific machine and machine type.

The setup script uses the contents of the problem directory and the site/OS type, together with a Units file, to generate the object/ directory, which contains links to the appropriate source files and makefile fragments. The Units file lists the names of all units which need to be included while building the FLASH application. This file is automatically generated when the user commonly provides the command-line -auto option, although it may be assembled by hand. When -auto option is used, the setup script starts with the Config file of the problem specified, finds its REQUIRED units and then works its way through their Config files. This process continues until all the dependencies are met and a self-consistent set of units has been found. At the end of this automatic generation, the Units file is created and placed in the object/ directory, where it can be edited if necessary. setup also creates the master makefile (object/Makefile) and several FORTRAN include files that are needed by the code in order to parse the runtime parameters. After running setup, the user can create the FLASH executable by running gmake in the object directory.

\(^1\) if a machine has multiple hostnames, setup tries them all
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

FLASH Transition

In FLASH2, the Units file was located in the FLASH root directory. In FLASH4, this file is found in the object/ directory.

Save some typing

- All the setup options can be shortened to unambiguous prefixes, e.g. instead of 
  
  ./setup -auto <problem-name>

  one can just say

  ./setup -a <problem-name>

  since there is only one setup option starting with a.

- The same abbreviation holds for the problem name as well. 
  
  ./setup

  -a IsentropicVortex can be abbreviated to

  ./setup -a Isen assuming that

  IsentropicVortex is the only problem name which starts with Isen.

- Unit names are usually specified by their paths relative to the source directory. However, setup also allows unit names to be prefixed with an extra “source/”, allowing you to use the TAB-completion features of your shell like this

  ./setup -a Isen -unit=sou<TAB>rce/IO/IOM<TAB>ain/hd<TAB>f5

- If you use a set of options repeatedly, you can define a shortcut for them. FLASH4 comes with a number of predefined shortcuts that significantly simplify the setup line, particularly when trying to match the Grid with a specific I/O implementation. For more details on creating shortcuts see Section 5.3. For detailed examples of I/O shortcuts please see Section 9.1 in the I/O chapter.

Reduce compilation time

- To reuse compiled code when changing setup configurations, use the -noclobber setup option. For details see Section 5.2.

5.1 Setup Arguments

The setup script accepts a large number of command line arguments which affect the simulation in various ways. These arguments are divided into three categories:

1. Setup Options (example: -auto) begin with a dash and are built into the setup script itself. Many of the most commonly used arguments are setup options.

2. Setup Variables (example: species=air,h2o) are defined by individual units. When writing a Config file for any unit, you can define a setup variable. Section 5.4 explains how setup variables can be created and used.

3. Setup Shortcuts (example: +ug) begin with a plus symbol and are essentially macros which automatically include a set of setup variables and/or setup options. New setup shortcuts can be easily defined, see Section 5.3 for more information.

Table 5.1 shows a list of some of the basic setup arguments that every FLASH user should know about. A comprehensive list of all setup arguments can be found in Section 5.2 alongside more detailed descriptions of these options.
Table 5.1: List of Commonly Used Setup Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-auto</td>
<td>this option should almost always be set</td>
</tr>
<tr>
<td>-unit=&lt;unit&gt;</td>
<td>include a specified unit</td>
</tr>
<tr>
<td>-objdir=&lt;dir&gt;</td>
<td>specify a different object directory location</td>
</tr>
<tr>
<td>-debug</td>
<td>compile for debugging</td>
</tr>
<tr>
<td>-opt</td>
<td>enable compiler optimization</td>
</tr>
<tr>
<td>-n[xyb]b=#</td>
<td>specify block size in each direction</td>
</tr>
<tr>
<td>-maxblocks=#</td>
<td>specify maximum number of blocks per process</td>
</tr>
<tr>
<td>-d</td>
<td>specify number of dimensions</td>
</tr>
<tr>
<td>+cartesian</td>
<td>use Cartesian geometry</td>
</tr>
<tr>
<td>+cylindrical</td>
<td>use cylindrical geometry</td>
</tr>
<tr>
<td>+polar</td>
<td>use polar geometry</td>
</tr>
<tr>
<td>+spherical</td>
<td>use spherical geometry</td>
</tr>
<tr>
<td>+noio</td>
<td>disable IO</td>
</tr>
<tr>
<td>+ug</td>
<td>use the uniform grid in a fixed block size mode</td>
</tr>
<tr>
<td>+nofbs</td>
<td>use the uniform grid in a non-fixed block size mode</td>
</tr>
<tr>
<td>+pm2</td>
<td>use the PARAMESH2 grid</td>
</tr>
<tr>
<td>+pm40</td>
<td>use the PARAMESH4.0 grid</td>
</tr>
<tr>
<td>+pm4dev</td>
<td>use the PARAMESH4DEV grid</td>
</tr>
<tr>
<td>+usm</td>
<td>use the Unsplit Staggered Mesh MHD solver</td>
</tr>
<tr>
<td>+unsplitHydro</td>
<td>use the Unsplit Hydro solver</td>
</tr>
</tbody>
</table>

5.2 Comprehensive List of Setup Arguments

- **verbose=<verbosity>**

Normally setup prints summary messages indicating its progress. Use the -verbose to make the messages more or less verbose. The different levels (in order of increasing verbosity) are ERROR, IMPINFO, WARN, INFO, DEBUG. The default is WARN.

- **auto**

Normally, setup requires that the user supply a plain text file called Units (in the object directory) that specifies the units to include. A sample Units file appears in Figure 5.1. Each line is either a comment (preceded by a hash mark (#)) or the name of a an include statement of the form INCLUDE unit. Specific implementations of a unit may be selected by specifying the complete path to the implementation in question; If no specific implementation is requested, setup picks the default listed in the unit’s Config file.

---

Formerly, (in FLASH2) it was located in the FLASH root directory
The `-auto` option enables `setup` to generate a “rough draft” of a `Units` file for the user. The Config file for each problem setup specifies its requirements in terms of other units it requires. For example, a problem may require the perfect-gas equation of state (physics/Eos/EosMain/Gamma) and an unspecified hydro solver (physics/Hydro). With `-auto`, `setup` creates a `Units` file by converting these requirements into unit include statements. Most users configuring a problem for the first time will want to run `setup` with `-auto` to generate a `Units` file and then to edit it directly to specify alternate implementations of certain units. After editing the `Units` file, the user must re-run `setup` without `-auto` in order to incorporate his/her changes into the code configuration. The user may also use the command-line option `-with-unit=<path>` in conjunction with the `-auto` option, in order to pick a specific implementation of a unit, and thus eliminate the need to hand-edit the `Units` file.

`-[123]d`

By default, `setup` creates a makefile which produces a FLASH executable capable of solving two-dimensional problems (equivalent to `-2d`). To generate a makefile with options appropriate to three-dimensional problems, use `-3d`. To generate a one-dimensional code, use `-1d`. These options are mutually exclusive and cause `setup` to add the appropriate compilation option to the makefile it generates.

`-maxblocks=#`

This option is also used by `setup` in constructing the makefile compiler options. It determines the amount of memory allocated at runtime to the adaptive mesh refinement (AMR) block data structure. For example, to allocate enough memory on each processor for 500 blocks, use `-maxblocks=500`. If the default block buffer size is too large for your system, you may wish to try a smaller number here; the default value depends upon the dimensionality of the simulation and the grid type. Alternatively, you may wish to experiment with larger buffer sizes, if your system has enough memory. A common cause of aborted simulations occurs when the AMR grid creates greater than `maxblocks` during refinement. Ressetup the simulation using a larger value of this option.

`-nxb=# -nyb=# -nzb=#`

These options are used by `setup` in constructing the makefile compiler options. The mesh on which the problem is solved is composed of blocks, and each block contains some number of cells. The `-nxb`, `-nyb`, and `-nzb` options determine how many cells each block contains (not counting guard cells). The default value for each is 8. These options do not have any effect when running in Uniform Grid non-fixed block size mode.

`[-debug|-opt|-test]`

The default Makefile built by `setup` will use the optimized setting (`-opt`) for compilation and linking. Using `-debug` will force `setup` to use the flags relevant for debugging (e.g., including `-g` in the compilation line). The user may use the option `-test` to experiment with different combinations of compiler and linker options. Exactly which compiler and linker options are associated with each of these flags is specified in `sites/<hostname>/Makefile*` where `<hostname>` is the hostname of the machine on which FLASH is running. For example, to tell an Intel Fortran compiler to use real numbers of size 64 when the `-test` option is specified, the user might add the following line to his/her `Makefile.h`:

```plaintext
FFLAGS_TEST = -real_size 64
```

`-objdir=<dir>`

 Overrides the default object directory with `<dir>`. Using this option allows you to have different simulations configured simultaneously in the FLASH4 distribution directory.

`-with-unit=<unit>, -unit=<unit>`

Use the specified `<unit>` in setting up the problem.
# Units file for Sod generated by setup

```
#Units file for Sod generated by setup

INCLUDE Driver/DriverMain/Split
INCLUDE Grid/GridBoundaryConditions
INCLUDE Grid/GridMain/paramesh/interpolation/Paramesh4/prolong
INCLUDE Grid/GridMain/paramesh/interpolation/prolong
INCLUDE Grid/GridMain/paramesh/paramesh4/Paramesh4.0/PM4_package/headers
INCLUDE Grid/GridMain/paramesh/paramesh4/Paramesh4.0/PM4_package/mpi_source
INCLUDE Grid/GridMain/paramesh/paramesh4/Paramesh4.0/PM4_package/source
INCLUDE Grid/GridMain/paramesh/paramesh4/Paramesh4.0/PM4_package/utilities/multigrid
INCLUDE Grid/localAPI
INCLUDE IO/IOMain/hdf5/serial/PM
INCLUDE IO/localAPI
INCLUDE PhysicalConstants/PhysicalConstantsMain
INCLUDE RuntimeParameters/RuntimeParametersMain
INCLUDE Simulation/SimulationMain/Sod
INCLUDE flashUtilities/contiguousConversion
INCLUDE flashUtilities/general
INCLUDE flashUtilities/interpolation/oneDim
INCLUDE flashUtilities/nameValueLL
INCLUDE monitors/Logfile/LogfileMain
INCLUDE monitors/Timers/TimersMain/MPINative
INCLUDE physics/Eos/EosMain/Gamma
INCLUDE physics/Hydro/HydroMain/split/PPM/PPMKernel
```

Figure 5.1: Example of the Units file used by setup to determine which Units to include

- **-curvilinear**
  
  Enable code in PARAMESH 4 that implements geometrically correct data restriction for curvilinear coordinates. This setting is automatically enabled if a non-cartesian geometry is chosen with the -geometry flag; so specifying -curvilinear only has an effect in the Cartesian case.

- **defines=<def>[,<def>]...**
  
  <def> is of the form SYMBOL or SYMBOL=value. This causes the specified pre-processor symbols to be defined when the code is being compiled. This is mainly useful for debugging the code. For e.g., -defines=DEBUG_ALL turns on all debugging messages. Each unit may have its own DEBUG_UNIT flag which you can selectively turn on.

- **[-fbs|nofbs]**
  
  Causes the code to be compiled in fixed-block or non-fixed-block size mode. Fixed-block mode is the default. In non-fixed block size mode, all storage space is allocated at runtime. This mode is available only with Uniform Grid.

- **-geometry=<geometry>**
  
  Choose one of the supported geometries cartesian, cylindrical, spherical, or polar. Some Grid implementations require the geometry to be known at compile-time while others don’t. This setup option can be used in either case; it is a good idea to specify the geometry here if it is known at setup-time. Choosing a non-Cartesian geometry here automatically sets the -gridinterpolation=monotonic option below.
-gridinterpolation=<scheme>

Select a scheme for Grid interpolation. Two schemes are currently supported:

- **monotonic**
  This scheme attempts to ensure that monotonicity is preserved in interpolation, so that interpolation does not introduce small-scale non-monotonicity in the data. The **monotonic** scheme is required for curvilinear coordinates and is automatically enabled if a non-cartesian geometry is chosen with the `-geometry` flag. For AMR Grid implementations, this flag will automatically add additional directories so that appropriate data interpolation methods are compiled in. The **monotonic** scheme is the default (by way of the `+default` shortcut), unlike in FLASH2.

- **native**
  Enable the interpolation that is native to the AMR Grid implementation (PARAMESH 2 or PARAMESH 4) by default. This option is only appropriate for Cartesian geometries.

### Change in Interpolation

Note that the default interpolation behavior has changed as of the FLASH3 beta release: the **native** interpolation used to be default.

### When to use native Grid interpolation

The **monotonic** interpolation method requires more layers of coarse guard cells next to a coarse guard cell in which interpolation is to be applied. It may therefore be necessary to use the **native** method if a simulation is set up to include fewer than four layers of guard cells.

-makefile=<extension>

Setu normally uses the Makefile.h from the directory determined by the hostname of the machine and the `-site` and `-os` options. If you have multiple compilers on your machine you can create Makefile.h.<extension> for different compilers. e.g., you can have a Makefile.h and Makefile.h.intel and Makefile.h.lahey for the three different compilers. setup will still use the Makefile.h file by default, but supplying `-makefile=intel` on the command-line causes setup to use Makefile.h.intel instead.

-index-reorder

Instructs setup that indexing of unk and related arrays should be changed. This may be needed in FLASH4 for compatibility with alternative grids. This is supported by both the Uniform Grid as well as PARAMESH, and is currently required for the Chombo grid.

-makehide

Ordinarily, the commands being executed during compilation of the FLASH executable are sent to standard out. It may be that you find this distracting, or that your terminal is not able to handle these long lines of display. Using the option `-makehide` causes setup to generate a Makefile so that gmake only displays the names of the files being compiled and not the exact compiler call and flags. This information remains available in setup_flags in the object/ directory.
5.2. COMPREHENSIVE LIST OF SETUP ARGUMENTS

-noclobber

`setup` normally removes all code in the `object` directory before linking in files for a simulation. The ensuing `gmake` must therefore compile all source files anew each time `setup` is run. The `-noclobber` option prevents `setup` from removing compiled code which has not changed from the previous `setup` in the same directory. This can speed up the `gmake` process significantly.

-os=<os>

If `setup` is unable to find a correct `sites/` directory it picks the `Makefile` based on the operating system. This option instructs `setup` to use the default `Makefile` corresponding to the specified operating system.

-parfile=<filename>

This causes `setup` to copy the specified runtime-parameters file in the simulation directory to the `object` directory with the new name `flash.par`.

-particlemethods=TYPE=<particle type>[,INIT=<init method>][,MAP=<map method>][,ADV=<advance method>]

This option instructs `setup` to adjust the particle methods for a particular particle type. It can only be used when a particle type has already been registered with a `PARTICLETYPE` line in a `Config` file (see Section 6.6.1). A possible scenario for using this option involves the user wanting to use a different passive particle initialization method without modifying the `PARTICLETYPE` line in the simulation `Config` file. In this case, an additional `-particlemethods=TYPE=passive,INIT=cellmass` adjusts the initialization method associated with passive particles in the `setup` generated `Particles_specifyMethods()` subroutine. Since the specification of a method for mapping and initialization requires inclusions of appropriate implementations of `ParticlesMapping` and `ParticlesInitialization` subunits, and the specification of a method for time advancement requires inclusion of an appropriate implementation under `ParticlesMain`, it is the user’s responsibility to adjust the included units appropriately. For example a user may want want to override `Config` file defined particle type `passive` using lattice initialization `CellMassBins` density based distribution method using the `setup` command line. Here the user must first specify `-without-unit=Particles/ParticlesInitialization/Lattice` to exclude the lattice initialization, followed by `-with-unit=Particles/ParticlesInitialization/-WithDensity/CellMassBins` specification to include the appropriate implementation. In general, using command line overrides of `-particlemethods` are not recommended, as this option increases the chance of creating an inconsistent simulation setup. More information on multiple particle types can be found in Chapter 19, especially Section 19.3.

-portable

This option causes `setup` to create a portable `object` directory by copying instead of linking to the source files. The resulting `object` directory can be tarred and sent to another machine for actual compilation.

-site=<site>

`setup` searches the `sites/` directory for a directory whose name is the hostname of the machine on which `setup` is being run. This option tells `setup` to use the `Makefile` of the specified site. This option is useful if `setup` is unable to find the right hostname (which can happen on multiprocessor or laptop machines). Also useful when combined with the `-portable` option.

-unitsfile=<filename>

This causes `setup` to copy the specified file to the `object` directory as `Units` before setting up the problem. This option can be used when `-auto` is not used, to specify an alternate `Units` file.
-with-library=<libname>[,args], -library=<libname>[,args]
This option instructs setup to link in the specified library when building the final executable. A
library is a piece of code which is independent of FLASH. Internal libraries are those libraries whose
code is included with FLASH. The setup script supports external as well as internal libraries.
Information about external libraries is usually found in the site specific Makefile. The additional
args if any are library-specific and may be used to select among multiple implementations. For
more information see Library-HOWTO.

-tau=<makefile>
This option causes the inclusion of an additional Makefile necessary for the operation of Tau,
which may be used by the user to profile the code. More information on Tau can be found at
http://acts.nersc.gov/tau/

-without-library=<libname>
Negates a previously specified -with-library=<libname>[,args]

-without-unit=<unit>
This removes all units specified in the command line so far, which are children of the specified unit
(including the unit itself). It also negates any REQUESTS keyword found in a Config file for
units which are children of the specified unit. However it does not negate a REQUIRES keyword
found in a Config file.

+default
This shortcut specifies using basic default settings and is equivalent to the following:
--with-library=mpi +io +grid-gridinterpolation=monotonic

+noio
This shortcut specifies a simulation without IO and is equivalent to the following:
--without-unit=physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser/LaserIO
--without-unit=IO

+io
This shortcut specifies a simulation with basic IO and is equivalent to the following:
--with-unit=IO

+serialIO
This shortcut specifies a simulation using serial IO, it has the effect of setting the setup variable
parallelIO = False

+parallelIO
This shortcut specifies a simulation using serial IO, it has the effect of setting the setup variable
parallelIO = True

+hdf5
This shortcut specifies a simulation using hdf5 for compatible binary IO output, it has the effect
of setting the setup variable
IO = hdf5

+pnetcdf
This shortcut specifies a simulation using pnetcdf for compatible binary IO output, it has the
effect of setting the setup variable
IO = pnetcdf
5.2. COMPREHENSIVE LIST OF SETUP ARGUMENTS

+hdf5TypeIO
This shortcut specifies a simulation using hdf5, with parallel io capability for compatible binary
IO output, and is equivalent to the following:
+io +parallelIO +hdf5 typeIO=True

+pnetTypeIO
This shortcut specifies a simulation using pnetcdf, with parallel io capability for compatible binary
IO output, and is equivalent to the following:
+io +parallelIO +pnetcdf typeIO=True

+nolog
This shortcut specifies a simulation without log capability it is equivalent to the following:
-without-unit=monitors/Logfile

+grid
This shortcut specifies a simulation with the Grid unit, it is equivalent to the following:
-unit=Grid

+ug
This shortcut specifies a simulation using a uniform grid, it is equivalent to the following:
+grid Grid=UG

+pm2
This shortcut specifies a simulation using Paramesh2 for the grid, it is equivalent to the following:
+grid Grid=PM2

+pm40
This shortcut specifies a simulation using Paramesh4.0 for the grid, it is equivalent to the following:
+grid Grid=PM40

+pm3
This shortcut (for backward compatibility) specifies a simulation using Paramesh4.0 for the grid,
it is equivalent to the following:
+pm40

+chombo_ug
This shortcut specifies a simulation using a Chombo uniform grid, it is equivalent to the
following:
-unit=Grid/GridMain/Chombo/UG -index-reorder Grid=Chombo -maxblocks=1 -nofbs
-makefile=chombo chomboCompatibleHydro=True

+chombo_amr
This shortcut specifies a simulation using a Chombo amr grid, it is equivalent to the following:
-unit=Grid/GridMain/Chombo/AMR -index-reorder Grid=Chombo -nofbs
-makefile=chombo chomboCompatibleHydro=True

+pm4dev_clean
This shortcut specifies a simulation using a version of Paramesh 4 that is closer to the version
available on sourceforge. It is equivalent to:
+grid Grid=PM4DEV ParameshLibraryMode=True

+pm4dev
This shortcut specifies a simulation using a modified version of Paramesh 4 that includes a more
scalable way of filling the surr_biks array. It is equivalent to:
+pm4dev_clean FlashAvoidOrrery=True
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

+8wave

This shortcut specifies a MHD simulation using the 8wave mhd solver, which only works with
the native interpolation. It is equivalent to:

```
--with-unit=physics/Hydro/HydroMain/split/MHD_8Wave +grid
-gridinterpolation=native
```

+usm

This shortcut specifies a MHD simulation using the unsplit staggered mesh hydro solver, if pure
hydro mode is used with the USM solver add +pureHydro in the setup line. It is equivalent to:

```
--with-unit=physics/Hydro/HydroMain/unsplit/MHD_StaggeredMesh
--without-unit=physics/Hydro/HydroMain/split/MHD_8Wave
```

+pureHydro

This shortcut specifies using pure hydro mode, it is equivalent to:

```
physicsMode=hydro
```

+unsplitHydro

This shortcut specifies a simulation using the unsplit hydro solver and is equivalent to:

```
--with-unit=physics/Hydro/HydroMain/unsplit/Hydro_Unsplit
```

+uhd

This shortcut specifies a simulation using the unsplit hydro solver and is equivalent to:

```
--with-unit=physics/Hydro/HydroMain/unsplit/Hydro_Unsplit
```

+supportWENO

This shortcut specifies a simulation using a specific Hydro method that requires an increased
number of guard cells, this may need to be combined with -nxb=... -nyb=... etc. where
the specified blocksize is greater than or equal to 12 (==2*GUARDCELLS). It is equivalent to:

```
SupportWeno=True
```

+supportPPMUpwind

This shortcut specifies a simulation using a specific Hydro method that requires an increased
number of guard cells, this may need to be combined with -nxb=... -nyb=... etc. where
the specified blocksize is greater than or equal to 12 (==2*GUARDCELLS). It is equivalent to:

```
SupportPpmUpwind=True
```

+cube64

This shortcut specifies a simulation with a block size of 64**3, it is equivalent to:

```
-nxb=64 -nyb=64 -nzb=64
```

+cube32

This shortcut specifies a simulation with a block size of 32**3, it is equivalent to:

```
-nxb=32 -nyb=32 -nzb=32
```

+cube16

This shortcut specifies a simulation with a block size of 16**3, it is equivalent to:

```
-nxb=16 -nyb=16 -nzb=16
```

+ptio

This shortcut specifies a simulation using particles and IO for uniform grid, it is equivalent to:

```
+ug -with-unit=Particles
```

+rnf

This shortcut is used for checking FLASH with rectangular block sizes and non-fixed block size.
It is equivalent to:

```
-3d -nxb=8 -nyb=16 -nzb=32 -nofbs +ug
```
5.2. COMPREHENSIVE LIST OF SETUP ARGUMENTS

+nofbs
This shortcut specifies a simulation using a uniform grid with a non-fixed block size. It is equivalent to:
-nofbs +ug parallelIO=True

+curvilinear
This shortcut specifies a simulation using curvilinear geometry. It is equivalent to:
-curved

+cartesian
This shortcut specifies a simulation using cartesian geometry. It is equivalent to:
-geometry=cartesian

+spherical
This shortcut specifies a simulation using spherical geometry. It is equivalent to:
-geometry=spherical

+polar
This shortcut specifies a simulation using polar geometry. It is equivalent to:
-geometry=polar

+cylindrical
This shortcut specifies a simulation using cylindrical geometry. It is equivalent to:
-geometry=cylindrical

+curv-pm2
This shortcut specifies a simulation using curvilinear coordinates along with Paramesh2, it is equivalent to:
+pm2 -unit=Grid/GridMain/paramesh/Paramesh2
-with-unit=Grid/GridMain/paramesh/Paramesh2/monotonic

+spherical-pm2
This shortcut specifies a simulation using spherical coordinates along with Paramesh2, it is equivalent to:
+pm2 +spherical

+ptdens
This shortcut specifies a simulation using passive particles initialized by density. It is equivalent to:
-without-unit=Particles/ParticlesInitialization/Lattice
-without-unit=Particles/ParticlesInitialization/WithDensity/CellMassBins
-unit=Particles/ParticlesMain
-unit=Particles/ParticlesInitialization/WithDensity
-particlemethods=TYPE=passive,INIT=With_Density

+npg
This shortcut specifies a simulation using NO’PERMANENT’GUARDCELLS mode in Paramesh4. It is equivalent to:
npg=True

+mpole
This shortcut specifies a simulation using multipole gravity, it is equivalent to:
-with-unit=physics/Gravity/GravityMain/Poisson/Multipole
+longrange

This shortcut specifies a simulation using long range active particles. It is equivalent to:
-\texttt{with-unit=Particles/ParticlesMain/active/longRange/gravity/ParticleMesh}

+gravPfftNofbs

This shortcut specifies a simulation using FFT based gravity solve on a uniform grid with no fixed block size. It is equivalent to:
+ug +nofbs -\texttt{with-unit=physics/Gravity/GravityMain/Poisson/Pfft}

+gravMgrid

This shortcut specifies a simulation using a multigrid based gravity solve. It is equivalent to:
+pm40 -\texttt{with-unit=physics/Gravity/GravityMain/Poisson/Multigrid}

+gravMpole

This shortcut specifies a simulation using multipole gravity. It is equivalent to:
-\texttt{with-unit=physics/Gravity/GravityMain/Poisson/Multipole}

+noDefaultMpole

This shortcut specifies a simulation *not* using the multipole based gravity solve. It is equivalent to:
-\texttt{without-unit=Grid/GridSolvers/Multipole}

+noMgrid

This shortcut specifies a simulation *not* using the multigrid based gravity solve. It is equivalent to:
-\texttt{without-unit=physics/Gravity/GravityMain/Poisson/Multigrid}

+newMpole

This shortcut specifies a simulation using the new multipole based gravity solve. It is equivalent to:
+noMgrid +noDefaultMpole +gravMpole -\texttt{with-unit=Grid/GridSolvers/Multipole_new}

+mpi1

This shortcut specifies a simulation using the MPI-1 standard. It is equivalent to:
\texttt{mpi1=True -defines=FLASH_MPI1}

+mpi2

This shortcut specifies a simulation using the MPI-2 standard. It is equivalent to:
\texttt{mpi2=True -defines=FLASH_MPI2}

+mtmmmt

This shortcut specifies use of the MultiTemp/MultiType and Tabulated EOSes (for HEDP simulations). It is equivalent to:
-\texttt{unit=physics/Eos/EosMain/multiTemp/Multitype -unit=physics/Eos/EosMain/Tabulated Mtmmmt=1}

+uhd3t

This shortcut specifies a simulation using unsplit hydro with MultiTemp EOS. It is equivalent to:
-\texttt{without-unit=physics/Hydro/HydroMain/split}
-\texttt{without-unit=physics/Hydro/HydroMain/unsplit/Hydro_Unsplit}

+usm3t

This shortcut specifies a simulation using unsplit MHD with MultiTemp EOS. It is equivalent to:
-\texttt{with-unit=physics/Hydro/HydroMain/unsplit/MHD_StaggeredMesh}
-\texttt{without-unit=physics/Hydro/HydroMain/split/MHD_8Wave}
5.2. COMPREHENSIVE LIST OF SETUP ARGUMENTS

+mgd
This shortcut specifies a simulation using the MGD (magneto gas dynamic) radiative transfer module. It is equivalent to:
-\texttt{-unit=physics/materialProperties/Opacity} -\texttt{-unit=physics/RadTrans/RadTransMain/MGD}

+laser
This shortcut specifies use of source terms for energy deposition. It is equivalent to:
-\texttt{-unit=physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser}
-\texttt{-without-unit=Particles}

+pic
This shortcut specifies use of proper particle units to perform PIC (particle in cell) method. It is equivalent to:
+\texttt{ug} -\texttt{-unit=Grid/GridParticles/GridParticlesMove}
-\texttt{-without-unit=Grid/GridParticles/GridParticlesMove/UG}
-\texttt{-without-unit=Grid/GridParticles/GridParticlesMove/UG/Directional}

Grid
This setup variable can be used to specify which gridding package to use in a simulation:
Name: \texttt{Grid}
Type: String
Values: PM4DEV, PM40, UG, PM2, Chombo

IO
This setup variable can be used to specify which IO package to use in a simulation:
Name: \texttt{IO}
Type: String
Values: hdf5, pnetcdf, MPIHybrid, MPIDump, direct

parallelIO
This setup variable can be used to specify which type of IO strategy will be used. A “parallel” strategy will be used if the value is true, a “serial” strategy otherwise.
Name: \texttt{parallelIO}
Type: Boolean
Values: True, False

fixedBlockSize
This setup variable indicates whether or not a fixed block size is to be used. This variable should not be assigned explicitly on the command line. It defaults to True, and the setup options \texttt{-nofbs} and \texttt{-fbs} modify the value of this variable.
Name: \texttt{fixedBlockSize}
Type: Boolean
Values: True, False

nDim
This setup variable gives the dimensionality of a simulation. This variable should not be set explicitly on the command line, it is automatically set by the setup options \texttt{-1d}, \texttt{-2d}, and \texttt{-3d}.
Name: \texttt{nDim}
Type: integer
Values: 1,2,3
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

GridIndexOrder
This setup variable indicates whether the \texttt{-index-reorder} setup option is in effect. This variable should not be assigned explicitly on the command line.
Name: GridIndexOrder
Type: Boolean
Values: True, False

nxb
This setup variable gives the number of zones in a block in the X direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option \texttt{-nxb}.
Name: nxb
Type: integer

nyb
This setup variable gives the number of zones in a block in the Y direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option \texttt{-nyb}.
Name: nyb
Type: integer

nz
This setup variable gives the number of zones in a block in the Z direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option \texttt{-nz}.
Name: nz
Type: integer

maxBlocks
This setup variable gives the maximum number of blocks per processor. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option \texttt{-maxblocks}.
Name: maxBlocks
Type: integer

ParameshLibraryMode
If true, the setup script will generate file \texttt{amr_runtime_parameters} from template \texttt{amr_runtime_parameters.tpl} found in either the object directory (preferred) or the setup script (bin) directory. Selects whether Paramesh4 should be compiled in LIBRARY mode, i.e., with the preprocessor symbol LIBRARY defined.
Name: ParameshLibraryMode
Type: Boolean
Values: True, False

PfftSolver
PfftSolver selects a PFFT solver variant when the hybrid (\textit{i.e.}, Multigrid with PFFT) Poisson solver is used.
Name: PfftSolver
Type: String
Values: DirectSolver (default), HomBcTrigSolver, others (unsupported) if recognized in source/Grid/GridSolvers/Multigrid/PfftTopLevelSolve/Config

Mttmmt
Automatically set True by \texttt{+mttt} shortcut. When true, this option activates the MTMMM MT EOS.
Name: Mttmmt
Type: Boolean
mgd\_meshgroups

`mgd\_meshgroups` * meshCopyCount sets the MAXIMUM number of radiation groups that can be used in a simulation. The ACTUAL number of groups (which must be less than `mgd\_meshgroups * meshCopyCount`) is set by the `rt\_mgdNumGroups` runtime parameter.

Name: `mgd\_meshgroups`
Type: `Integer`

species

This setup variable can be used as an alternative specifying species using the SPECIES Config file directive by listing the species in the setup command. Some units, like the Multispecies Opacity unit, will ONLY work when the species setup variable is set. This is because they use the species name to automatically create runtime parameters which include the species names.

Name: `species`
Type: `String`, comma seperated list of strings (e.g., `species=air,sf6`)

ed\_maxPulses

Name: `ed\_maxPulses`
Type: `integer`
Remark: Maximum number of laser pulses (defaults to 5)

ed\_maxBeams

Name: `ed\_maxBeams`
Type: `integer`
Remark: Maximum number of laser beams (defaults to 6)

threadHydroBlockList

This is used to turn on block list OPENMP threading of hydro routines.

Name: `threadHydroBlockList`
Type: `Boolean`
Values: `True, False`

threadMpoleBlockList

This is used to turn on block list OPENMP threading of the multipole routine.

Name: `threadMpoleBlockList`
Type: `Boolean`
Values: `True, False`

threadRayTrace

This is used to turn on block list OPENMP threading of Energy Deposition source term routines.

Name: `threadRayTrace`
Type: `Boolean`
Values: `True, False`

threadHydroWithinBlock

This is used to turn on within block OPENMP threading of hydro routines.

Name: `threadHydroWithinBlock`
Type: `Boolean`
Values: `True, False`

threadEosWithinBlock

This is used to turn on within block OPENMP threading of Eos routines.

Name: `threadEosWithinBlock`
Type: `Boolean`
Values: `True, False`
threadMpoleWithinBlock

This is used to turn on within block OPENMP threading of then multipole routine.
Name: threadMpoleWithinBlock
Type: Boolean
Values: True, False

5.3 Using Shortcuts

Apart from the various setup options the setup script also allows you to use shortcuts for frequently used combinations of options. For example, instead of typing in

```
./setup -a Sod -with-unit=Grid/GridMain/UG
```

you can just type

```
./setup -a Sod +ug
```

The +ug or any setup option starting with a ‘+’ is considered as a shortcut. By default, setup looks at `bin/setup.shortcuts.txt` for a list of declared shortcuts. You can also specify a “:” delimited list of files in the environment variable `SETUP_SHORTCUTS` and `setup` will read all the files specified (and ignore those which don’t exist) for shortcut declarations. See `Figure 5.2` for an example file.

The shortcuts are replaced by their expansions in place, so options which come after the shortcut override (or conflict with) options implied by the shortcut. A shortcut can also refer to other shortcuts as long as there are no cyclic references.

The “default” shortcut is special. `setup` always prepends `+default` to its command line thus making

```
./setup -a Sod equivalent to ./setup +default -a Sod
```

Thus changing the default IO to “hdf5/parallel”, is as simple as changing the definition of the “default” shortcut.

Some of the more commonly used shortcuts are described below:

5.4 Setup Variables and Preprocessing Config Files

`setup` allows you to assign values to “Setup Variables”. These variables can be string-valued, integer-valued, or boolean. A setup call like

```
./setup -a Sod Foo=Bar Baz=True
```

sets the variable “Foo” to string “Bar” and “Baz” to boolean True. `setup` can conditionally include and exclude parts of the Config file it reads based on the values of these variables. For example, the IO/IOMain/hdf5/Config file contains

The code sets IO to its default value of “serial” and then resets it to “parallel” if the setup variable “parallelIO” is True. The USESETUPVARS keyword in the Config file instructs setup that the specified variables must be defined; undefined variables will be set to the empty string.

Through judicious use of setup variables, the user can ensure that specific implementations are included or the simulation is properly configured. For example, the setup line `./setup -a Sod +ug` expands to

```
./setup -a Sod -unit=Grid/GridMain/ Grid=UG
```

The relevant part of the Grid/GridMain/Config file is given below:

---

3 All non-integral values not equal to True/False/Yes/No/On/Off are considered to be string values.
# comment line
# each line is of the form # shortcut:arg1:arg2:...:
# These shortcuts can refer to each other.

default:--with-library=mpi:-unit=IO/IOMain:-gridinterpolation=monotonic

# io choices
noio:--without-unit=IO/IOMain:
io:--with-unit=IO/IOMain:

# Choice of Grid
ug:-unit=Grid/GridMain/UG:
pm2:-unit=Grid/GridMain/paramesh/Paramesh2:
pm40:-unit=Grid/GridMain/paramesh/paramesh4/Paramesh4.0:
pm4dev:-unit=Grid/GridMain/paramesh/paramesh4/Paramesh4dev:

# frequently used geometries
cube64:-nxb=64:-nyb=64:-nzb=64:

Figure 5.2: A sample `setup_shortcuts.txt` file

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+cartesian</td>
<td>use cartesian geometry</td>
</tr>
<tr>
<td>+cylindrical</td>
<td>use cylindrical geometry</td>
</tr>
<tr>
<td>+noio</td>
<td>omit IO</td>
</tr>
<tr>
<td>+nolog</td>
<td>omit logging</td>
</tr>
<tr>
<td>+pm2</td>
<td>use the PARAMESH2 grid</td>
</tr>
<tr>
<td>+pm40</td>
<td>use the PARAMESH4.0 grid</td>
</tr>
<tr>
<td>+pm4dev</td>
<td>use the PARAMESH4DEV grid</td>
</tr>
<tr>
<td>+polar</td>
<td>use polar geometry</td>
</tr>
<tr>
<td>+spherical</td>
<td>use spherical geometry</td>
</tr>
<tr>
<td>+ug</td>
<td>use the uniform grid in a fixed block size mode</td>
</tr>
<tr>
<td>+nofbs</td>
<td>use the uniform grid in a non-fixed block size mode</td>
</tr>
<tr>
<td>+usm</td>
<td>use the Unsplit Staggered Mesh MHD solver</td>
</tr>
<tr>
<td>+8wave</td>
<td>use the 8-wave MHD solver</td>
</tr>
<tr>
<td>+unsplitHydro</td>
<td>use the Unsplit Hydro solver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+mtmmmt</td>
<td>Use the 3-T, multimaterial, multitype EOS</td>
</tr>
<tr>
<td>+uhd3t</td>
<td>Use the 3-T version of Unsplit Hydro</td>
</tr>
<tr>
<td>+usm3t</td>
<td>Use the 3-T version of Unsplit Staggered Mesh MHD</td>
</tr>
<tr>
<td>+mgd</td>
<td>Use Multigroup Radiation Diffusion and Opacities</td>
</tr>
<tr>
<td>+laser</td>
<td>Use the Laser Ray Trace package</td>
</tr>
</tbody>
</table>
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

```plaintext
DEFAULT serial
USESETUPVARS parallelIO
IF parallelIO
DEFAULT parallel
ENDIF

# Requires use of the Grid SetupVariable
USESETUPVARS Grid

DEFAULT paramesh
IF Grid=='UG'
DEFAULT UG
ENDIF
IF Grid=='PM2'
DEFAULT paramesh/Paramesh2
ENDIF
```

The Grid/GridMain/Config file defaults to choosing Paramesh. But when the setup variable Grid is set to “UG” through the shortcut +ug, the default implementation is set to “UG”. The same technique is used to ensure that the right IO unit is automatically included.

See bin/Readme.SetupVars for an exhaustive list of Setup Variables which are used in the various Config files. For example the setup variable nDim can be test to ensure that a simulation is configured with the appropriate dimensionality (see for example Simulation/SimulationMain/unitTest/Eos/Config).

5.5 Config Files

Information about unit dependencies, default sub-units, runtime parameter definitions, library requirements, and physical variables, etc. is contained in plain text files named Config in the different unit directories. These are parsed by setup when configuring the source tree and are used to create the code needed to register unit variables, to implement the runtime parameters, to choose specific sub-units when only a generic unit has been specified, to prevent mutually exclusive units from being included together, and to flag problems when dependencies are not resolved by some included unit. Some of the Config files contain additional information about unit interrelationships. As mentioned earlier, setup starts from the Config file in the Simulation directory of the problem being built.

5.5.1 Configuration file syntax

Configuration files come in two syntactic flavors: static text and python. In static mode, configuration directives are listed as lines in a plain text file. This mode is the most readable and intuitive of the two, but it lacks flexibility. The python mode has been introduced to circumvent this inflexibility by allowing the configuration file author to specify the configuration directives as a function of the setup variables with a python procedure. This allows the content of each directive and the number of directives in total to be amenable to general programming.

The rule the setup script uses for deciding which flavor of configuration file it’s dealing with is simple. Python configuration files have as their first line #python:genLines. If the first line does not match this
5.5. CONFIG FILES

string, then static mode is assumed and each line of the file is interpreted verbatim as a directive.

If python mode is triggered, then the entire file is considered as valid python source code (as if it were a .py). From this python code, a function of the form `def genLines(setupvars)` is located and executed to generate the configuration directives as an array (or any iterable collection) of strings. The sole argument to genLines is a dictionary that maps setup variable names to their corresponding string values.

As an example, here is a configuration file in python mode that registers runtime parameters named `indexed_parameter_x` where `x` ranges from 1 to NP and NP is a setup line variable.

```python
##python:genLines

# We define genLines as a generator with the very friendly "yield" syntax.
# Alternatively, we could have genLines return an array of strings or even
# one huge multiline string.
def genLines(setupvars):
    # emit some directives that dont depend on any setup variables
    yield ""
    REQUIRES Driver
    REQUIRES physics/Hydro
    REQUIRES physics/Eos
    ""
    # read a setup variable value from the dictionary
    np = int(setupvars("NP")) # must be converted from a string
    # loop from 0 to np-1
    for x in xrange(np):
        yield "PARAMETER indexed_parameter_\%d REAL 0." \% (x+1)
```

When setting up a problem with NP=5 on the setup command line, the following directives will be processed:

```
REQUIRES Driver
REQUIRES physics/Hydro
REQUIRES physics/Eos
PARAMETER indexed_parameter_1 REAL 0.
PARAMETER indexed_parameter_2 REAL 0.
PARAMETER indexed_parameter_3 REAL 0.
PARAMETER indexed_parameter_4 REAL 0.
PARAMETER indexed_parameter_5 REAL 0.
```

5.5.2 Configuration directives

The syntax of the configuration directives is described here. Arbitrarily many spaces and/or tabs may be used, but all keywords must be in uppercase. Lines not matching an admissible pattern will raise an error when running setup.

- **# comment**
  A comment. Can appear as a separate line or at the end of a line.

- **DEFAULT sub-unit**
  Every unit and sub-unit designates one implementation to be the “default”, as defined by the keyword DEFAULT in its Config file. If no specific implementation of the unit or its sub-units is selected by
the application, the designated default implementation gets included. For example, the Config file for the EosMain specifies Gamma as the default. If no specific implementation is explicitly included (i.e., INCLUDE physics/Eos/EosMain/Multigamma), then this command instructs setup to include the Gamma implementation, as though INCLUDE physics/Eos/EosMain/Gamma had been placed in the Units file.

- **EXCLUSIVE implementation...**
  Specifies a list of implementations that cannot be included together. If no EXCLUSIVE instruction is given, it is perfectly legal to simultaneously include more than one implementation in the code. Using “EXCLUSIVE” means that at most one implementation can be included.

- **CONFLICTS unit1[/sub-unit[/implementation...]] ...**
  Specifies that the current unit, sub-unit, or specific implementation is not compatible with the list of units, sub-units or other implementations that follows. setup issues an error if the user attempts a conflicting unit configuration.

- **REQUIRES unit[/sub-unit[/implementation...]] [ OR unit[/sub-unit...]]...**
  Specifies a unit requirement. Unit requirements can be general, without asking for a specific implementation, so that unit dependencies are not tied to particular algorithms. For example, the statement REQUIRES physics/Eos in a unit’s Config file indicates to setup that the physics/Eos unit is needed, but no particular equation of state is specified. As long as an Eos implementation is included, the dependency will be satisfied. More specific dependencies can be indicated by explicitly asking for an implementation. For example, if there are multiple species in a simulation, the Multigamma equation of state is the only valid option. To ask for it explicitly, use REQUIRES physics/Eos/EosMain/Multigamma. Giving a complete set of unit requirements is helpful, because setup uses them to generate the units file when invoked with the -auto option.

- **REQUESTS unit[/sub-unit[/implementation...]]**
  Requests that a unit be added to the Simulation. All requests are upgraded to a “REQUIRES” if they are not negated by a ”-without-unit” option from the command line. If negated, the REQUEST is ignored. This can be used to turn off profilers and other “optional” units which are included by default.

- **SUGGEST unitname unitname ...**
  Unlike REQUIRES, this keyword suggests that the current unit be used along with one of the specified units. The setup script will print details of the suggestions which have been ignored. This is useful in catching inadvertently omitted units before the run starts, thus avoiding a waste of computing resources.

- **PARAMETER name type [CONSTANT] default [range-spec]**
  Specifies a runtime parameter. Parameter names are unique up to 20 characters and may not contain spaces. Admissible types include REAL, INTEGER, STRING, and BOOLEAN. Default values for REAL and INTEGER parameters must be valid numbers, or the compilation will fail. Default STRING values must be enclosed in double quotes ("). Default BOOLEAN values must be .true. or .false. to avoid compilation errors. Once defined, runtime parameters are available to the entire code. Optionally, any parameter may be specified with the CONSTANT attribute (e.g., PARAMETER foo REAL CONSTANT 2.2). If a user attempts to set a constant parameter via the runtime parameter file, an error will occur.

The range specification is optional and can be used to specify valid ranges for the parameters. The range specification is allowed only for REAL, INTEGER, STRING variables and must be enclosed in ['.]. For a STRING variable, the range specification is a comma-separated list of strings (enclosed in quotes). For a INTEGER, REAL variable, the range specification is a comma-separated list of (closed) intervals specified by min ... max, where min and max are the end points of the interval. If min or max is omitted, it is assumed to be $-\infty$ and $+\infty$ respectively. Finally val is a shortcut for val ... val. For example

PARAMETER pres REAL 1.0 [ 0.1 ... 9.9, 25.0 ... ]
PARAMETER coords STRING "polar" ["polar","cylindrical","2d","3d"]
indicates that \texttt{pres} is a \texttt{REAL} variable which is allowed to take values between 0.1 and 9.9 or above 25.0. Similarly \texttt{coords} is a \texttt{string} variable which can take one of the four specified values.

- \texttt{D} parameter-name \texttt{comment}

  Any line in a \texttt{Config} file is considered a parameter comment line if it begins with the token \texttt{D}. The first token after the comment line is taken to be the parameter name. The remaining tokens are taken to be a description of the parameter’s purpose. A token is delineated by one or more white spaces. For example,

  \texttt{D SOME\_PARAMETER The purpose of this parameter is whatever}

  If the parameter comment requires additional lines, the \& is used to indicate continuation lines. For example,

  \texttt{D SOME\_PARAMETER The purpose of this parameter is whatever}
  \texttt{D & This is a second line of description}

  You can also use this to describe other variables, fluxes, species, etc. For example, to describe a species called "xyz", create a comment for the parameter “xyz\_species”. In general the name should be followed by an underscore and then by the lower case name of the keyword used to define the name.

  Parameter comment lines are special because they are used by \texttt{setup} to build a formatted list of commented runtime parameters for a particular problem. This information is generated in the file \texttt{setup.params} in the \texttt{object} directory.

- \texttt{VARIABLE name /TYPE: vartype} \{eosmap-spec\}

  Registers variable with the framework with name \texttt{name} and a variable type defined by \texttt{vartype}. The \texttt{setup} script collects variables from all the included units, and creates a comprehensive list with no duplications. It then assigns defined constants to each variable and calculates the amount of storage required in the data structures for storing these variables. The defined constants and the calculated sizes are written to the file \texttt{Flash.h}.

  The possible types for \texttt{vartype} are as follows:

  - \texttt{PER\_VOLUME}
    
    This solution variable is represented in \texttt{conserved} form, \textit{i.e.}, it represents the density of a conserved extensive quantity. The prime example is a variable directly representing mass density. Energy densities, momentum densities, and partial mass densities would be other examples (but these quantities are usually represented in \texttt{PER\_MASS} form instead).

  - \texttt{PER\_MASS}
    
    This solution variable is represented in \texttt{mass-specific} form, \textit{i.e.}, it represents quantities whose nature is extensive quantity per mass unit. Examples are specific energies, velocities of material (since they are equal to momentum per mass unit), and abundances or mass fractions (partial density divided by density).

  - \texttt{GENERIC}
    
    This is the default \texttt{vartype} and need not be specified. This type should be used for any variables that do not clearly belong to one of the previous two categories.

In the current version of the code, the \texttt{TYPE} attribute is only used to determine which variables should be converted to conservative form for certain \texttt{Grid} operations that may require interpolation (\textit{i.e.}, prolongation, guardcell filling, and restriction) when one of the runtime parameters \texttt{convertToConsdvForMeshCalls} or \texttt{convertToConsdvInMeshInterp} is set \texttt{true}. Only variables of type \texttt{PER\_MASS} are converted: values are multiplied cell-by-cell with the value of the "\texttt{dens}" variable, and potential interpolation results are converted back by cell-by-cell division by "\texttt{dens}" values after interpolation.

Note that therefore

- variables types are irrelevant for uniform grids,
variables types are irrelevant if neither `convertToConsvdForMeshCalls` nor `convertToConsvdInMeshInterp` is true, and

variable types (and conversion to and from conserved form) only take effect if a

VARIABLE dens ...

exists.

An `eosmap-spec` has the syntax `EOSMAP: eos-role | ( [EOSMAPIN: eos-role] [EOSMAPOUT: eos-role ] )`, where `eos-role` stands for a role as defined in `Eos_map.h`. These roles are used within implementations of the `Eos_wrapped` interface, via the subroutines `Eos_getData` and `Eos_putData`, to map variables from Grid data structures to the `eosData` array that `Eos` understands, and back. For example,

VARIABLE eint TYPE: PER_MASS EOSMAPIN: EINT

means that within `Eos_wrapped`, the `EINT_VAR` component of `unk` will be treated as the grid variable in the “internal energy” role for the purpose of constructing input to `Eos`, and

VARIABLE gamc EOSMAPOUT: GAMC

means that within `Eos_wrapped`, the `GAMC_VAR` component of `unk` will be treated as the grid variable in the `EOSMAP_GAMC` role for the purpose of returning results from calling `Eos` to the grid. The specification

VARIABLE pres EOSMAP: PRES

has the same effect as

VARIABLE pres EOSMAPIN: PRES EOSMAPOUT: PRES

Note that not all roles defined in `Eos_map.h` are necessarily meaningful or actually used in a given `Eos` implementation. An `eosmap-spec` for a VARIABLE is only used in an `Eos_wrapped` invocation when the optional `gridDataStruct` argument is absent or has a value of CENTER.

• FACEVAR name [eosmap-spec]
  This keyword has the same meaning for face-centered variables, that VARIABLE does for cell-centered variables. It allocates space in the grid data structure that contains face-centered physical variables for “name”. See Section 6.1 for more information

  For eosmap-spec, see above under VARIABLE. An eosmap-spec for FACEVAR is only used when `Eos_wrapped` is called with an optional `gridDataStruct` argument of `FACEX`, `FACEY`, or `FACEZ`.

• FLUX name
  Registers flux variable `name` with the framework. When using an adaptive mesh, flux conservation is needed at fine-coarse boundaries. `PARAMESH` uses a data structure for this purpose, the flux variables provide indices into that data structure. See Section 6.3 for more information.

• SCRATCHCENTERVAR name [eosmap-spec]
  This keyword is used in connection with the grid scope scratch space for cell-centered data supported by FLASH. It allows the user to ask for scratch space with “name”. The scratch variables do not participate in the process of guardcell filling, and their values become invalid after a grid refinement step. While users can define scratch variables to be written to the plotfiles, they are not by default written to checkpoint files. Note this feature wasn’t available in FLASH2. See Section 6.4 for more information.

• SCRATCHFACEVAR name [eosmap-spec]
  This keyword is used in connection with the grid scope scratch space for face-centered data, it is identical in every other respect to SCRATCHCENTERVAR.
- **SCRATCHVAR** name [eosmap-spec]
  This keyword is used for specifying instances of general purpose grid scope scratch space. The same space can support cell-centered as well as face-centered data. Like other scratch data structures, the variables in this data structure can also be asked with “name” and do not participate in guardcell filling.

  For eosmap-spec, see above under VARIABLE. An eosmap-spec for SCRATCHVAR is only used when Eos_wrapped is called with an optional gridDataStruct argument of SCRATCH.

- **MASS_SCALAR** name [RENORM: group-name] [eosmap-spec]
  If a quantity is defined with keyword MASS_SCALAR, space is created for it in the grid “unk” data structure. It is treated like any other variable by PARAMESH, but the hydrodynamic unit treats it differently. It is advected, but other physical characteristics don’t apply to it. If the optional “RENORM” is given, this mass-scalar will be added to the renormalization group of the accompanying group name. The hydrodynamic solver will renormalize all mass-scalars in a given group, ensuring that all variables in that group will sum to 1 within an individual cell. See Section 6.2

  For eosmap-spec, see above under VARIABLE. An eosmap-spec for a MASS_SCALAR may be used in an Eos_wrapped invocation when the optional gridDataStruct argument is absent or has a value of CENTER.

### Avoid Confusion!

It is inadvisable to name variables, species, and mass scalars with the same prefix, as post-processing routines have difficulty deciphering the type of data from the output files. For example, don’t create a variable “temp” to hold temperature and a mass scalar “temp” indicating a temporary variable. Although the Flash.h file can distinguish between these two types of variables, many plotting routines cannot.

- **PARTICLETYPE** particle-type INITMETHOD initialization-method MAPMETHOD map-method ADVMETHOD time-advance-method
  This keyword associates a particle type with mapping and initialization sub-units of Particles unit to operate on this particle type during the simulation. Here, map-method describes the method used to map the particle properties to and from the mesh (see Section 19.2), initialization-method describes the method used to distribute the particles at initialization, and time-advance-method describes the method used to advance the associated particle type in time (see Section 19.1, and in general Section 19.3). This keyword has been introduced to facilitate inclusion of multiple particle types in the same simulation. It imposes certain requirements on the use of the ParticlesMapping and ParticlesInitialization subunits. Particles (of any type, whether called passive or anything else) do not have default methods for initialization, mapping, or time integration, so a PARTICLETYPE directive in a Config file (or an equivalent -particlemethods= setup option, see Table 5.4) is the only way to specify the appropriate implementations of the Particles subunits to be used. The declaration should be accompanied by appropriate “REQUESTS” or “REQUIRES” directives to specify the paths of the appropriate subunit implementation directories to be included. For clarity, our technique has been to include this information in the simulation directory Config files only. All the currently available mapping and initialization methods have a corresponding identifier in the form of preprocessor definition in Particles.h. The user may select any particle-type name, but the map-method, initialization-method and time-advance-method must correspond to existing identifiers defined in Particles.h. This is necessary to navigate the data structure that stores the particle type and its associated mapping and initialization methods. Users desirous of adding new methods for mapping or initialization should also update the Particles.h file with additional identifiers and their preprocessor definitions. Note, it is possible to use the same methods for different particle types, but each particle type name must only appear once. Finally, the Simulations Config file is also expected to request appropriate implementations of mapping and initialization subunits using the keyword REQUESTS, since the corresponding Config files do not specify
a default implementation to include. For example, to include passive particle types with Quadratic mapping, Lattice initialization, and Euler for advancing in time the following code segment should appear in the Config file of the Simulations directory.

```
PARTICLETYPE passive INITMETHOD lattice MAPMETHOD quadratic ADVMETHOD Euler
REQUIRES Particles/ParticlesMain
REQUESTS Particles/ParticlesMain/passive/Euler
REQUESTS Particles/ParticlesMapping/Quadratic
REQUESTS Particles/ParticlesInitialization/Lattice
```

- **PARTICLEPROP name type**
  This keyword indicates that the particles data structure will allocate space for a sub-variable “NAME_-PART_PROP.” For example if the Config file contains

```
PARTICLEPROP dens
```

then the code can directly access this property as

```
particles(DENS_PART_PROP,1:localNumParticles) = densInitial
```

type may be REAL or INT, however INT is presently unused. See Section 6.6 for more information and examples.

- **PARTICLEMAP TO partname FROM vartype varname**
  This keyword maps the value of the particle property partname to the variable varname. vartype can take the values VARIABLE, MASS, SCALAR, SPECIES, FACEX, FACEY, FACEZ, or one of SCRATCH types (SCRATCHVAR/SCRATCHCENTERVAR, SCRATCHFACEXVAR, SCRATCHFACEYVAR, SCRATCHFACEZVAR). These maps are used to generate Simulation_mapParticlesVar, which takes the particle property partname and returns varname and vartype. For example, to have a particle property tracing density:

```
PARTICLEPROP dens REAL
PARTICLEMAP TO dens FROM VARIABLE dens
```

or, in a more advanced case, particle properties tracing some face-valued function Mag:

```
PARTICLEPROP Mag_x REAL
PARTICLEPROP Mag_y REAL
PARTICLEPROP Mag_z REAL
PARTICLEMAP TO Mag_x FROM FACEX Mag
PARTICLEMAP TO Mag_y FROM FACEY Mag
PARTICLEMAP TO Mag_z FROM FACEZ Mag
```

Additional information on creating Config files for particles is obtained in Section 19.3.2.

- **SPECIES name [TO number of ions]**
  An application that uses multiple species uses this keyword to define them. See Section 6.2 for more information. The user may also specify an optional number of ions for each element, name. For example, SPECIES o TO 8 creates 9 spaces in unk for Oxygen, that is, a single space for Oxygen and 8 spaces for each of its ions. This is relevant to simulations using the ionize unit. (Omitting the optional TO specifier is equivalent to specifying TO 0).

- **DATAFILES wildcard**
  Declares that all files matching the given wildcard in the unit directory should be copied over to the object directory. For example,

```
DATAFILES *.dat
```

will copy all the “.dat” files to the object directory.
5.5. CONFIG FILES

• KERNEL [subdir]
  Declares that all subdirectories must be recursively included. This usually marks the end of the high level architecture of a unit. Directories below it may be third party software or a highly optimized solver, and are therefore not required to conform to FLASH architecture.

  Without a subdir, the current directory (i.e., the one containing the Config file with the KERNEL keyword) is marked as a kernel directory, so code from all its subdirectories (with the exception of subdirectories whose name begins with a dot) is included. When a subdir is given, then that subdirectory must exist, and it is treated as a kernel directory in the same way.

  Note that currently the setup script can process only one KERNEL directive per Config file.

• LIBRARY name
  Specifies a library requirement. Different FLASH units require different libraries, and they must inform setup so it can link the libraries into the executable. Some valid library names are HDF5, MPI. Support for external libraries can be added by modifying the site-specific Makefile.h files to include appropriate Makefile macros. It is possible to use internal libraries, as well as switch libraries at setup time. To use these features, see Library-HOWTO.

• LINKIF filename unitname
  Specifies that the file filename should be used only when the unit unitname is included. This keyword allows a unit to have multiple implementations of any part of its functionality, even down to the kernel level, without the necessity of creating children for every alternative. This is especially useful in Simulation setups where users may want to use different implementations of specific functions based upon the units included. For instance, a user may wish to supply his/her own implementation of Grid_markRefineDerefine.F90, instead of using the default one provided by FLASH. However, this function is aware of the internal workings of Grid, and has different implementations for different grid packages. The user could therefore specify different versions of his/her own file that are intended for use with the different grids. For example, adding

  LINKIF Grid_markRefineDerefine.F90.ug Grid/GridMain/UG
  LINKIF Grid_markRefineDerefine.F90.pmesh Grid/GridMain/paramesh

  to the Config file ensures that if the application is built with UG, the file Grid_markRefineDerefine.F90.ug will be linked in as Grid_markRefineDerefine.F90, whereas if it is built with Paramesh2 or Paramesh4.0 or Paramesh4dev, then the file Grid_markRefineDerefine-.F90.pmesh will be linked in as Grid_markRefineDerefine.F90. Alternatively, the user may want to provide only one implementation specific to, say, PARAMESH. In this case, adding

  LINKIF Grid_markRefineDerefine.F90 Grid/GridMain/paramesh

  to the Config file ensures that the user-supplied file is included when using PARAMESH (either version), while the default FLASH file is included when using UG.

• PPDEFINE sym1 sym2 ...
  Instructs setup to add the PreProcessor symbols SYM1 and SYM2 to the generated Flash.h. Here SYM1 is sym1 converted to uppercase. These pre-process symbols can be used in the code to distinguish between which units have been used in an application. For example, a Fortran subroutine could include

  #ifdef FLASH_GRID_UG
    ug specific code
  #endif

  #ifdef FLASH_GRID_PARAMESH3OR4
    pm3+ specific code
  #endif
By convention, many preprocessor symbols defined in Config files included in the FLASH code distribution start with the prefix “FLASH_”.

- **USESETUPVARS** var1, var2, ...
  This tells `setup` that the specified “Setup Variables” are being used in this Config file. The variables initialize to an empty string if no values are specified for them. Note that commas are required if listing several variables.

- **CHILDORDER** child1 child2 ...
  When `setup` links several implementations of the same function, it ensures that implementations of children override that of the parent. Its method is to lexicographically sort all the names and allow implementations occurring later to override those occurring earlier. This means that if two siblings implement the same code, the names of the siblings determine which implementation wins. Although it is very rare for two siblings to implement the same function, it does occur. This keyword permits the Config file to override the lexicographic order by one preferred by the user. Lexicographic ordering will prevail as usual when deciding among implementations that are not explicitly listed.

- **GUARDCELLS** num
  Allows an application to choose the stencil size for updating grid points. The stencil determines the number of guardcells needed. The PPM algorithm requires 4 guardcells, hence that is the default value. If an application specifies a smaller value, it will probably not be able to use the default monotonic AMR Grid interpolation; see the `-gridinterpolation setup` flag for additional information.

- **SETUPERROR** error message
  This causes `setup` to abort with the specified error message. This is usually used only inside a conditional IF/ENDIF block (see below).

- **IF, ELSEIF, ELSE, ENDIF**
  A conditional block is of the following form:

  ```
  IF cond
    ...
  ELSEIF cond
    ...
  ELSE
    ...
  ENDIF
  ```

  where the ELSEIF and ELSE blocks are optional. There is no limit on the number of ELSEIF blocks. “...” is any sequence of valid Config file syntax. The conditional blocks may be nested. “cond” is any boolean valued Python expression using the setup variables specified in the USESETUPVARS.

- **NONREP unktype name localmax globalparam ioformat**
  Declares an array of UNK variables that will be partitioned across the replicated meshes. Using various preprocessor macros in Flash.h each copy of the mesh can determine at runtime its own subset of indexes into this global array. This allows an easy form of parallelism where regular "replicated" mesh variables are computed redundantly across processors, but the variables in the "non-replicated" array are computed in parallel.

  - **unktype**: must be either MASS_SCALAR or VARIABLE
  - **name**: the name of this variable array. It is suggested that it be all capital letters, and must conform to what the C preprocessor will consider as a valid symbol for use in a `#define` statement.
  - **localmax**: a positive integer specifying the maximum number of elements from the global variable array a mesh can hold. This is the actual number of UNK variables that are allocated on each processor, though not all of them will necessarily be used.
– *globalparam*: the name of a runtime parameter which dictates the size of this global array of variables.

– *ioformat*: a string representing how the elements of the array will be named when written to the output files. The question mark character ? is used as a placeholder for the digits of the array index. As an example, the format string x?? will generate the dataset names x001, x002, x003, etc. This string must be no more than four characters in length.

The number of meshes is dictated by the runtime parameter *meshCopyCount*. The following constraint must be satisfied or FLASH will fail at runtime:

\[ \text{globalparam} \leq \text{meshCopyCount} \times \text{localmax} \]

The reason for this restriction is that *localmax* is the maximum number of array elements a mesh can be responsible for, and *meshCopyCount* is the number of meshes, so their product bounds the size of the array.

Example:

Config file:

```plaintext
NONREP MASS_SCALAR A 4 numA a???
NONREP MASS_SCALAR B 5 numB b???
```

flash.par file:

```plaintext
meshCopyCount = 3
numA = 11
numB = 15
```

In this case two non-replicated mass-scalar arrays are defined, A and B. Their lengths are specified by the runtime parameters *numA* and *numB* respectively. *numB* is set to its maximum value of 5 * *meshCopyCount* = 15, but *numA* is one less than its maximum value of 4 * *meshCopyCount* = 12 so at runtime one of the meshes will not have all of its UNK variables in use. The dataset names generated by IO will take the form a001 ...a011 and b001 ...b015.

The preprocessor macros defined in *Flash.h* for these arrays will have the prefixes *A_* and *B_* respectively. For details about these macros and how they will distribute the array elements across the meshes see Section 6.7.

### 5.6 Creating a Site-specific Makefile

If *setup* does not find your hostname in the *sites/* directory it picks a default *Makefile* based on the operating system. This *Makefile* is not always correct but can be used as a template to create a *Makefile* for your machine. To create a Makefile specific to your system follow these instructions.

- Create the directory *sites/<hostname>*, where *<hostname>* is the hostname of your machine.
- Start by copying *os/<your os>/Makefile.h* to *sites/<hostname>*.
• Use `bin/suggestMakefile.sh` to help identify the locations of various libraries on your system. The script scans your system and displays the locations of some libraries. You must note the location of MPI library as well. If your compiler is actually an mpi-wrapper (`e.g. mpif90`), you must still define `LIB_MPI` in your site specific `Makefile.h` as the empty string.

• Edit `sites/<hostname>/Makefile.h` to provide the locations of various libraries on your system.

• Edit `sites/<hostname>/Makefile.h` to specify the FORTRAN and C compilers to be used.

---

**Actual Compiler or MPI wrapper?**

If you have MPI installed, you can either specify the actual compiler (`e.g.f90`) or the mpi-wraper (`e.g.mpif90`) for the “compiler” to be used on your system. Specifying the actual compiler and the location of the MPI libraries in the site-specific Makefile allows you the possibility of switching your MPI implementation. For more information see Library-HOWTO.

---

**Compilation warning**

The `Makefile.h` must include a compiler flag to promote Fortran Reals to Double Precision. FLASH performs all MPI communication of Fortran Reals using `MPI_DOUBLE_PRECISION` type, and assumes that Fortran Reals are interoperable with C doubles in the I/O unit.

---

### 5.7 Files Created During the setup Process

When `setup` is run it generates many files in the `object` directory. They fall into three major categories:

(a) Files not required to build the FLASH executable, but which contain useful information,

(b) Generated F90 or C code, and

(c) Makefiles required to compile the FLASH executable.

#### 5.7.1 Informational files

These files are generated before compilation by `setup`. Each of these files begins with the prefix `setup_` for easy identification.
5.7. FILES CREATED DURING THE SETUP PROCESS

| setup_call | contains the options with which setup was called and the command line resulting after shortcut expansion |
| setup_libraries | contains the list of libraries and their arguments (if any) which was linked in to generate the executable |
| setup_units | contains the list of all units which were included in the current setup |
| setup_defines | contains a list of all pre-process symbols passed to the compiler invocation directly |
| setup_flags | contains the exact compiler and linker flags |
| setup_params | contains the list of runtime parameters defined in the Config files processed by setup |
| setup_vars | contains the list of variables, fluxes, species, particle properties, and mass scalars used in the current setup, together with their descriptions. |

5.7.2 Code generated by the setup call

These routines are generated by the setup call and provide simulation-specific code.

| setup_buildstamp.F90 | contains code for the subroutine setup_buildstamp which returns the setup and build time as well as code for setup_systemInfo which returns the uname of the system used to setup the problem |
| setup_buildstats.c | contains code which returns build statistics including the actual setup call as well as the compiler flags used for the build |
| setup_getFlashUnits.F90 | contains code to retrieve the number and list of flashUnits used to compile code |
| setup_flashRelease.F90 | contains code to retrieve the version of FLASH used for the build |
| Flash.h | contains simulation specific preprocessor macros, which change based upon setup unlike constants.h. It is described in Chapter 6 |
| Simulation_mapIntToStr.F90 | contains code to map an index described in Flash.h to a string described in the Config file. |
| Simulation_mapStrToInt.F90 | contains code to map a string described in the Config file to an integer index described in the Flash.h file. |
| Simulation_mapParticlesVar.F90 | contains a mapping between particle properties and grid variables. Only generated when particles are included in a simulation. |
| Particles_specifyMethods.F90 | contains code to make a data structure with information about the mapping and initialization method for each type of particle. Only generated when particles are included in a simulation. |

5.7.3 Makefiles generated by setup

Apart from the master Makefile, setup generates a makefile for each unit, which is “included” in the master Makefile. This is true even if the unit is not included in the application. These unit makefiles are named Makefile.Unit and are a concatenation of all the Makefiles found in unit hierarchy processed by setup.
For example, if an application uses `Grid/GridMain/paramesh/paramesh4/Paramesh4.0`, the file `Makefile.Grid` will be a concatenation of the Makefiles found in

- `Grid`
- `Grid/GridMain`
- `Grid/GridMain/paramesh`
- `Grid/GridMain/paramesh/paramesh4`, and
- `Grid/GridMain/paramesh/paramesh4/Paramesh4.0`

As another example, if an application does not use `PhysicalConstants`, then `Makefile.Physical-Constants` is just the contents of `PhysicalConstants/Makefile` at the API level.

Since the order of concatenation is arbitrary, the behavior of the Makefiles should not depend on the order in which they have been concatenated. The makefiles inside the units contain lines of the form:

```
Unit += file1.o file2.o ...
```

where `Unit` is the name of the unit, which was `Grid` in the example above. Dependency on data modules files need not be specified since the setup process determines this requirement automatically.

## 5.8 Setup a hybrid MPI+OpenMP FLASH application

There is the experimental inclusion of FLASH multithreading with OpenMP in the FLASH4 beta release. The units which have support for multithreading are split hydrodynamics 14.1.2, unsplit hydrodynamics 14.1.3, Gamma law and multigamma EOS 15.2, Helmholtz EOS 15.3, Multipole Poisson solver (improved version (support for 2D cylindrical and 3D cartesian)) 8.10.2.2 and energy deposition 16.4.

The FLASH multithreading requires a MPI-2 installation built with thread support (building with an MPI-1 installation or an MPI-2 installation without thread support is possible but strongly discouraged). The FLASH application requests the thread support level `MPI_THREAD_SERIALIZED` to ensure that the MPI library is thread-safe and that any OpenMP thread can call MPI functions safely. You should also make sure that your compiler provides a version of OpenMP which is compliant with at least the OpenMP 2.5 (200505) standard (older versions may also work but I have not checked).

In order to make use of the multithreaded code you must setup your application with one of the setup variables `threadBlockList`, `threadWithinBlock` or `threadRayTrace` equal to `True`, e.g.

```
./setup Sedov -auto threadBlockList=True
./setup Sedov -auto threadBlockList=True +mpi1 (compatible with MPI-1 - unsafe!)
```

When you do this the setup script will insert `USEOPENMP = 1` instead of `USEOPENMP = 0` in the generated `Makefile`. If it is equal to 1 the `Makefile` will prepend an OpenMP variable to the `FFLAGS`, `CFLAGS`, `LFLAGS` variables.

### Makefile.h variables

In general you should not define `FLAGS`, `CFLAGS` and `LFLAGS` in your `Makefile.h`. It is much better to define `FFLAGS_OPT`, `FFLAGS_TEST`, `FFLAGS_DEBUG`, `CFLAGS_OPT`, `CFLAGS_TEST`, `CFLAGS_DEBUG`, `LFLAGS_OPT`, `LFLAGS_TEST` and `LFLAGS_DEBUG` in your `Makefile.h`. The setup script will then initialize the `FFLAGS`, `CFLAGS` and `LFLAGS` variables in the `Makefile` appropriately for an optimized, test or debug build.

The OpenMP variables should be defined in your `Makefile.h` and contain a compiler flag to recognize OpenMP directives. In most cases it is sufficient to define a single variable named `OPENMP`, but you may encounter special situations when you need to define `OPENMP_FORTRAN`, `OPENMP_C` and `OPENMP_LINK`. If you want to build FLASH with the GNU Fortran compiler `gfortran` and the GNU C compiler `gcc` then your `Makefile.h` should contain
5.8. SETUP A HYBRID MPI+OPENMP FLASH APPLICATION

OPENMP = -fopenmp

If you want to do something more complicated like build FLASH with the Lahey Fortran compiler lf90 and the GNU C compiler gcc then your Makefile.h should contain

OPENMP_FORTRAN = --openmp --kpureomp
OPENMP_C = -fopenmp
OPENMP_LINK = --openmp --kpureomp

When you run the hybrid FLASH application it will print the level of thread support provided by the MPI library and the number of OpenMP threads in each parallel region

[Driver_initParallel]: Called MPI_Init_thread - requested level 2, given level 2
[Driver_initParallel]: Number of OpenMP threads in each parallel region 4

Note that the FLASH application will still run if the MPI library does not provide the requested level of thread support, but will print a warning message alerting you to an unsafe level of MPI thread support. There is no guarantee that the program will work! I strongly recommend that you stop using this FLASH application - you should build a MPI-2 library with thread support and then rebuild FLASH.

We record extra version and runtime information in the FLASH log file for a threaded application. Table 5.7 shows log file entries from a threaded FLASH application along with example safe and unsafe values. All cells colored red show unsafe values.

<table>
<thead>
<tr>
<th>Log file stamp</th>
<th>safe</th>
<th>unsafe (1)</th>
<th>unsafe (2)</th>
<th>unsafe (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of MPI tasks:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MPI version:</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MPI subversion:</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MPI thread support:</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>OpenMP threads/MPI task:</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>OpenMP version:</td>
<td>200805</td>
<td>200505</td>
<td>200505</td>
<td>200805</td>
</tr>
<tr>
<td>Is &quot;_OPENMP&quot; macro defined:</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 5.7: Log file entries showing safe and unsafe threaded FLASH applications

The FLASH applications in Table 5.7 are unsafe because

1. we are using an MPI-1 implementation.
2. we are using an MPI-2 implementation which is not built with thread support - the “MPI thread support in OpenMPI” Flash tip may help.
3. we are using a compiler that does not define the macro _OPENMP when it compiles source files with OpenMP support (see OpenMP standard). I have noticed that Absoft 64-bit Pro Fortran 11.1.3 for Linux x86_64 does not define this macro. We use this macro in Driver_initParallel.F90 to conditionally initialize MPI with MPI_Init_thread. If you find that _OPENMP is not defined you should define it in your Makefile.h in a manner similar to the following:

OPENMP_FORTRAN = -openmp -D_OPENMP=200805

### MPI thread support in OpenMPI

A default installation of OpenMPI-1.5 (and earlier) does not provide any level of MPI thread support. To include MPI thread support you must configure OpenMPI-1.5 with --enable-mpi-thread-multiple or --enable-opal-multi-threads. We prefer to configure with --enable-mpi-thread-multiple so that we can (in future) use the highest level of thread support. The configure option is named --enable-mpi-threads in earlier versions of OpenMPI.
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

MPI-IO issues when using a threaded FLASH application

The ROMIO in older versions of MPICH2 and OpenMPI is known to be buggy. We have encountered a segmentation fault on one platform and a deadlock on another platform during MPI-IO when we used OpenMPI-1.4.4 with a multithreaded FLASH application. We solved the error by using OpenMPI-1.5.4 (it should be possible to use OpenMPI-1.5.2 or greater because the release notes for OpenMPI-1.5.2 state “Updated ROMIO from MPICH v1.3.1 (plus one additional patch)”). We have not tested to find the minimum version of MPICH2 but MPICH2-1.4.1p1 works fine. If it is not possible to use a newer MPI implementation you can avoid MPI-IO altogether by setting up your FLASH application with +serialIO.

You should not setup a FLASH application with both threadBlockList and threadWithinBlock equal to True - nested OpenMP parallelism is not supported. For further information about FLASH multithreaded applications please refer to Chapter 33.

5.9 Setup a FLASH+Chombo application

5.9.1 Overview

In FLASH4 we have introduced a new grid implementation that makes use of Chombo library (Section 8.7). This allows us to create FLASH applications that use an adaptive patch-based mesh for the first time.

You can create a FLASH application using Chombo mesh with the standard FLASH setup script and the designated shortcuts +chombo_ug or +chombo_amr. These shortcuts instruct the setup script to:

- Add the appropriate FLASH C++ source files that interact with Chombo library to the object directory. There are different C++ wrapper classes for a uniform grid (+chombo_ug) and adaptive grid (+chombo_amr) configuration.
- Reorder the arrays so that FLASH blocks normally accessed with solnData(v,i,j,k) are now accessed with solnData(i,j,k,v). This is a pre-processing step that is described in Section 5.2.
- Undefine the FIXEDBLOCKSIZE macro in Flash.h so that the code in FLASH source files does not assume that blocks contain a fixed number of cells.
- Define a setup variable named chomboCompatibleHydro. This variable is used in Hydro Config files to include a version of Hydro that is compatible with Chombo flux correction. At the current time we have only implemented flux correction with Chombo library for Split Hydro.
- Use the Makefile header file named Makefile.h.chombo instead of Makefile.h.

The shortcuts are used in FLASH setup lines as follows:

./setup Sedov -auto +chombo_ug -parfile=test_chombo_ug_2d.par
./setup Sod -auto +chombo_amr -parfile=test_chombo_amr_2d.par

The setup lines for all the problems that we have ever tested can be found in the file sites/code.uchicago.edu/flash_test/test.info.

5.9.2 Build procedure

A FLASH application that makes use of Chombo library has the following dependencies:

- C++ compiler
- Fortran compiler with C interoperability features
We expect to pick up the complete set of integer values from 0 to 23 in contiguous memory locations.

Fortran array element: 1 1 1 1. Expected: 0, actual: 0.
Fortran array element: 1 2 1 1. Expected: 1, actual: 1.
Fortran array element: 1 1 2 1. Expected: 2, actual: 2.
Fortran array element: 1 2 2 1. Expected: 3, actual: 3.
Fortran array element: 1 2 3 1. Expected: 5, actual: 5.
Fortran array element: 1 1 1 2. Expected: 6, actual: 3.
Fortran array element: 1 1 2 2. Expected: 8, actual: 5.
Fortran array element: 1 2 2 2. Expected: 9, actual: 6.
Fortran array element: 1 1 3 2. Expected: 10, actual: 7.
Fortran array element: 1 2 3 2. Expected: 11, actual: 8.
Fortran array element: 1 1 1 4. Expected: 18, actual: 9.

FAILURE! Picked up unexpected C++ assigned values.

Figure 5.3: Unit test stdout for gfortran versions affected by GCC bug 40962.

- MPI build of Chombo
- parallel HDF5

The mixed-language Fortran/C interoperability features are critical for this project. Not all Fortran compilers have working C interoperability as this is a new feature that was introduced in the Fortran 2003 standard. Before spending time building Chombo and customizing FLASH Makefile headers, you should use our standalone unit test to determine whether your compilers have the basic mixed-language functionality needed by this project. The unit test is at source/Grid/GridMain/Chombo/wrapper/unit_tests/1 and is independent of FLASH. It is a standalone Fortran application that uses the iso_c_binding function c_f_pointer to construct a Fortran pointer object given a raw C memory address and a shape argument. To build, enter this directory and execute make "compiler name", where "compiler name" is gnu, intel, ibm, sun.

Old compilers, such as gfortran version 4.1.2, will give a compile-time error because iso_c_binding module is not available, and at least one version of gfortran, namely gfortran version 4.4.0, has an array indexing run-time bug that is detected by this unit test. The array indexing bug is documented at GCC bugzilla 40962 and gives the following incorrect answers shown in Figure 5.3.

If this test fails at compile-time or run-time then FLASH-Chombo applications will not work! You can save a lot of time and frustration by running this unit test before continuing with the build process.

In contrast to Paramesh, the Chombo source code is not included in the FLASH source tree. This means that you, the user, must manually download and then build Chombo before building FLASH applications.
CXX = icpc
FC = ifort
MPI = TRUE
MPICXX = /home/cdaley/software/mpich2/1.3.1/intel/bin/mpicxx
USE_64 = TRUE
USE_HDF = TRUE
HDFINCFLAGS = -I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include -DH5_USE_16_API
HDFLIBFLAGS = -L/home/cdaley/software/hdf5/1.8.5-patch1/intel/lib -lhdf5
HDFMPIINCFLAGS = -I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include -DH5_USE_16_API
HDFMPILIBFLAGS = -L/home/cdaley/software/hdf5/1.8.5-patch1/intel/lib -lhdf5
USE_MT = FALSE
syslibflags = -lstdc++ -lz

Figure 5.4: Example macro definitions used to build Chombo in a FLASH-compatible way.

that make use of Chombo. You should download version 3.0 or 3.1 of Chombo from
https://seesar.lbl.gov/anag/chombo/ as these are the only versions of Chombo that have been tested
with FLASH. Please refer to their user guide which is available from the same download page for help
building Chombo. It contains a very clear and detailed build procedure along with a Chombo support email
address that you can contact if you experience problems building Chombo.

You should build 1D, 2D and 3D versions of Chombo library in a MPI configuration (MPI = TRUE) with
parallel HDF5 (USE_HDF = TRUE). A parallel HDF5 implementation means a HDF5 installation that is
configured with --enable-parallel. You can check whether you have a parallel HDF5 build using the
following command line:

grep "Parallel HDF5" /path/to/hdf5/lib/libhdf5.settings

You should also explicitly turn off Chombo memory tracking (USE_MT = FALSE) because it is a feature
that does not work with FLASH’s usage of Chombo. Note that it is not always sufficient to set USE_MT =
FALSE in your Make.defs.local; for Intel compiler we also had to remove USE_MT = TRUE from the default
compiler settings in ./lib/mk/compiler/Make.defs.Intel.

Only after you have successfully built and run the Chombo unit tests described in their user guide should
you have any expectation that FLASH will work with Chombo library. For your reference we show the
Make.defs.local macro definitions that we have used to build Chombo on a x86_64 machine in a FLASH-
compatible way in Figure 5.4.

Compiling Chombo on IBM BG/P

The file src/BaseTools/ClockTicks.H from the Chombo-3.0 distribution needs to be modified
in order to compile Chombo on IBM BG/P. Change line 51 from

#elif defined(_POWER) || defined(_POWERPC) || defined(_-POWERPC-)

to

#elif defined(_POWER) || defined(_POWERPC) || defined(_-POWERPC-) ||
defined(_-powerpc-)

At this stage you can optionally build and run the unit test source/Grid/GridMain/Chombo/wrapper/-
unit_tests/2. This is a standalone Fortran MPI application that uses Chombo library to create a dis-
tributed mesh over 4 MPI processes. It can be built by entering this directory and then executing the
5.9. SETUP A FLASH+CHOMBO APPLICATION

command `make all`. We expect that this program will be helpful for debugging mixed-language programming issues between FLASH and Chombo (especially on untested architectures / compiler combinations) independently of a full-blown FLASH application. Note that you will need to edit the Makefile appropriately for your installation of Chombo.

In order to build a FLASH application with Chombo you must create a custom FLASH Makefile.h named `Makefile.h.chombo`. We use a new file named `Makefile.h.chombo` because it includes convenience shell functions (described later) that will fail if a Chombo package is not available. At minimum the FLASH Makefile.h.chombo must include definitions for the new macros `CFLAGS_CHOMBO` and `LIB_CHOMBO`. These can be manually defined, but this is not recommended because it does not guarantee that FLASH’s C++ wrapper classes are built the same way as the Chombo library. Example macro definitions for a FLASH 2D simulations are shown below:

```bash
CFLAGS_CHOMBO: -I/home/cdaley/flash/chomboForFlash/current/lib/include
        -DCH_SPACEDIM=2 -DCH_LINUX -DCH_IFC -DCH_MPI -DMPICH_SKIP_MPI
        -DULAM_WANT_MPI2CPP -DMPI_NO_CPPBIND -DCH_USE_SETVAL -DCH_USE_COMPLEX
        -DCH_USE_64 -DCH_USE_DOUBLE -DCH_USE_HDF5
-I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include
        -DH5_USE_16_API -DCH_FORT_UNDERSCORE
-I/home/cdaley/flash/chomboForFlash/current/lib/src/BaseTools
        -DCH_LANG_CC

LIB_CHOMBO: -L/home/cdaley/flash/chomboForFlash/current/lib
        -lamrtime-dependent2d.Linux.64.mpicxx.ifort.DEBUG.MPI
        -lamrtools2d.Linux.64.mpicxx.ifort.DEBUG.MPI
        -lboxtools2d.Linux.64.mpicxx.ifort.DEBUG.MPI
        -lbasetools2d.Linux.64.mpicxx.ifort.DEBUG.MPI
        -lstdc++
```

This is obviously hugely inconvenient because the macros must be kept in sync with the current Chombo build and also the Chombo library names are dimension dependent. Our mechanism in FLASH4 to hide this build complexity is to add shell functions in `Makefile.h.chombo` that extract:

- application dimensionality from `Flash.h`.
- information about the Chombo build from the `make vars` rule in the main Chombo Makefile for the specified application dimensionality.

The shell functions appear in the `Makefile.h.chombo` in sites/code.uchicago.edu, so this file should be used as your reference. A subset of this file is shown in Figure 5.5.

Notice that there are several temporary macros (`_DIM, _MPI, _CPP, _LIB, _PHDF_INC, _PHDF_LIB`) that store information about the Chombo build at location `CHOMBO_PATH`. These temporary macros are later used to give values for macros referenced in the actual FLASH Makefile.

You should now be ready to setup and build FLASH applications with Chombo mesh package.
CHOMBO_PATH = /home/cdaley/flash/chomboForFlash/current

# Extract dimensionality from Flash.h.
# The code in this section should not need to be modified.
#--------------------------------------------------------------------------------

_DIM := \$(shell grep "define NDIM" Flash.h | cut -d " " -f 3)

#--------------------------------------------------------------------------------

# Extract Chombo build information from the Makefile at CHOMBO_PATH.
# The code in this section should not need to be modified.
#--------------------------------------------------------------------------------

_MPI := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \ awk -F 'MPICXX=' '/^MPICXX/{print \$\$2}')
ifeq (\$(strip \$(\_MPI)),)
 \$(error "Chombo MPICXX variable is empty")
endif

_CPP := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \ awk -F 'CPPFLAGS=' '/^CPPFLAGS/{print \$\$2}')
_LIB := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \ awk -F 'config=' '/^config/{print \$\$2}')
_PHDF_INC := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \ awk -F 'HDFMPIINCFLAGS=' '/^HDFMPIINCFLAGS/{print \$\$2}')
_PHDF_LIB := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \ awk -F 'HDFMPILIBFLAGS=' '/^HDFMPILIBFLAGS/{print \$\$2}')

# Use Chombo build information to get consistent macro values for the FLASH build.
#--------------------------------------------------------------------------------

# Use two passes of dirname to strip the bin/mpicxx
MPI_PATH := \$(shell dirname \$(shell dirname \$(shell which \$(\_MPI))))
FCOMP = \${MPI_PATH}/bin/mpif90
CCOMP = \${MPI_PATH}/bin/mpicc
CPPCOMP = \${MPI_PATH}/bin/mpicxx
LINK = \${MPI_PATH}/bin/mpif90

CFLAGS_CHOMBO = -I\${CHOMBO_PATH}/lib/include \${_CPP} -DCH_LANG_CC
CFLAGS_HDF5 = \$(\_PHDF_INC)

LIB_CHOMBO = -L\$(CHOMBO_PATH)/lib \\ -lamrtimedependent\$(\_LIB) \\ -lamrtools\$(\_LIB) \\ -lboxtools\$(\_LIB) \\ -lbasetools\$(\_LIB) \\ -lstdc++
LIB_HDF5 = \$(\_PHDF_LIB) -lz

Figure 5.5: gmake shell functions to build FLASH and Chombo consistently.
Chapter 6

The Flash.h file

Flash.h is a critical header file in FLASH4 which holds many of the key quantities related to the particular simulation. The Flash.h file is written by the setup script and should not be modified by the user. The Flash.h file will be different for various applications. When the setup script is building an application, it parses the Config files, collects definitions of variables, fluxes, grid vars, species, and mass scalars, and writes a symbol (an index into one of the data structures maintained by the Grid unit) for each defined entity to the Flash.h header file. This chapter explains these symbols and some of the other important quantities and indices defined in the Flash.h file.

6.1 UNK, FACE(XYZ) Dimensions

Variables in a simulation are stored and manipulated by the Grid unit. The basic data structures in the Grid unit are 5-dimensional arrays: unk, facex, facey, and facez. The array unk stores the cell-centered values of various quantities (density, pressure, etc.). The facex, facey and facez variables store face-centered values of some or all of the same quantities. Face centered variables are commonly used in Magnetohydrodynamics (MHD) simulations to hold vector-quantity fields. The first dimension of each of these arrays indicates the variable, the 2nd, 3rd, and 4th dimensions indicate the cell location in a block, and the 5th dimension is the block identifier for the local processor. The size of the block, dimensions of the domain, and other parameters which influence the Grid data structures are defined in Flash.h.

NXB, NYB, NZB
The number of interior cells in the x,y,z-dimension per block

MAXCELLS
The maximum of (NXB,NYB,NZB)

MAXBLOCKS
The maximum number of blocks which can be allocated in a single processor

GRID_(IJK)LO
The index of the lowest numbered cell in the x,y,z-direction in a block (not including guard cells)

GRID_(IJK)HI
The index of the highest numbered cell in the x,y,z-direction in a block (not including guard cells)

GRID_(IJK)LO_GC
The index of the lowest numbered cell in the x,y,z-direction in a block (including guard cells)

GRID_(IJK)HI_GC
The index of the highest numbered cell in the x,y,z-direction in a block (including guard cells)
CHAPTER 6. THE FLASH.H FILE

NGUARD

The number of guard cells in each dimension.

All of these constants have meaning when operating in FIXEDBLOCKSIZE mode only. FIXEDBLOCKSIZE mode is when the sizes and the block bounds are determined at compile time. In NONFIXEDBLOCKSIZE mode, the block sizes and the block bounds are determined at runtime. PARAMESH always runs in FIXEDBLOCKSIZE mode, while the Uniform Grid can be run in either FIXEDBLOCKSIZE or NONFIXEDBLOCKSIZE mode. See Section 5.2 and Section 8.5.2 for more information.

6.2 Property Variables, Species and Mass Scalars

The unk data structure stores, in order, property variables (like density, pressure, temperature), the mass fraction of species, and mass scalars ¹. However, in FLASH4 the user does not need to be intimately aware of the unk array layout, as starting and ending indices of these groups of quantities are defined in Flash.h. The following pre-processor symbols define the indices of the various quantities related to a given cell. These symbols are primarily used to perform some computation with all property variables, species mass fractions, or all mass scalars.

NPROP_VARS

The number of property variables in the simulation

NSPECIES

The total number of species in the simulation

NMASS_SCALARS

The number of mass scalars in the simulation

NUNK_VARS

The total number of quantities stored for each cell in the simulation. This equals NPROP_VARS + NSPECIES + NMASS_SCALARS

PROP_VARS_BEGIN,PROP_VARS_END

The indices in the unk array used for property variable data

SPECIES_BEGIN,SPECIES_END

The indices in the unk array used for species data

MASS_SCALARS_BEGIN,MASS_SCALARS_END

The indices in the unk array used for mass scalars data

UNK_VARS_BEGIN,UNK_VARS_END

The low and high indices for the unk array

The indices where specific properties (e.g., density) are stored can also be accessed via pre-processor symbols. All properties are declared in Config files and consist of 4 letters. For example, if a Config file declares a “dens” variable, its index in the unk array is available via the pre-processor symbol DENS_VAR (append _VAR to the uppercase name of the variable) which is guaranteed to be an integer. The same is true for species and mass scalars. In the case of species, the pre-processor symbol is created by appending _SPEC to the uppercase name of the species (e.g., SF6_SPEC, AIR_SPEC). Finally, for mass scalars, _MSCALAR is appended to the uppercase name of the mass scalars.

It is inadvisable to name variables, species, and mass scalars with the same prefix as post-processing routines have difficulty deciphering the type of data from the output files. For example, don’t create

¹See (14.9) for more information about mass scalars
a variable “temp” to hold temperature and a mass scalar “temp” indicating a temporary variable. Although the Flash.h file can distinguish between these two types of variables, many plotting routines such as fidlr3.0 cannot.

6.3 Fluxes

The fluxes are stored in their own data structure and are only necessary when an adaptive grid is in use. The index order works in much the same way as with the unk data structure. There are the traditional property fluxes, like density, pressure, etc. Additionally, there are species fluxes and mass scalars fluxes. The name of the pre-processor symbol is assembled by appending _FLUX to the uppercase name of the declared flux (e.g., EINT_FLUX, U_FLUX). For flux species and flux mass scalars, the suffix _FLUX_SPECIES and _FLUX_MSCALAR are appended to the uppercase names of flux species and flux mass scalars, respectively, as declared in the Config file. Useful defined variables are calculated as follows:

NPROP_FLUX
The number of property variables in the simulation

NSPECIES_FLUX
The total number of species in the simulation

NMASS_SCALARS_FLUX
The number of mass scalars in the simulation

NFLUXES
The total number of quantities stored for each cell in the simulation. This equals (NPROP_FLUX + NSPECIES_FLUX + NMASS_SCALARS_FLUX)

PROP_FLUX_BEGIN, PROP_FLUX_END
The indices in the fluxes data structure used for property variable data

SPECIES_FLUX_BEGIN, SPECIES_FLUX_END
The indices in the fluxes data structure used for species data

MASS_SCALARS_FLUX_BEGIN, MASS_SCALARS_FLUX_END
The indices in the fluxes data structure used for mass scalars data

FLUXES_BEGIN
The first index for the fluxes data structure

6.4 Scratch Vars

In FLASH4 the user is allowed to declare ‘scratch’ space for grid scope variables which resemble cell-centered or face-centered in shape and are dimensioned accordingly, but are not advected or transformed by the usual evolution steps. They do not participate in the guard-cell filling or regridding processes. For example a user could declare a scratch variable to store the temperature change on the grid from one timestep to another. They can be requested using keyword SCRATCHCENTERVAR for cell-centered scratch variables, or SCRATCHFACEVAR for face-centered scratch variables. A special case is SCRATCHVAR, which has one extra cell than the cell-centered variables along every dimension. We have provided this data structure to enable the reuse of the same scratch space by both cell-centered and each of the face-centered variables. Similar to the mesh variables used in the evolution, the scratch data structures are 4-dimensional arrays per block, where the first dimension enumerates the variables and the next three dimensions are the spatial dimensions. Scratch variables are indexed by postpending one of _SCRATCH_GRID_VAR, _SCRATCH_CENTER_VAR or _SCRATCH_FACE_X/Y/Z_VAR to the capitalized four letter variable defined in the Config file. Similarly to property variables, NSCRATCHCENTER_VARS, SCRATCHCENTER_VARS_BEGIN, and SCRATCHCENTER_VARS_END
are defined to hold the number and endpoints of the cell-centered scratch variables. For face-centered scratch variables the CENTER is in the above terms is replaced with FACEX/Y/Z while for SCRATCHVARS, the CENTER is replace with GRID.

6.5 Fluid Variables Example

The snippet of code below shows a Config file and parts of a corresponding Flash.h file.

```
# Config file for explicit split PPM hydrodynamics.
# source/physics/Hydro/HydroMain/split/PPM

REQUIRES physics/Hydro/HydroMain/utilities
REQUIRES physics/Eos

DEFAULPPMKernel

VARIABLE dens TYPE: PER_VOLUME # density
VARIABLE velx TYPE: PER_MASS # x-velocity
VARIABLE vely TYPE: PER_MASS # y-velocity
VARIABLE velz TYPE: PER_MASS # z-velocity
VARIABLE pres TYPE: GENERIC # pressure
VARIABLE ener TYPE: PER_MASS # specific total energy (T+U)
VARIABLE temp TYPE: GENERIC # temperature
VARIABLE eint TYPE: PER_MASS # specific internal energy

FLUX rho
FLUX u
FLUX p
FLUX ut
FLUX utt
FLUX e
FLUX eint

SCRATCHVAR otmp
SCRATCHCENTERVAR ftmp

The Flash.h files would declare the property variables, fluxes and scratch variables as: (The setup script alphabetizes the names.)

#define DENS_VAR 1
#define EINT_VAR 2
#define ENER_VAR 3
#define PRES_VAR 4
#define TEMP_VAR 5
#define VELX_VAR 6
#define VELY_VAR 7
#define VELZ_VAR 8
#define E_FLUX 1
#define EINT_FLUX 2
#define P_FLUX 3
#define RHO_FLUX 4
#define U_FLUX 5
6.6 Particles

6.6.1 Particles Types

FLASH4 now supports the co-existence of multiple particle types in the same simulation. To facilitate this ability, the particles are now defined in the Config files with PARTICLETYPE keyword, which is also accompanied by an associated initialization and mapping method. The following example shows a Config file with passive particles, and the corresponding generated Flash.h lines.

Config file:

PARTICLETYPE passive INITMETHOD lattice MAPMETHOD quadratic ADVMETHOD rungekutta

Flash.h:

#define PASSIVE_PART_TYPE 1
#define PART_TYPES_BEGIN CONSTANT_ONE
#define NPART_TYPES 1
#define PART_TYPES_END (PART_TYPES_BEGIN + NPART_TYPES - CONSTANT_ONE)

One line describing the type, initialization, and mapping methods must be provided for each type of particle included in the simulation.

6.6.2 Particles Properties

Particle properties are defined within the particles data structure. The individual properties will be listed in Flash.h if the Particles unit is defined in a simulation. The variables NPART_PROPS, PART_PROPS_BEGIN and PART_PROPS_END indicate the number and location of particle properties indices. For example if a Config file has the following specifications:

PARTICLEPROP dens
PARTICLEPROP pres
PARTICLEPROP velx

then the relevant portion of Flash.h will contain:

```c
#define NPART_PROPS 3
#define PART_PROPS_BEGIN CONSTANT_ONE
#define PART_PROPS_END (PART_PROPS_BEGIN + NPART_PROPS - CONSTANT_ONE)
```
#define DENS_PART_PROP 1
#define PRES_PART_PROP 2
#define VELX_PART_PROP 3
...
#define PART_PROPS_BEGIN CONSTANT_ONE
#define NPART_PROPS 3
#define PART_PROPS_END (PART_PROPS_BEGIN + NPART_PROPS - CONSTANT_ONE)

6.7 Non-Replicated Variable Arrays

For each non-replicated variable array defined in the Config file (see Section 5.5.1), various macros and constants are defined to assist the user in retrieving the information from the NONREP line as well as determining the distribution of array element variables across the meshes.

6.7.1 Per-Array Macros

For each non-replicated variable array FOO as defined by the following Config line:

```
NONREP unktype FOO localmax globalparam ioformat
```

the following will be placed into Flash.h:

- **#define FOO_NONREP**
  An integer constant greater than zero to be used as an id for this array. The user should rarely ever need to use this value. Instead, it is much more likely that the user would just query the preprocessor for the existence of this symbol (via `#ifdef FOO_NONREP`) to enable sections of code.

- **#define FOO_NONREP_LOC2UNK(loc)**
  Given the 1-based index into this mesh’s local subset of the global array, returns the UNK index where that variable is stored.

- **#define FOO_NONREP_MAXLOCS**
  The integer constant localmax as supplied in the Config file NONREP line.

- **#define FOO_NONREP_RPCOUNT**
  The string literal value of globalparam from the NONREP line.

6.7.2 Array Partitioning Macros

These are the macros used to calculate the subset of indices into the global variable array each mesh is responsible for:

- **#define NONREP_NLOCS(mesh,meshes,globs)**
  Returns the number of elements in the mesh’s local subset. This value will always be less than or equal to FOO_NONREP_MAXLOCS.

- **#define NONREP_LOC2GLOB(loc,mesh,meshes)**
  Maps an index into a mesh’s local subset to its index in the global array.

- **#define NONREP_GLOB2LOC(glob,mesh,meshes)**
  Maps an index of the global array to the index into a mesh’s local subset. If the mesh provided does not have that global element the result is undefined.

- **#define NONREP_MESHOFGLOB(glob,meshes)**
  Returns the mesh that owns a given global array index.

Descriptions of arguments to the above macros:

- **loc**: 1-based index into the mesh-local subset of the global array’s indices.
6.7. NON-REPLICATED VARIABLE ARRAYS

- **glob**: 1-based index into the global array.

- **glob**: Length of the global array. This should be the value read from the array's runtime parameter `FOO NONREP RTCOUNT`.

- **mesh**: 0-based index of the mesh in question. For a processor, this is its rank in the `MESH ACROSS COMM` communicator.

- **meshes**: The number of meshes. This should be the value of the runtime parameter `meshCopyCount`, or equivalently the number of processors in the `MESH ACROSS COMM` communicator.

### 6.7.3 Example

In this example the global array `FOO` has eight elements, but there are three meshes, so two of the meshes will receive three elements of the array and one will get two. How that distribution is decided is hidden from the user. The output datasets will contain variables `foo1 ... foo8`.

**Config:**

```
NONREP MASS SCALAR FOO 3 numFoo foo?
```

**flash.par:**

```
meshCopyCount = 3
numFoo = 8
```

**Fortran 90:**

```
#include "Flash.h"
! this shows how to iterate over the UNKs corresponding to this
! processor's subset of FOO
integer :: nfooglob, nfooloc ! number of foo vars, global and local
integer :: mesh, nmesh ! this mesh number, and total number of meshes
integer :: fooloc, fooglob ! local and global foo indices
integer :: unk
!
! get the location in UNK
unk = FOO NONREP LOC2UNK(fooloc)
! get the global index
fooglob = NONREP LOC2GLOB(fooloc, mesh, nmesh)
!
! you have what you need, do your worst...
end do
```
6.8 Other Preprocessor Symbols

The constants \texttt{FIXEDBLOCKSIZE} and \texttt{NDIM} are both included for convenience in this file. \texttt{NDIM} gives the dimensionality of the problem, and \texttt{FIXEDBLOCKSIZE} is defined if and only if fixed blocksize mode is selected at compile time.

Each \texttt{Config} file can include the \texttt{PPDEFINE} keyword to define additional preprocessor symbols. Each “\texttt{PPDEFINE symbol [value]}” gets translated to a “\texttt{#define symbol [value]}”. This mechanism can be used to write code that depends on which units are included in the simulation. See Section 5.5.1 for concrete usage examples.
Part III

Driver Unit
Chapter 7

Driver Unit

The **Driver** unit controls the initialization and evolution of FLASH simulations. In addition, at the highest level, the **Driver** unit organizes the interaction between units. Initialization can be from scratch or from a stored checkpoint file. For advancing the solution, the drivers can use either an operator-splitting technique (**Split**) or an unsplit operator (**Unsplit**). The unsplit operator is currently specific to the problems that use the unsplit StaggeredMesh MHD solver. The **Driver** unit also calls the **IO** unit at the end of every timestep to produce checkpoint files, plot files, or other output.

7.1 Driver Routines

The most important routines in the **Driver** API are those that initialize, evolve, and finalize the FLASH program. The file Flash.F90 contains the main FLASH program (equivalent to **main()** in C). The default top-level program of FLASH, Simulation/Flash.F90, calls **Driver** routines in this order:

```fortran
program Flash
    implicit none

    call Driver_initParallel()
    call Driver_initFlash()
    call Driver_evolveFlash()
    call Driver_finalizeFlash()
```

Figure 7.1: The **Driver** unit directory tree.

---

85
Therefore the no-operation stubs for these routines in the Driver source directory must be overridden by an implementation function in a unit implementation directory under the Driver or Simulation directory trees, in order for a simulation to perform any meaningful actions. The most commonly used implementation for each of these files is located in the Driver/DriverMain/Split unit implementation directory.

7.1.1 Driver_initFlash

The first of these routines is Driver_initParallel, which initializes the parallel environment for the simulation. New in FLASH4 is an ability to replicate the mesh where more than one copy of the discretized mesh may exist with some overlapping and some non-overlapping variables. Because of this feature, the parallel environment differentiates between global and mesh communicators. All the necessary communicators, and the attendant meta-data is generated in this routine. Also because of this modification, runtime parameters such as iProcs, jProcs etc, which were under the control of the Grid unit in FLASH3, are now under the control of the Driver unit. Several new accessor interface allow other code units to query the driver unit for this information. The Driver_initFlash, the next routine, in general calls the initialization routines in each of the units. If a unit is not included in a simulation, its stub (or empty) implementation is called. Having stub implementations is very useful in the Driver unit because it allows the user to avoid writing a new driver for each simulation. For a more detailed explanation of stub implementations please see Section 4.2. It is important to note that when individual units are being initialized, order is often very important and the order of initialization is different depending on whether the run is from scratch or being restarted from a checkpoint file.

7.1.2 Driver_evolveFlash

The next routine is Driver_evolveFlash which controls the timestepping of the simulation, as well as the normal termination of FLASH based on time. Driver_evolveFlash checks the parameters tmax, nend and zFinal to determine that the run should end, having reached a particular point in time, a certain number of steps, or a particular cosmological redshift, respectively. Likewise the initial simulation time, step number and cosmological redshift for a simulation can be set using the runtime parameters tmin, nbegin, and zInitial. This version of FLASH includes versions of Driver_evolveFlash for directionally-split and unsplit staggered mesh operators.

7.1.2.1 Strang Split Evolution

The code in the Driver/DriverMain/Split unit implementation directory is the default time update method, and is used for most problems currently run with FLASH. The default Driver_evolveFlash implements a Strang-split method of time advancement where each physics unit updates the solution data for two equal timesteps – thus the sequence of calls to the physics units is: Hydro, SourceTerms, Particles, Gravity, Hydro, SourceTerms, Particles, Gravity, Grid. The hydrodynamics update routines take a “sweep order” argument since they must be directionally split to work with this driver. Here, the first call uses the ordering x − y − z, and the second call uses z − y − x. Each of the update routines is assumed to directly modify the solution variables. At the end of one loop of timestep advancement, the condition for updating the mesh refinement pattern is tested if the adaptive mesh is being used, and a refinement update is carried out if required.

7.1.2.2 Unsplit Evolution

The driver implementation in the Driver/DriverMain/Unsplit is used specifically for the two unsplit solvers: unsplit staggered mesh MHD solver (Section 14.3.3) and the unsplit gas hydrodynamics solver (Section 14.1.3). This alternative implementation calls each of the physics routines only once and each call advances solution vectors by one timestep. At the end of one loop of timestep advancement, the condition for updating the adaptive mesh refinement pattern is tested and applied.
7.1. DRIVER ROUTINES

7.1.2.3 Super-Time-Stepping (STS)

A new timestepping method implemented in FLASH4 is a technique called Super-Time-Stepping (STS). The STS is a simple explicit method which is used to accelerate restrictive parabolic timestepping advancements \( \Delta t_{\text{CFL, para}} \approx \Delta x^2 \) by relaxing the CFL stability condition of parabolic equation system.

The STS has been proposed by Alexiades et al., (1996), and used in computational astrophysics and sciences recenly (see Mignone et al., 2007; O’Sullivan & Downes, 2006; Commerçon et al., 2011; Lee, D. et al, 2011) for solving systems of parabolic PDEs numerically. The method increases its effective time steps \( \Delta t_{\text{sts}} \) using two properties of stability and optimality in Chebychev polynomial of degree \( n \). These properties optimally maximize the time step \( \Delta t_{\text{sts}} \) by which a solution vector can be evolved. A stability condition is imposed only after each time step \( \Delta t_{\text{sts}} \), which is further subdivided into smaller \( N_{\text{sts}} \) sub-time steps, \( \tau_i \), that is, \( \Delta t_{\text{sts}} = \sum_{i=1}^{N_{\text{sts}}} \tau_i \), where the sub-time step is given by

\[
\tau_i = \Delta t_{\text{CFL, para}} \left[ \left( -1 + \nu_{\text{sts}} \right) \cos \left( \frac{\pi(2j - 1)}{2N_{\text{sts}}} + 1 + \nu_{\text{sts}} \right) \right]^{-1},
\]

where \( \Delta t_{\text{CFL, para}} \) is an explicit time step for a given parabolic system based on the CFL stability condition. \( \nu \) (\( \nu_{\text{sts}} \)) is a free parameter less than unity. For \( \nu \to 0 \), STS is asymptotically \( N_{\text{sts}} \) times faster than the conventional explicit scheme based on the CFL condition. During the \( N_{\text{sts}} \) sub-time steps, the STS method still solves solutions at each intermediate step \( \tau_i \); however, such solutions should not be considered as meaningful solutions.

Extended from the original STS method for accelerating parabolic timestepping, our STS method advances advection and/or diffusion (hyperbolic and/or parabolic) system of equations. This means that the STS algorithm in FLASH invokes a single \( \Delta t_{\text{sts}} \) for both advection and diffusion, and does not use any sub-cycling for diffusion based on a given advection time step. In this case, \( \tau_i \) is given by

\[
\tau_i = \Delta t_{\text{CFL}} \left[ \left( -1 + \nu_{\text{sts}} \right) \cos \left( \frac{\pi(2j - 1)}{2N_{\text{sts}}} + 1 + \nu_{\text{sts}} \right) \right]^{-1},
\]

where \( \Delta t_{\text{CFL}} \) can be an explicit time step for advection (\( \Delta t_{\text{CFL, adv}} \)) or parabolic (\( \Delta t_{\text{CFL, para}} \)) systems. In cases of advection-diffusion system, \( \Delta t_{\text{CFL}} \) takes (\( \Delta t_{\text{CFL, para}} \)) when it is smaller than (\( \Delta t_{\text{CFL, adv}} \)); otherwise, FLASH’s timestepping will proceed without using STS iterations (i.e., using standard explicit timestepping that is either Strang split evolution or unsplit evolution).

Since the method is explicit, it works equally well on both a uniform grid and AMR grids without modification. The STS method is first-order accurate in time.

Both directionally-split and unsplit hydro solvers can use the STS method, simply by invoking a runtime parameter \texttt{useSTS = .true.} in \texttt{flash.par}. There are couple of runtime parameters that control solution accuracy and stability. They are decribed in Table 7.1.

7.1.2.4 Runtime Parameters

The Driver unit supplies certain runtime parameters regardless of which type of driver is chosen. These are described in the online Runtime Parameters Documentation page.
### Table 7.1: Runtime parameters for STS

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useSTS</td>
<td>logical</td>
<td>.false.</td>
<td>Enable STS</td>
</tr>
<tr>
<td>nstepTotalSTS</td>
<td>integer</td>
<td>5</td>
<td>Suggestion: $\sim 5$ for hyperbolic; $\sim 10$ for parabolic</td>
</tr>
<tr>
<td>nuSTS</td>
<td>real</td>
<td>0.2</td>
<td>Suggestion: $\sim 0.2$ for hyperbolic; $\sim 0.01$ for parabolic. It is known that a very low value of $\nu$ may result in unstable temporal integrations, while a value close to unity can decrease the expected efficiency of STS.</td>
</tr>
<tr>
<td>useSTSforDiffusion</td>
<td>logical</td>
<td>.false.</td>
<td>This setup will use the STS for overcoming small diffusion time steps assuming $\Delta t_{\text{CFL,adv}} &gt; \Delta t_{\text{CFL,para}}$. In implementation, it will set $\Delta t_{\text{CFL}} = \Delta t_{\text{CFL,para}}$ in Eqn. 7.2. Do not allow to turn on this switch when there is no diffusion (viscosity, conductivity, and magnetic resistivity) used.</td>
</tr>
<tr>
<td>allowDtSTSDominate</td>
<td>logical</td>
<td>.false.</td>
<td>If true, this will allow to have $\tau_i &gt; \Delta t_{\text{CFL,adv}}$, which may result in unstable integrations.</td>
</tr>
</tbody>
</table>

#### FLASH Transition

The **Driver** unit no longer provides runtime parameters, physical constants, or log file management. Those services have been placed in separate units. The **Driver** unit also does not declare boolean values to include a unit in a simulation or not. For example, in FLASH2, the **Driver** declared a runtime parameter `iburn` to turn on and off burning.

```plaintext
if(iburn) then
    call burning ....
end if
```

In FLASH4 the individual unit declares a runtime parameter that determines whether the unit is used during the simulation e.g., the **Burn** unit declares `useBurn` within the **Burn** unit code that turns burning on or off. This way the **Driver** is no longer responsible for knowing what is included in a simulation. A unit gets called from the **Driver**, and if it is not included in a simulation, a stub gets called. If a unit, like **Burn**, is included but the user wants to turn burning off, then the runtime parameter declared in the **Burn** unit would be set to false.

#### 7.1.3 Driver_finalizeFlash

Finally, the **Driver** unit calls `Driver_finalizeFlash` which calls the finalize routines for each unit. Typically this involves deallocating memory and any other necessary cleanup.

#### 7.1.4 Driver accessor functions

In FLASH4 the **Driver** unit also provides a number of accessor functions to get data stored in the **Driver** unit, for example `Driver_getDt, Driver_getNStep, Driver_getElapsedWCTime, Driver_getSimTime`. 
FLASH Transition

In FLASH4 most of the quantities that were in the FLASH2 database are stored in the Grid unit or are replaced with functionality in the Flash.h file. A few scalars quantities like dt, the current timestep number nstep, simulation time and elapsed wall clock time, however, are now stored in the Driver_data FORTRAN90 module.

The Driver unit API also defines two interfaces for halting the code, Driver_abortFlash and Driver_abortFlashC.c. The 'c' routine version is available for calls written in C, so that the user does not have to worry about any name mangling. Both of these routines print an error message and call MPI_Abort.
Part IV

Infrastructure Units
Figure 8.1: The Grid unit: structure of GridMain and GridBoundaryCondition subunits.
Figure 8.2: The Grid unit: structure of GridParticles subunit.
8.1 Overview

The Grid unit has four subunits: GridMain is responsible for maintaining the Eulerian grid used to discretize the spatial dimensions of a simulation; GridParticles manages the data movement related to active, and Lagrangian tracer particles; GridBoundaryConditions handles the application of boundary conditions at the physical boundaries of the domain; and GridSolvers provides services for solving some types of partial differential equations on the grid. In the Eulerian grid, discretization is achieved by dividing the computational domain into one or more sub-domains or blocks, and using these blocks as the primary computational entity visible to the physics units. A block contains a number of computational cells \((n_{xb} \text{ in the } x\text{-direction}, n_{yb} \text{ in the } y\text{-direction, and } n_{zb} \text{ in the } z\text{-direction})\). A perimeter of guardcells, of width \(n_{guard}\) cells in each coordinate direction, surrounds each block of local data, providing it with data from the neighboring blocks or with boundary conditions, as shown in Figure 8.4. Since the majority of physics solvers used in FLASH are explicit, a block with its surrounding guard cells becomes a self-contained computational domain. Thus the physics units see and operate on only one block at a time, and this abstraction is reflected in their design.

Therefore any mesh package that can present a self contained block as a computational domain to a client unit can be used with FLASH. However, such interchangeability of grid packages also requires a careful design of the Grid API to make the underlying management of the discretized grid completely transparent to outside units. The data structures for physical variables, the spatial coordinates, and the management of the grid are kept private to the Grid unit, and client units can access them only through accessor functions. This strict protocol for data management along with the use of blocks as computational entities enables FLASH to abstract the grid from physics solvers and facilitates the ability of FLASH to use multiple mesh packages.

Any unit in the code can retrieve all or part of a block of data from the Grid unit along with the coordinates of corresponding cells; it can then use this information for internal computations, and finally return the modified data to the Grid unit. The Grid unit also manages the parallelization of FLASH. It consists of a suite of subroutines which handle distribution of work to processors and guard cell filling. When using an adaptive mesh, the Grid unit is also responsible for refinement/derefinement and conservation of flux across block boundaries.

FLASH can interchangeably use either a uniform or adaptive grid for most problems. Additionally, a new feature in FLASH4 is an option to replicate the mesh; that is processors are assumed to be partitioned into groups, each group gets a copy of the entire domain mesh. This feature is useful when it is possible to decompose the computation based upon certain compute intensive tasks that apply across the domain. One such example is radiation transfer with multigroup flux limited diffusion where each group needs an implicit solve. Here the state variable of the mesh are replicated on each group of processors, while the groups are unique. Thus at the cost of some memory redundancy, it becomes possible to compute a higher
fidelity problem (see Chapter 22 for an example). Because of this feature, the parallel environment of the simulation is now controlled by the Driver which differentiates between global communicators and mesh communicators. The Grid unit queries the Driver unit for mesh communicators. In all other respects this change is transparent to the Grid unit. Mesh replication can be invoked through the runtime parameter `meshCopyCount`.

The uniform grid supported in FLASH discretizes the physical domain by placing grid points at regular intervals defined by the geometry of the problem. The grid configuration remains unchanged throughout the simulation, and exactly one block is mapped per processor. An adaptive grid changes the discretization over the course of the computation, and several blocks can be mapped to each computational processor. Two AMR packages are currently supported in FLASH for providing adaptive grid capability. The block-structured oct-tree based AMR package, `PARAMESH` has been the work horse since the beginning of the code. Version 2 and Version 4 of paramesh are both supported, version 2 is kept mostly to enable cross checking against FLASH2 results. FLASH4, for the first time, includes patch based `Chombo` as an alternative AMR package. By default, `PARAMESH` 4 is chosen when setting up an application, unless another implementation is explicitly specified. The use of paramesh 2 is deprecated, and `Chombo` has not so far been used for production at the Flash Center.
The following two commands will create the same (identical) application: a simulation of a Sod shock tube in 3 dimensions with PARAMESH 4 managing the grid.

```bash
./setup Sod -3d -auto
./setup Sod -3d -auto -unit=Grid/GridMain/paramesh/paramesh4/Paramesh4.0
```

However, if the command is changed to

```bash
./setup Sod -3d -auto -unit=Grid/GridMain/UG
```

the application is set up with a uniform grid instead. Additionally, because two different grids types are supported in FLASH, the user must match up the correct IO alternative implementation with the correct Grid alternative implementation. Please see Chapter 9 for more details. Note that the setup script has capabilities to let the user set up shortcuts, such as “+ugio”, which makes sure that the appropriate branch of IO is included when the uniform grid is being used. Please see Section 5.3 for more information. Also see grid tips for shortcuts useful for the Grid unit.

## 8.2 GridMain Data Structures

The **Grid** unit is the most extensive infrastructure unit in the FLASH code, and it owns data that most other units wish to fetch and modify. Since the data layout in this unit has implications on the manageability and performance of the code, we describe it in some detail here.

FLASH can be run with a grid discretization that assumes cell-centered data, face-centered data, or a combination of the two. Paramesh and Uniform Grid store physical data in multidimensional F90 arrays; cell-centered variables in `unk`, short for “unknowns”, and face-centered variables in arrays called `facevarx`, `facevary`, and `facevarz`, which contain the face-centered data along the x, y, and z dimensions, respectively. The cell-centered array `unk` is dimensioned as `array(NUNK_VARS,nxb,nyb,nzb,blocks)`, where `nxb`, `nyb`, `nzb` are the spatial dimensions of a single block, and `blocks` is the number of blocks per processor (MAXBLOCKS for PARAMESH and 1 for UG). The face-centered arrays have one extra data point along the dimension they are representing, for example `facevarx` is dimensioned as `array(NFACE_VARS,nxb+1,nyb,nzb,blocks)`. Some or all of the actual values dimensioning these arrays are determined at application setup time. The number of variables and the value of MAXBLOCKS are always determined at setup time. The spatial dimensions `nxb,nyb,nzb` can either be fixed at setup time, or they may be determined at runtime. These two modes are referred to as FIXEDBLOCKSIZE and NONFIXEDBLOCKSIZE. Chombo, which is written in C++, maintains its internal data very differently. However, through the wrapper layer provided by Grid, the Chombo data structures mimic the data layout understood by the FLASH solvers. Details of Grid implementation with Chombo are described in section Section 8.7

All values determined at setup time are defined as constants in a file `Flash.h` generated by the setup tool. This file contains all application-specific global constants such as the number and naming of physical variables, number and naming of fluxes and species, etc.; it is described in detail in Chapter 6.

For cell-centered variables, the **Grid** unit also stores a **variable type** that can be retrieved using the `Simulation_getVarnameType` routine; see Section 5.5.1 for the syntax and meaning of the optional `TYPE` attribute that can be specified as part of a `VARIABLE` definition read by the setup tool.

In addition to the primary physical variables, the **Grid** unit has another set of data structures for storing auxiliary fluid variables. This set of data structures provides a mechanism for storing such variables whose spatial scope is the entire physical domain, but who do not need to maintain their guard cells updated at all times. The data structures in this set include: `SCRATCHCENTERVAR`, which has the same shape as the cell centered variables data structure; and `SCRATCHFACEXVAR`, `SCRATCHFACEYVAR` and `SCRATCHFACEZVAR`, which have the same shape as the corresponding face centered variables data structures. Early releases of
FLASH3 had `SCRATCHVAR`, dimensioned array(`NSCRATCH_GRID_VARS,nxb+1,nyb+1,nzb+1,blocks`), as the only grid scope scratch data structure. For reasons of backward compatibility, and to maximize reusability of space, `SCRATCHVAR` continues to exist as a supported data structure, though its use is deprecated. The datastructures for face variables, though supported, are not used anywhere in the released code base. The unsplit MHD solver `StaggeredMesh` discussed in Section 14.3.3 gives an example of the use of some of these data structures. It is important to note that there is no guardcell filling for the scratch variables, and the values in the scratch variables become invalid after a grid refinement step. While users can define scratch variables to be written to the plotfiles, they are not by default written to checkpoint files. The `Grid` unit also stores the metadata necessary for work distribution, load balancing, and other housekeeping activities. These activities are further discussed in Section 8.5 and Section 8.6, which describe individual implementations of the `Grid` unit.

### 8.3 Computational Domain

The size of the computational domain in physical units is specified at runtime through the `(xmin, xmax)`, `(ymin, ymax)`, and `(zmin, zmax)` runtime parameters. When working with curvilinear coordinates (see below in Section 8.11), the extrema for angle coordinates are specified in degrees. Internally all angles are represented in radians, so angles are converted to radians at `Grid` initialization.

---

**FLASH Transition**

The convention for specifying the ranges for angular coordinates has changed from FLASH2, which used units of $\pi$ instead of degrees for angular coordinates.

---

The physical domain is mapped into a computational domain at problem initialization through routine `Grid_initDomain` in PARAMESH and Chombo, and `Grid_init` in UG. When using the uniform grid UG, the mapping is easy: one block is created for each processor in the run, which can be sized either at build time or runtime depending upon the mode of UG use. \(^1\) Further description can be found in Section 8.5. When using the AMR grid PARAMESH, the mapping is non-trivial. The adaptive mesh `gr_createDomain` function creates an initial mesh of `nblockx * nblocky * nblockz` top level blocks, where `nblockx`, `nblocky`, and `nblockz` are runtime parameters which default to 1.\(^2\) The resolution of the computational domain is usually very coarse and unsuitable for computation after the initial mapping. The `gr_expandDomain` routine remedies the situation by applying the refinement process to the initial domain until a satisfactory level of resolution is reached everywhere in the domain. This method of mapping the physical domain to computational domain is effective because the resultant resolution in any section is related to the demands of the initial conditions there.

---

\(^1\)Note that the term processor, as used here and elsewhere in the documentation, does not necessarily correspond to a separate hardware processor. It is also possible to have several logical “processors” mapped to the same hardware, which can be useful for debugging and testing; this is a matter for the operating environment to regulate.

\(^2\)The `gr_createDomain` function also can remove certain blocks of this initial mesh, if this is requested by a non-default `Simulation_defineDomain` implementation.
8.4 Boundary Conditions

Much of the FLASH3 code within the Grid unit that deals with implementing boundary conditions has been organized into a separate subunit, GridBoundaryConditions. Note that the following aspects are still handled elsewhere:

- Recognition of boundary condition names as strings (in runtime parameters) and constants (in the source code); these are defined in RuntimeParameters_mapStrToInt and in constants.h, respectively.

- Handling of periodic boundary conditions; this is done within the underlying GridMain implementation. When using PARAMESH, the subroutine gr_createDomain is responsible for setting the neighbors of top-level blocks (to either other top-level blocks or to external boundary conditions) at initialization, after Nblockx × Nblocky × Nblockz root blocks have been created. Periodic (wrap-around) boundary conditions are initially configured in this routine as well. If periodic boundary conditions are set in the x-direction, for instance, the first blocks in the x-direction are set to have as their left-most neighbor the blocks that are last in the x-direction, and vice versa. Thus, when the guard cell filling is performed, the periodic boundary conditions are automatically maintained.

- Handling of user-defined boundary conditions; this should be implemented by code under the Simulation directory.

- Low-level implementation and interfacing, such as are part of the PARAMESH code.

- Behavior of particles at a domain boundary. This is based on the boundary types described below, but their handling is implemented in GridParticles.

Although the GridBoundaryConditions subunit is included in a setup by default, it can be excluded (if no Config file “REQUIRES” it) by specifying -without-unit=Grid/GridBoundaryConditions. This will generally only make sense if all domain boundaries are to be treated as periodic. (All relevant runtime parameters xl_boundary_type etc. need to be set to “periodic” in that case.)

8.4.1 Boundary Condition Types

Boundary conditions are determined by the physical problem. Within FLASH, the parallel structure of blocks means that each processor works independently. If a block is on a physical boundary, the guard
cells are filled by calculation since there are no neighboring blocks from which to copy values. Boundaries are selected by setting runtime parameters such as xl_boundary_type (for the ‘left’ X–boundary) to one of the supported boundary types (Section 8.4.1) in flash.par. Even though the runtime parameters for specifying boundary condition types are strings, the Grid unit understands them as defined integer constants defined in the file constants.h, which contains all global constants for the code. The translation from the string specified in “flash.par” to the constant understood by the Grid unit is done by the routine RuntimeParameters_mapStrToInt.

\[\begin{array}{|c|}
\hline
ab_boundary_type & Description \\
\hline
periodic & Periodic (‘wrap-around’) \\
reflect & Non-penetrating boundaries; plane symmetry, the normal vector components change sign \\
outflow & Zero-gradient boundary conditions; allows shocks to leave the domain \\
diode & like outflow, but fluid velocities are never allowed to let matter flow into the domain: normal velocity components are forced to zero in guard cells if necessary \\
axisymmetric & like reflect, but both normal and toroidal vector components change sign. Typically used with cylindrical geometry (R-Z) for the Z symmetry axis. \\
eqtsymmetric & like reflect for velocities but the magnetic field components, poloidal and toroidal, change sign. The sign of the normal magnetic field component remains the same. Typically used with cylindrical geometry (R-Z) for the R axis to emulate equatorial symmetry. \\
hydrostatic-f2 & Hydrostatic boundary handling as in FLASH2. See remark in text. \\
hydrostatic-f2+nvrefl, hydrostatic-f2+nvout, hydrostatic-f2+nvdiode & Variants of hydrostatic-f2, where the normal velocity is handled specially in various ways, analogous to reflect, outflow, and diode boundary conditions, respectively. See remark in text. \\
user-defined or user & The user must implement the desired boundary behavior; see text. \\
\hline
\end{array}\]

Table 8.1: Hydrodynamical boundary conditions supported by FLASH. Boundary type \(ab\) may be replaced with \(a=\{x,y,z\}\) for direction and \(b=\{l,r\}\) for left/right edge. All boundary types listed except the last (user) have an implementation in GridBoundaryConditions.

To use any of the hydrostatic-f2* boundary conditions, the setup must include Grid/GridBoundaryConditions/Flash2HSE. This must usually be explicitly requested, for example with a line

\texttt{REQUIRES Grid/GridBoundaryConditions/Flash2HSE}

in the simulation directory’s Config file.

Note that the grav_boundary_type runtime parameter is used by some implementations of the Gravity unit to define the type of boundary for solving a self-gravity (Poisson) problem; see Gravity_init. This runtime parameter is separate from the ab_boundary_type ones interpreted by GridBoundaryConditions, and its recognized values are not the same (although there is some overlap).

8.4.2 Boundary Conditions at Obstacles

The initial coarse grid of root blocks can be modified by removing certain blocks. This is done by providing a non-trivial implementation of Simulation_defineDomain. Effectively this creates additional domain boundaries at the interface between blocks removed and regions still included. All boundary conditions other than
8.4. BOUNDARY CONDITIONS

<table>
<thead>
<tr>
<th>ab_boundary_type</th>
<th>Constant</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>isolated</td>
<td>—</td>
<td>used by Gravity only for grav_boundary_type</td>
</tr>
<tr>
<td>—</td>
<td>DIRICHLET</td>
<td>used for multigrid solver</td>
</tr>
<tr>
<td>—</td>
<td>GRIDBC_MG_EXTRAPOLATE</td>
<td>for use by multigrid solver</td>
</tr>
<tr>
<td>—</td>
<td>PNEUMANN</td>
<td>(for use by multigrid solver)</td>
</tr>
<tr>
<td>hydrostatic</td>
<td>HYDROSTATIC</td>
<td>Hydrostatic, other implementation than FLASH2</td>
</tr>
<tr>
<td>hydrostatic+nvrefl</td>
<td>HYDROSTATIC_NVREFL</td>
<td>Hydrostatic variant, other impl. than FLASH2</td>
</tr>
<tr>
<td>hydrostatic+nvout</td>
<td>HYDROSTATIC_NVOUT</td>
<td>Hydrostatic variant, other impl. than FLASH2</td>
</tr>
<tr>
<td>hydrostatic+nvdiode</td>
<td>HYDROSTATIC_NVDIODE</td>
<td>Hydrostatic variant, other impl. than FLASH2</td>
</tr>
</tbody>
</table>

Table 8.2: Additional boundary condition types recognized by FLASH. Boundary type ab may be replaced with a={x,y,z} for direction and b={l,r} for left/right edge. These boundary types are either reserved for implementation by users and/or future FLASH versions for a specific purpose (as indicate by the remarks), or are for special uses within the Grid unit.

periodic are possible at these additional boundaries, and are handled there in the same way as on external domain boundaries. This feature is only available with PARAMESH. See the documentation and example in Simulation defineDomain for more details and some caveats.

8.4.3 Implementing Boundary Conditions

Users may need to implement boundary conditions beyond those provided with FLASH3, and the Grid-BoundaryConditions subunit provides several ways to achieve this. Users can provide an implementation for the user boundary type; or can provide or override an implementation for one of the other recognized types.

The simple boundary condition types reflect, outflow, diode are implemented in the Grid_bcApplyToRegion.F90 file in Grid/GridBoundaryConditions. A users can add or modify the handling of some boundary condition types in a version of this file in the simulation directory, which overrides the regular version. There is, however, also the interface Grid_bcApplyToRegionSpecialized which by default is only provided as a stub and is explicitly intended to be implemented by users. A Grid_bcApplyToRegionSpecialized implementation gets called before Grid_bcApplyToRegion and can decide to either handle a specific combination of boundary condition type, direction, grid data structure, etc., or leave it to Grid_bcApplyToRegion. These calls operate on a region of one block’s cells at a time. FLASH will pass additional information beyond that needed for handling simple boundary conditions to Grid_bcApplyToRegionSpecialized, in particular a block handle through which an implementation can retrieve coordinate information and access other information associated with a block and its cells.

The GridBoundaryConditions subunit also provides a simpler kind of interface if one includes Grid/GridBoundaryConditions/OneRow in the setup. When using this style of interface, users can implement guard cell filling one row at a time. FLASH passes to the implementation one row of cells at a time, some of which are interior cells while the others represent guard cells outside the boundary that are to be modified in the call. A row here means a contiguous set of cells along a line perpendicular to the boundary surface. There are two versions of this interface: Grid_applyBCEdge is given only one fluid variable at a time, but can also handle data structures other than unk; whereas Grid_applyBCEdgeAllUnkVars handles all variables of unk along a row in one call. Cell coordinate information is included in the call arguments. FLASH invokes these functions through an implementation of Grid_bcApplyToRegionSpecialized in Grid/GridBoundaryConditions/OneRow which acts as a wrapper. GridBoundaryConditions/OneRow also provides a default implementation of Grid_applyBCEdge (which implements the simple boundary conditions) and Grid_applyBCEdgeAllUnkVars (which calls Grid_applyBCEdge) each. Another implementation of Grid_applyBCEdgeAllUnkVars can be found in GridBoundaryConditions/OneRow/Flash2HSE, this one calls Grid_applyBCEdge or, for FLASH2-type hydrostatic boundaries, the code for handling them. These can be used as templates for overriding implementations under Simulation. It is not recommended to try to mix both Grid_bcApplyToRegion*-style and Grid_applyBCEdge*-style overriding implementations in a simulation directory, since this could become confusing.
Note that in all of these cases, i.e., whether boundary guard cell filling for a boundary type is implemented in `Grid_bcApplyToRegion`, `Grid_bcApplyToRegionSpecialized`, `Grid_applyBCEdge`, or `Grid_applyBCEdgeAllUnkVars`, the implementation does not fill guard cells in permanent data storage (the `unk` array and similar data structures) directly, but operates on buffers. FLASH3 fills some parts of the buffers with current values for interior cells before the call, and copies updated guardcell data from some (other) parts of the buffers back into `unk` (or similar) storage after the handling routine returns.

All calls to handlers for boundary conditions are for one face in a given dimension at a time. Thus for each of the `IAXIS`, `JAXIS`, and `KAXIS` dimensions there can be up to two series of calls, once for the left, i.e., “LOW,” and once for the right, i.e., “HIGH,” face. `PARAMESH 4` makes additional calls for filling guard cells in edge and corner regions of blocks, these calls result in additional `Grid_bcApplyToRegion` invocations for those cells that lie in diagonal directions from the block interior.

The boundary condition handling interfaces described so far can be implemented (and will be used!) independent of the `Grid` implementation chosen. At a lower level, the various implementations of `GridMain` have different ways of requesting that boundary guard cells be filled. The `GridBoundaryConditions` subunit collaborates with `GridMain` implementations to provide to user code uniform interfaces that are agnostic of lower-level details. However, it is also possible — but not recommended — for users to replace a routine that is located deeper in the `Grid` unit. For `PARAMESH 4`, the most relevant routine is `amr_1blk_bcset.F90`, for `PARAMESH 2` it is `tot_bnd.F90`, and for uniform grid `UG` it is `gr_bcApplyToAllBlks.F90`.

### 8.4.3.1 Additional Concerns with `PARAMESH 4`

Boundary condition handling has become significantly more complex in FLASH3. In part this is so because `PARAMESH 4` imposes requirements on guard cell filling code that do not exist in the other `GridMain` implementations:

1. In other `Grid` implementations, filling of domain boundary guard cells is under control of the “user” (in this context, the user of the grid implementation, i.e., FLASH): These cells can be filled for all blocks at a time that is predictable to the user code, as a standard part of handling `Grid_fillGuardCells`, only. With `PARAMESH 4`, the user-provided `amr_1blk_bcset` routine can be called from within the depths of `PARAMESH` on individual blocks (and cell regions, see below) during guard cell filling and at other times when the user has called a `PARAMESH` routine. It is not easy to predict when and in which sequence this will happen.

2. `PARAMESH 4` does not want all boundary guard cells filled in one call, but requests individual regions in various calls.

3. `PARAMESH 4` does not let the user routine `amr_1blk_bcset` operate on permanent storage (`unk` etc.) directly, but on (regions of) one-block buffers.

4. `PARAMESH 4` occasionally invokes `amr_1blk_bcset` to operate on regions of a block that belongs to a remote processor (and for which data has been cached locally). Such block data is not associated with a valid local `blockID`, making it more complicated for user code to retrieve metadata that may be needed to implement the desired boundary handling.

Some consequences of this for FLASH3 users:

- User code that implements boundary conditions for the grid inherits the requirement that it must be ready to be called at various times (when certain `Grid` routines are called).
- User code that implements boundary conditions for the grid inherits the requirement that it must operate on a region of the cells of a block, where the region is specified by the caller.
- Such user code must not assume that it operates on permanent data (in `unk` etc.). Rather, it must be prepared to fill guardcells for a block-shaped buffer that may or may not end up being copied back to permanent storage.

User code also is not allowed to make certain `PARAMESH 4` calls while a call to `amr_1blk_bcset` is active, namely those that would modify the same one-block buffers that the current call is working on.
• The user code must not assume that the block data it is acting on belongs to a local block. The data may not have a valid blockID. The code will be passed a “block handle” which can be used in some ways, but not all, like a valid blockID.

Caveat Block Handles

See the README file in Grid/GridBoundaryConditions for more information on how a block handle can be used.

8.5 Uniform Grid

The Uniform Grid has the same resolution in all the blocks throughout the domain, and each processor has exactly one block. The uniform grid can operate in either of two modes: fixed block size (FIXEDBLOCKSIZE) mode, and non-fixed block size (NONFIXEDBLOCKSIZE) mode. The default fixed block size grid is statically defined at compile time and can therefore take advantage of compile-time optimizations. The non-fixed block size version uses dynamic memory allocation of grid variables.

8.5.1 FIXEDBLOCKSIZE Mode

FIXEDBLOCKSIZE mode, also called static mode, is the default for the uniform grid. In this mode, the block size is specified at compile time as \(NXB \times NYB \times NZB\). These variables default to 8 if the dimension is defined and 1 otherwise – e.g.for a two-dimensional simulation, the defaults are \(NXB = 8, NYB = 8, NZB = 1\). To change the static dimensions, specify the desired values on the command line of the setup script; for example

\[./setup Sod -auto -3d -nxb=12 -nyb=12 -nzb=4 +ug\]

The distribution of processors along the three dimensions is given at run time as \(iprocs \times jprocs \times kprocs\) with the constraint that this product must be equal to the number of processors that the simulation is using. The global domain size in terms of number of grid points is \(NXB \times iprocs \times NYB \times jprocs \times NZB \times kprocs\). For example, if \(iprocs = jprocs = 4\) and \(kprocs = 1\), the execution command should specify \(np = 16\) processors.

\[mpirun -np 16 flash3\]

When working in static mode, the simulation is constrained to run on the same number of processors when restarting, since any different configuration of processors would change the domain size.

At Grid initialization time, the domain is created and the communication machinery is also generated. This initialization includes MPI communicators and datatypes for directional guardcell exchanges. If we view processors as arranged in a three-dimensional processor grid, then a row of processors along each dimension becomes a part of the same communicator. We also define MPI datatypes for each of these communicators, which describe the layout of the block on the processor to MPI. The communicators and datatypes, once generated, persist for the entire run of the application. Thus the \(MPI_{SEND/RECV}\) function with specific communicator and its corresponding datatype is able to carry out all data exchange for guardcell fill in the selected direction in a single step.

Since all blocks exist at the same resolution in the Uniform Grid, there is no need for interpolation while filling the guardcells. Simple exchange of correct data between processors, and the application of boundary conditions where needed is sufficient. The guard cells along the face of a block are filled with the layers of the interior cells of the block on the neighboring processor if that face is shared with another block, or calculated based upon the boundary conditions if the face is on the physical domain boundary. Also, because there are no jumps in refinement in the Uniform Grid, the flux conservation step across processor boundaries is unnecessary. For correct functioning of the Uniform Grid in FLASH, this conservation step should be explicitly turned off with a runtime parameter \(flux\_correct\) which controls whether or not to run the flux conservation step in the PPM Hydrodynamics implementation. AMR sets it by default to true, while UG sets it to false. Users should exercise care if they wish to override the defaults via their “flash.par” file.
8.5.2 NONFIXEDBLOCKSIZE mode

Up to version 2, FLASH always ran in a mode where all blocks have exactly the same number of grid points in exactly the same shape, and these were fixed at compile time. FLASH was limited to use the fixed block size mode described above. With FLASH3 this constraint was eliminated through an option at setup time. The two main reasons for this development were: one, to allow a uniform grid based simulation to be able to restart with different number of processors, and two, to open up the possibility of using other AMR packages with FLASH. Patch-based packages typically have different-sized block configurations at different times. This mode, called the “NONFIXEDBLOCKSIZE” mode, can currently be selected for use with Uniform Grid, and is the default mode with Chombo. To run an application in “NONFIXEDBLOCKSIZE” mode, the “-nofbs” option must be used when invoking the setup tool; see Chapter 5 for more information. For example:

```
./setup Sod -3d -auto -nofbs
```

Note that NONFIXEDBLOCKSIZE mode requires the use of its own IO implementation, and a convenient shortcut has been provided to ensure that this mode is used as in the example below:

```
./setup Sod -3d -auto +nofbs
```

In this mode, the blocksize in UG is determined at execution from runtime parameters iGridSize, jGridSize and kGridSize. These parameters specify the global number of grid points in the computational domain along each dimension. The blocksize then is \((i\text{GridSize}/i\text{procs}) \times (j\text{GridSize}/j\text{procs}) \times (k\text{GridSize}/k\text{procs})\).

Unlike FIXEDBLOCKSIZE mode, where memory is allocated at compile time, in the NONFIXEDBLOCKSIZE mode all memory allocation is dynamic. The global data structures are allocated when the simulation initializes and deallocated when the simulation finalizes, whereas the local scratch space is allocated and deallocated every time a unit is invoked in the simulation. Clearly there is a trade-off between flexibility and performance as the NONFIXEDBLOCKSIZE mode typically runs about 10-15% slower. We support both to give choice to the users. The amount of memory consumed by the Grid data structure of the Uniform Grid is \(n\text{var} \times (2 \times n\text{guard} + n\text{xb}) \times (2 \times n\text{guard} + n\text{yb}) \times (2 \times n\text{guard} + n\text{zb})\) irrespective of the mode. Note that this is not the total amount of memory used by the code, since fluxes, temporary variables, coordinate information and scratch space also consume a large amount of memory.

The example shown below gives two possible ways to define parameters in flash.par for a 3d problem of global domain size 64 \(\times\) 64 \(\times\) 64, being run on 8 processors.

```
iprocs = 2
jprocs = 2
kprocs = 2
iGridSize = 64
jGridSize = 64
kGridSize = 64
```

This specification will result in each processor getting a block of size 32 \(\times\) 32 \(\times\) 32. Now consider the following specification for the number of processors along each dimension, keeping the global domain size the same.

```
iprocs = 4
jprocs = 2
kprocs = 1
```

In this case, each processor will now have blocks of size 16 \(\times\) 32 \(\times\) 64.

8.6 Adaptive Mesh Refinement (AMR) Grid with Paramesh

The default package in FLASH is PARAMESH (MacNeice et al. 1999) for implementing the adaptive mesh refinement (AMR) grid. PARAMESH uses a block-structured adaptive mesh refinement scheme similar to others in the literature (e.g., Parashar 1999; Berger & Oliger 1984; Berger & Colella 1989; DeZeeuw & Powell 1993).
8.6. ADAPTIVE MESH REFINEMENT (AMR) GRID WITH PARAMESH

Figure 8.5: A simple computational domain showing varying levels of refinement in a total of 16 blocks. The dotted lines outline the guard cells for the block marked with a circle.

It also shares ideas with schemes which refine on an individual cell basis (Khokhlov 1997). In block-structured AMR, the fundamental data structure is a block of cells arranged in a logically Cartesian fashion. “Logically Cartesian” implies that each cell can be specified using a block identifier (processor number and local block number) and a coordinate triple \((i, j, k)\), where \(i = 1 \ldots \text{nxb}\), \(j = 1 \ldots \text{nyb}\), and \(k = 1 \ldots \text{nzr}\) refer to the \(x\)-, \(y\)-, and \(z\)-directions, respectively. It does not require a physically rectangular coordinate system; for example a spherical grid can be indexed in this same manner.

The complete computational grid consists of a collection of blocks with different physical cell sizes, which are related to each other in a hierarchical fashion using a tree data structure. The blocks at the root of the tree have the largest cells, while their children have smaller cells and are said to be refined. Three rules govern the establishment of refined child blocks in PARAMESH. First, a refined child block must be one-half as large as its parent block in each spatial dimension. Second, a block’s children must be nested; i.e., the child blocks must fit within their parent block and cannot overlap one another, and the complete set of children of a block must fill its volume. Thus, in \(d\) dimensions a given block has either zero or \(2^d\) children. Third, blocks which share a common border may not differ from each other by more than one level of refinement.

A simple two-dimensional domain is shown in Figure 8.5, illustrating the rules above. Each block contains \(\text{nxb} \times \text{nyb} \times \text{nzrb}\) interior cells and a set of guard cells. The guard cells contain boundary information needed to update the interior cells. These can be obtained from physically neighboring blocks, externally specified boundary conditions, or both. The number of guard cells needed depends upon the interpolation schemes and the differing stencils used by the various physics units (usually hydrodynamics). For the explicit PPM algorithm distributed with FLASH, four guard cells are needed in each direction, as illustrated in Figure 8.4. The blocksize while using the adaptive grid is fixed at compile time, resulting in static memory allocation. The total number of blocks a processor can manage is determined by \text{MAXBLOCKS}, which can be overridden at setup time with the \text{setup \ldots -maxblocks=#} argument. The amount of memory consumed by the Grid data structure of code when running with PARAMESH is \(\text{NUNK_VARS} \times (2 \ast \text{nguard} + \text{nxb}) \times (2 \ast \text{nguard} + \text{nyb}) \times (2 \ast \text{nguard} + \text{nzr}) \times \text{MAXBLOCKS}\). PARAMESH also needs memory to store its tree data structure for adaptive mesh management, over and above what is already mentioned with Uniform Grid. As the levels of refinement increase, the size of the tree also grows.

PARAMESH handles the filling of guard cells with information from other blocks or, at the boundaries of the physical domain, from an external boundary routine (see Section 8.4). If a block’s neighbor exists and has the same level of refinement, PARAMESH fills the corresponding guard cells using a direct copy.
Figure 8.6: Flux conservation at a jump in refinement. The fluxes in the fine cells are added and replace the coarse cell flux (F).

from the neighbor’s interior cells. If the neighbor has a different level of refinement, the data from the neighbor’s cells must be adjusted by either interpolation (to a finer level of resolution) or averaging (to a coarser level)—see Section 8.6.2 below for more information. If the block and its neighbor are stored in the memory of different processors, PARAMESH handles the appropriate parallel communications (blocks are never split between processors). The filling of guard cells is a global operation that is triggered by calling Grid_fillGuardCells.

Grid Interpolation is also used when filling the blocks of children newly created in the course of automatic refinement. This happens during Grid_updateRefinement processing. Averaging is also used to regularly update the solution data in at least one level of parent blocks in the oct-tree. This ensures that after leaf nodes are removed during automatic refinement processing (in regions of the domain where the mesh is becoming coarser), the new leaf nodes automatically have valid data. This averaging happens as an initial step in Grid_fillGuardCells processing.

PARAMESH also enforces flux conservation at jumps in refinement, as described by Berger and Colella (1989). At jumps in refinement, the fluxes of mass, momentum, energy (total and internal), and species density in the fine cells across boundary cell faces are summed and passed to their parent. The parent’s neighboring cell will be at the same level of refinement as the summed flux cell because PARAMESH limits the jumps in refinement to one level between blocks. The flux in the parent that was computed by the more accurate fine cells is taken as the correct flux through the interface and is passed to the corresponding coarse face on the neighboring block (see Figure 8.6). The summing allows a geometrical weighting to be implemented for non-Cartesian geometries, which ensures that the proper volume-corrected flux is computed.

8.6.1 Additional Data Structures

PARAMESH maintains much additional information about the mesh. In particular, oct-tree related information is kept in various arrays which are declared in a F90 module called "tree". It includes the physical coordinate of a block’s center, its physical size, level of refinement, and much more. These data structures also acts as temporary storage while updating refinement in the grid and moving the blocks. This metadata should in general not be accessed directly by application code. The Grid API contains subroutines for accessing the most important pars of this metadata on a block by block basis, like Grid_getBlkBoundBox, Grid_getBlkCenterCoords, Grid_getBlkPhysicalSize, Grid_getBlkRefineLevel, and Grid_getBlkTyp.

FLASH has its own oneBlock data structure that stores block specific information. This data structure keeps the physical coordinates of each cell in the block. For each dimension, the coordinates are stored for the LEFT_EDGE, the RIGHT_EDGE and the center of the cell. The coordinates are determined from “cornerID"
8.6. ADAPTIVE MESH REFINEMENT (AMR) GRID WITH PARAMESH

which is also a part of this data structure.

The concept of cornerID was introduced in FLASH3; it serves three purposes. First, it creates a unique global identity for every cell that can come into existence at any time in the course of the simulation. Second, it can prevent machine precision error from creeping into the spatial coordinates calculation. Finally, it can help pinpoint the location of a block within the oct-tree of PARAMESH. Another useful integer variable is the concept of a stride. A stride indicates the spacing factor between one cell and the cell directly to its right when calculating the cornerID. At the maximum refinement level, the stride is 1, at the next higher level it is 2, and so on. Two consecutive cells at refinement level \( n \) are numbered with a stride of \( 2^{l_{\text{refine}}_{\text{max}}-n} \) where \( 1 \leq n \leq l_{\text{refine}}_{\text{max}} \).

The routine Grid.getBlkCornerID provides a convenient way for the user to retrieve the location of a block or cell. A usage example is provided in the documentation for that routine. The user should retrieve accurate physical and grid coordinates by calling the routines Grid.getBlkCornerID, Grid.getCellCoords, Grid.getBlkCenterCoords and Grid.getBlkPhysicalSize, instead of calculating their own from local block information, since they take advantage of the cornerID scheme, and therefore avoid the possibility of introducing machine precision induced numerical drift in the calculations.

8.6.2 Grid Interpolation (and Averaging)

The adaptive grid requires data interpolation or averaging when the refinement level (i.e., mesh resolution) changes in space or in time. If during guardcell filling a block’s neighbor has a coarser level of refinement, the neighbor’s cells are used to interpolate guard cell values to the cells of the finer block. Interpolation is also used when filling the blocks of children newly created in the course of automatic refinement. Data averaging is used to adapt data in the opposite direction, i.e., from fine to coarse.

In the AMR context, the term prolongation is used to refer to data interpolation (because it is used when the tree of blocks grows longer). Similarly, the term restriction is used to refer to fine-to-coarse data averaging.

The algorithm used for restriction is straightforward (equal-weight) averaging in Cartesian coordinates, but has to take cell volume factors into account for curvilinear coordinates; see Section 8.11.5.

PARAMESH supports various interpolation schemes, to which user-specified interpolation schemes can be added. FLASH4 currently allows to choose between two interpolation schemes:

1. monotonic
2. native

The choice is made at setup time, see Section 5.2.

The versions of PARAMESH supplied with FLASH4 supply their own default interpolation scheme, which is used when FLASH4 is configured with the -gridinterpolation=native setup option (see Section 5.2). The default schemes are only appropriate for Cartesian coordinates. If FLASH4 is configured with curvilinear support, an alternative scheme (appropriate for all supported geometries) is compiled in. This so-called “monotonic” interpolation attempts to ensure that interpolation does not introduce small-scale non-monotoncity in the data. The order of “monotonic” interpolation can be chosen with the interpolorder runtime parameter. See Section 8.11.5 for some more details on prolongation for curvilinear coordinates. At setup time, monotonic interpolation is the default interpolation used.

8.6.2.1 Interpolation for mass-specific solution variables

To accurately preserve the total amount of conserved quantities, the interpolation routines have to be applied to solution data in conserved, i.e., volume-specific, form. However, many variables are usually stored in the unk array in mass-specific form, e.g., specific internal and total energies, velocities, and mass fractions. See Section 5.5.1 for how to use the optional TYPE attribute in a Config file’s VARIABLE definitions to inform the Grid unit which variables are considered mass-specific.

FLASH4 provides three ways to deal with this:

---

3Particles and Physics units may make additional use of interpolation as part of their function, and the algorithms may or may not be different. This subsection only discusses interpolation automatically performed by the Grid unit on the fluid variables in a way that should be transparent to other units.
1. Do nothing—i.e., assume that ignoring the difference between mass-specific and conserved form is a reasonable approximation. Depending on the smoothness of solutions in regions where refinement, derefinement, and jumps in refinement level occur, this assumption may be acceptable. This behavior can be forced by setting the `convertToConservedInMeshInterp` runtime parameter to `.false.`

2. Convert mass-specific variables to conserved form in all blocks throughout the physical domain before invoking a Grid function that may result in some data interpolation or restriction (refinement, derefinement, guardcell filling); and convert back after these functions return. Conversion is done by cell-by-cell multiplication with the density (i.e., the value of the “dens” variable, which should be declared as

   \[
   \text{VARIABLE dens TYPE: PER_VOLUME}
   \]

   in a Config file).

   This behavior is available in both PARAMESH 2 and PARAMESH 4. It is enabled by setting the `convertToConservedForMeshCalls` runtime parameter and corresponds roughly to FLASH2 with `conserved_var` enabled.

3. Convert mass-specific variables to conserved form only where and when necessary, from the Grid user’s point of view as part of data interpolation. Again, conversion is done by cell-by-cell multiplication with the value of density. In the actual implementation of this approach, the conversion and back-conversion operations are closely bracketing the interpolation (or restriction) calls. The implementation avoids spurious back-and-forth conversions (i.e., repeated successive multiplications and divisions of data by the density) in blocks that should not be modified by interpolation or restriction.

   This behavior is available only for PARAMESH 4. As of FLASH4, this is the default behavior whenever available. It can be enabled explicitly (only necessary in setups that change the default) by setting the `convertToConservedInMeshInterp` runtime parameter.

8.6.3 Refinement

8.6.3.1 Refinement Criteria

The refinement criterion used by PARAMESH is adapted from Löhner (1987). Löhner’s error estimator was originally developed for finite element applications and has the advantage that it uses a mostly local calculation. Furthermore, the estimator is dimensionless and can be applied with complete generality to any of the field variables of the simulation or any combination of them.

FLASH Transition

FLASH4 does not define any refinement variables by default. Therefore simulations using AMR have to make the appropriate runtime parameter definitions in `flash.par`, or in the simulation’s Config file. If this is not done, the program generates a warning at startup, and no automatic refinement will be performed. The mistake of not specifying refinement variables is thus easily detected. To define a refinement variable, use `refine_var_#` (where # stands for a number from 1 to 4) in the `flash.par` file.

Löhner’s estimator is a modified second derivative, normalized by the average of the gradient over one computational cell. In one dimension on a uniform mesh, it is given by

\[
E_i = \frac{1}{u_{i+1} - u_i + \epsilon} \left( \frac{|u_{i+1} - 2u_i + u_{i-1}|}{|u_{i+1} - u_i| + |u_i - u_{i-1}| + \epsilon |u_{i+1}| + 2 |u_i| + |u_{i-1}|} \right), \tag{8.1}
\]

where \( u_i \) is the refinement test variable’s value in the \( i \)-th cell. The last term in the denominator of this expression acts as a filter, preventing refinement of small ripples, where \( \epsilon \) should be a small constant.
When extending this criterion to multidimensions, all cross derivatives are computed, and the following generalization of the above expression is used

\[
E_{i_{1}i_{2}i_{3}} = \left\{ \sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right)^2 \right\}^{1/2},
\]

where the sums are carried out over coordinate directions, and where, unless otherwise noted, partial derivatives are evaluated at the center of the \(i_{1}i_{2}i_{3}\)-th cell.

The estimator actually used in FLASH4’s default refinement criterion is a modification of the above, as follows:

\[
E_i = \left| \frac{u_{i+2} - 2u_i + u_{i-2}}{\Delta x^2} \right| + \left| u_{i} - u_{i-2} \right| + \epsilon \left( \frac{u_{i+2} + u_{i-2}}{\Delta x^2} + \left| u_{i} \right| + \left| u_{i-2} \right| \right),
\]

where again \(u_i\) is the refinement test variable’s value in the \(i\)-th cell. The last term in the denominator of this expression acts as a filter, preventing refinement of small ripples, where \(\epsilon\) is a small constant.

When extending this criterion to multidimensions, all cross derivatives are computed, and the following generalization of the above expression is used

\[
E_{i_{X}i_{Y}i_{Z}} = \left\{ \sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \right)^2 \right\}^{1/2},
\]

where again the sums are carried out over coordinate directions, where, unless otherwise noted, partial derivatives are actually finite-difference approximations evaluated at the center of the \(i_{X}i_{Y}i_{Z}\)-th cell, and \(|u_{pq}|\) stands for an average of the values of \(|u|\) over several neighboring cells in the \(p\) and \(q\) directions.

The constant \(\epsilon\) is by default given a value of \(10^{-2}\), and can be overridden through the runtime parameters. Blocks are marked for refinement when the value of \(E_{i_{X}i_{Y}i_{Z}}\) for any of the block’s cells exceeds a threshold given by the runtime parameters \(\text{refine.cutoff.}\#\), where the number \# matching the number of the runtime parameters \(\text{refine.var.}\#\) runtime parameter selecting the refinement variable. Similarly, blocks are marked for derefinement when the values of \(E_{i_{X}i_{Y}i_{Z}}\) for all of the block’s cells lie below another threshold given by the runtime parameters \(\text{derefine.cutoff.}\#\).

Although PPM is formally second-order and its leading error terms scale as the third derivative, we have found the second derivative criterion to be very good at detecting discontinuities and sharp features in the flow variable \(u\). When \(\text{Particles}\) (active or tracer) are being used in a simulation, their count in a block can also be used as a refinement criterion by setting \(\text{refine.on.particle.count}\) to true and setting \(\text{max.particles.per.blk}\) to the desired count.

### 8.6.3.2 Refinement Processing

Each processor decides when to refine or derefine its blocks by computing a user-defined error estimator for each block. Refinement involves creation of either zero or \(2^d\) refined child blocks, while derefinement involves deletion of all of a parent’s child blocks (\(2^d\) blocks). As child blocks are created, they are temporarily placed at the end of the processor’s block list. After the refinements and derefinements are complete, the blocks are redistributed among the processors using a work-weighted Morton space-filling curve in a manner similar to that described by Warren and Salmon (1987) for a parallel treecode. An example of a Morton curve is shown in Figure 8.7.

During the distribution step, each block is assigned a weight which estimates the relative amount of time required to update the block. The Morton number of the block is then computed by interleaving the bits of its integer coordinates, as described by Warren and Salmon (1987). This reordering determines its location.
Figure 8.7: Morton space-filling curve for adaptive mesh grids.

along the space-filling curve. Finally, the list of all blocks is partitioned among the processors using the block weights, equalizing the estimated workload of each processor. By default, all leaf-blocks are weighted twice as heavily as all other blocks in the simulation.

8.6.3.3 Specialized Refinement Routines

Sometimes, it may be desirable to refine a particular region of the grid independent of the second derivative of the variables. This criterion might be, for example, to better resolve the flow at the boundaries of the domain, to refine a region where there is vigorous nuclear burning, or to better resolve some smooth initial condition. For curvilinear coordinates, regions around the coordinate origin or the polar z-axis may require special consideration for refinement. A collection of methods that can refine a (logically) rectangular region or a circular region in Cartesian coordinates, or can automatically refine by using some variable threshold, are available through the Grid_markRefineSpecialized routine. It is intended to be called from the Grid_markRefineDerefine routine. The interface works by allowing the calling routine to pick one of the routines in the suite through an integer argument. The calling routine is also expected to populate the data structure specs before making the call. A copy of the file Grid_markRefineDerefine.F90 should be placed in the Simulation directory, and the interface file Grid_interface.F90 should be used in the header of the function.

Warning

This collection of specialized refinement routines have had limited testing. The routine that refines in a specified region has been tested with some of the setups included in the release. All the other routines should be used mostly as guidelines for the user’s code.

FLASH3.2 added additional support in the standard implementation of refinement routine Grid_markRefineDerefine for enforcing a maximum on refinement based on location or simulation time. These work by effectively lowering the absolute ceiling on refinement level represented by lrefine_max. See the following runtime parameters:

- gr_lrefineMaxRedDoByLogR
8.7 Chombo

We have included an experimental Grid implementation which makes use of Chombo library in FLASH4. Since this is very much a work in progress it should not be considered as production grade yet. We expect new developments and improvement to happen at a rapid pace, so please contact us if you would like to use the latest snapshots of our Chombo integration in FLASH before our next major code release.

Most flash.par parameters relevant to the Grid unit will continue to work; the differences are described later in this section. There are some important restrictions to be aware of when using this Grid implementation.

- Only Cartesian geometries are handled.
- Only cell-centered solution variables are supported, i.e. those variables defined with keywords VARIABLE and SCRATCHCENTERVAR in Config files.

FLASH does not include the source code for Chombo library and so Chombo must be independently downloaded and then built using the instructions given in Section 5.9. Please also see the restrictions in Hydro Section 14.1.5 and I/O Section 9.12.

Our integration of FLASH with Chombo is slightly unusual because we do not use the higher-level Chombo classes for solving time-dependent problems on an AMR mesh. The reason is that these higher level classes also ”drive” the simulation, that is, control the initialization and evolution of simulations. These are tasks that already happen in a FLASH application in the Driver unit. Our approach is therefore to use the lower-level Chombo classes so that Chombo library is only responsible for Grid related tasks.

The actual patch-based mesh is allocated and managed in Chombo library. Each individual patch is stored in contiguous memory locations with the slowest varying dimension being the solution variable. This has allowed us to create a Grid_getBlkPtr that builds a 4D Fortran pointer object from the memory address of the first element of a specified patch. The patch is treated as a standard FLASH block in the physics units of FLASH. The Grid unit obtains the memory address and other metadata such as patch size, coordinates, index limits, cell spacing and refinement level through non-decorated C++ function calls which query Chombo objects.

8.7.1 Using Chombo in a UG configuration

This is a Grid implementation that operates in the same way as the standard FLASH uniform grid mode in which there is one block per MPI process. The parameters accepted are identical to those described for a non fixed blocksize uniform grid in Section 8.5.2. The only difference is that the checkpoint file is written in Chombo file layout rather than FLASH file layout. Setup a simulation using:

```
./setup Sedov -auto +chombo_ug -parfile=test_chombo_ug_2d.par
```

8.7.2 Using Chombo in a AMR configuration

The real motivation for integrating Chombo into FLASH is for its adaptive mesh which is patch-based rather than an oct-tree like Paramesh. It has several desirable features including:

- refinement can happen cell-by-cell rather than block-by-block.
blocks (or patches in Chombo terminology) are not limited to a fixed number of cells.

refinement jumps of greater than 2 can happen between neighboring blocks.

A major difference in the generated meshes is that there is no longer the concept of “LEAF” blocks which cover the global domain. This was useful in Paramesh because a union of the space covered by the children of a block cover is exactly equal to the space covered by the parent block. It is therefore possible to evolve solution only on leaf blocks. Such complete overlap is not guaranteed in Chombo, in fact it rarely happens. Therefore, to be compatible with Chombo, physics units must evolve a solution on all blocks. For reference, the LEAF definition in Flash.h for FLASH-Chombo application now means all blocks.

A FLASH application with Chombo uses the refinement criteria similar to the one described in Section 8.6.3. The slight modification is that the \( i_X i_Y i_Z \)-th cell is tagged when \( E_{i_X i_Y i_Z} \) exceeds a threshold value. Recall that in the Paramesh, the entire block is tagged for refinement when the error estimator of any of cells within a block exceeds a given threshold. As a further extension to this cell tagging procedure we also tag all cells within a small radius of a given tagged cell. The tagging radius is set using the runtime parameter \( \text{tagradius} \); a value of 1 means that 1 cell to the left and right of a tagged cell in each dimension is also tagged. Values below the default value of 2 may cause problems because of introduction of fine-coarse interfaces in regions of the domain where there are steep gradients. For example, a Sod problem does not converge in Riemann solver when the value is 0.

The runtime parameters are described below:

- \( \text{lrefine\_max} \) Used in an identical way to Paramesh. It is still a unit-based parameter and not zero-based. Defaults to 1.

- \( \text{igridsize}, \text{jgridsize}, \text{kgridsize} \) The global number of cells on the coarsest level of the adaptive mesh. Varying \( \text{igridsize} \) gives the same level of control as the unused Paramesh parameters \( \text{lrefine\_min} \) and \( \text{nblockx} \). Defaults to 16.

- \( \text{maxBlockSize} \) The maximum number of cells along a single dimension of a block. Defaults to 16.

- \( \text{BRMeshRefineBlockFactor} \) The minimum number of cells along a single dimension of a block. See BRMeshRefine class in Chombo user-guide. Defaults to 8.

- \( \text{BRMeshRefineFillRatio} \) Overall grid efficiency to be generated. If this number is set low, the grids will tend to be larger and less filled with tags. If this number is set high, the grids will tend to be smaller and more filled with tags. See BRMeshRefine class in Chombo user-guide. Defaults to 0.75.

- \( \text{BRMeshRefineBufferSize} \) Proper nesting buffer size. This will be the minimum number of level 1 cells between any level l+1 cell and a level l-1 cell. See BRMeshRefine class in Chombo user-guide. Defaults to 1.

- \( \text{verbosity} \) The verbosity of debugging output written to local log files - higher values mean progressively more debugging output. As a rough guide 0 means no debugging output, 2 means debugging output from FLASH objects, and values above 2 add debugging output from Chombo objects. Defaults to 0.

- \( \text{refRatio} \) The refinement jump between fine-coarse levels, typically 2, 4 or 8. Defaults to 2.

- \( \text{restrictBeforeGhostExchange} \) Invokes a full mesh restriction before guard cells are exchanged. This is to make the guard cell exchange behave the same way as Paramesh guard cell exchange. We do not know yet if it is necessary. Defaults to True.

- \( \text{tagRadius} \) The radius of cells around a tagged cell that should also be tagged. Defaults to 2.

Setup a simulation using:

\[ ./\text{setup Sedov -auto +chombo\_amr -parfile=test\_chombo\_amr\_2d.par} \]

The GridParticles and GridSolvers sub-unit do not interoperate with Chombo yet.
8.8 GridMain Usage

The Grid unit has the largest API of all units, since it is the custodian of the bulk of the simulation data, and is responsible for most of the code housekeeping. The Grid\_init routine, like all other Unit\_init routines, collects the runtime parameters needed by the unit and stores values in the data module. If using UG, the Grid\_init also creates the computational domain and the housekeeping data structures and initializes them. If using AMR, the computational domain is created by the Grid\_initDomain routine, which also makes a call to mesh package’s own initialization routine. The physical variables are all owned by the Grid unit, and it initializes them by calling the Simulation\_initBlock routine which applies the specified initial conditions to the domain. If using an adaptive grid, the initialization routine also goes through a few refinement iterations to bring the grid to desired initial resolution, and then applies the Eos function to bring all simulation variables to thermodynamic equilibrium. Even though the mesh-based variables are under Grid’s control, all the physics units can operate on and modify them.

A suite of get/put accessor/mutator functions allows the calling unit to fetch or send data by the block. One option is to get a pointer Grid\_getBlkPtr, which gives unrestricted access to the whole block and the client unit can modify the data as needed. The more conservative but slower option is to get some portion of the block data, make a local copy, operate on and modify the local copy and then send the data back through the “put” functions. The Grid interface allows the client units to fetch the whole block (Grid\_getBlkData), a partial or full plane from a block (Grid\_getPlaneData), a partial or full row (Grid\_getRowData), or a single point (Grid\_getPointData). Corresponding “put” functions allow the data to be sent back to the Grid unit after the calling routine has operated on it. Various getData functions can also be used to fetch some derived quantities such as the cell volume or face area of individual cells or groups of cells. There are several other accessor functions available to query the housekeeping information from the grid. For example Grid\_getListOfBlocks returns a list of blocks that meet the specified criterion such as being “LEAF” blocks in PARAMESH, or residing on the physical boundary.

**FLASH Transition**

The Grid\_getBlkData and Grid\_putBlkData functions replace the dBaseGetData and dBasePutData functions in FLASH2. The bulk of the dBase functionality from FLASH2 is now handled by the Grid unit. For example, the global mesh data structures “unk” and “tree” now belong to Grid, and any information about them is queried from it. However, dBase and Grid are not identical. dBase was a central storage for all data, whereas in FLASH4 some of the data, such as simulation parameters like dt and simulation time are owned by the Driver unit instead of the Grid unit.

**FLASH Transition**

In FLASH2, variables nxb, nyb and nzb traditionally described blocksize. From FLASH3 on, the symbols NXB, NYB, and NYB are intended not to be used directly except in the Grid unit itself. The variables like iHi\_gc and iHi etc. that marked the endpoints of the blocks were replaced in FLASH3 with in their new incarnation (GRID\_IHI\_GC etc.) Again, these new variables are used only in sizing arrays in declaration headers. Even when they are used for array sizing, they are enclosed in preprocessor blocks that can be eliminated at compile time. This separation was done to compartmentalize the FLASH3 code. The grid layout is not required to be known in any other unit (with the exception of IO). FLASH3 provides an API function Grid\_getBlkIndexLimits that fetches the block endpoints from the Grid unit for each individual block. The fetched values are then used as do loop end points in all \((i,j,k)\) loops in all the client units. When working in NONFIXEDBLOCKSIZE mode, the same fetched values are also used to size and allocate arrays. We have retained GRID\_IHI\_GC etc. for array sizing as compile time option for performance reasons. Statically allocated arrays allow better compiler optimization.
In addition to the functions to access the data, the Grid unit also provides a collection of routines that drive some housekeeping functions of the grid without explicitly fetching any data. A good example of such routines is Grid_fillGuardCells. Here no data transaction takes place between Grid and the calling unit. The calling unit simply instructs the Grid unit that it is ready for the guard cells to be updated, and doesn’t concern itself with the details. The Grid_fillGuardCells routine makes sure that all the blocks get the right data in their guardcells from their neighbors, whether they are at the same, lower or higher resolution, and if instructed by the calling routine, also ensures that EOS is applied to them.

In large-scale, highly parallel FLASH simulations with AMR, the processing of Grid_fillGuardCells calls may take up a significant part of available resource like CPU time, communication bandwidth, and buffer space. It can therefore be important to optimize these calls in particular. From FLASH3.1, Grid_fillGuardCells provides ways to

- operate on only a subset of the variables in unk (and facevarx, facevary, and facevarz), by masking out other variables;
- fill only some the nguard layers of guard cells that surround the interior of a block (while possibly excepting a “sweep” direction);
- combine guard cell filling with EOS calls (which often follow guard cell exchanges in the normal flow of execution of a simulation in order to ensure thermodynamical consistency in all cells, and which may also be very expensive), by letting Grid_fillGuardCells make the calls on cells where necessary;
- automatically determine masks and whether to call EOS, based on the set of variables that the calling code actually needs updated, by masking out other variables.

These options are controlled by OPTIONAL arguments, see Grid_fillGuardCells for documentation. When these optional arguments are absent or when a Grid implementation does not support them, FLASH falls back to safe default behavior which may, however, be needlessly expensive.

Another routine that may change the global state of the grid is Grid_updateRefinement. This function is called when the client unit wishes to update the grid’s resolution. Again, the calling unit does not need to know any of the details of the refinement process.

---

**FLASH Transition**

As mentioned in Chapter 4, FLASH allows every unit to identify scalar variables for checkpointing. In the Grid unit, the function that takes care of consolidating user specified checkpoint variable is Grid_sendOutputData. Users can select their own variables to checkpoint by including an implementation of this function specific to their requirements in their Simulation setup directory.

---

8.9 GridParticles

FLASH is primarily an Eulerian code, however, there is support for tracing the flow using Lagrangian particles. In FLASH4 we have generalized the interfaces in the Lagrangian framework of the Grid unit in such a way that it can also be used for miscellaneous non-Eulerian data such as tracing the path of a ray through the domain, or tracking the motion of solid bodies immersed in the fluid. FLASH also uses active particles with mass in cosmological simulations, and charged particles in a hybrid PIC solver. Each particle has an associated data structure, which contains fields such as its physical position and velocity, and relevant physical attributes such as mass or field values in active particles. Depending upon the time advance method, there may be other fields to store intermediate values. Also, depending upon the requirements of the simulation, other physical variables such as temperature etc. may be added to the data structure. The GridParticles subunit of the Grid unit has two sub-subunits of its own. The GridParticlesMove sub-subunit moves the data structures associated with individual particles when the particles move between blocks; the actual movement of the particles through time advancement is the responsibility of the Particles
8.9. GRIDPARTICLES

Figure 8.8: The Grid unit: structure of GridParticles subunit.

unit. Particles move from one block to another when their time advance places them outside their current block. In AMR, the particles can also change their block through the process of refinement and derefinement. The GridParticlesMap sub-subunit provides mapping between particles data and the mesh variables. The mesh variables are either cell-centered or face-centered, whereas a particle’s position could be anywhere in the cell. The GridParticlesMap sub-subunit calculates the particle’s properties at its position from the corresponding mesh variable values in the appropriate cell. When using active particles, this sub-subunit also maps the mass of the particles onto the specified mesh variable in appropriate cells. The next sections describe the algorithms for moving and mapping particles data.

8.9.1 GridParticlesMove

FLASH4 has implementations of three different parallel algorithms for moving the particles data when they are displaced from their current block. FLASH3 had an additional algorithm, Perfect Tree Level which made use of the oct-tree structure. However, because in all performance experiments it performed significantly worse than the other two algorithms, it is not supported currently in FLASH4. The simplest algorithm, Directional algorithm is applicable only to the uniform grid when it is configured with one block per processor. This algorithm uses directional movement of data, and is easy because the directional neighbors are trivially known. The movement of particles data is much more challenging with AMR even when the grid is not refining. Since the blocks are at various levels of refinement at any given moment, a block may have more than one neighbor along one or more of its faces. The distribution of blocks based on space-filling curve is an added complication since the neighboring blocks along a face may reside at a non-neighboring processor. The remaining two algorithmss included in FLASH4 implement GridParticlesMove subunit for the adaptive mesh; Point to Point and Sieve, of which only the Sieve algorithm is able to move the data when the mesh refines. Thus even when a user opts for the PointToPoint implementation for moving particles with time evolution, some part of the Sieve implementation must necessarily be included to successfully move the data upon refinement.
8.9.1.1 Directional Move

The Directional Move algorithm for moving particles in a Uniform Grid minimizes the number of communication steps instead of minimizing the volume of data moved. Its implementation has the following steps:

1. Scan particle positions. Place all particles with their $x$ coordinate value greater than the block bounding box in the Rightmove bin, and place those with $x$ coordinate less than block bounding box in Leftmove bin.

2. Exchange contents of Rightbin with the right block neighbor’s Leftbin contents, and those of the Leftbin with left neighbor’s Rightbin contents.

3. Merge newly arrived particles from step 2 with those which did not move outside their original block.

4. Repeat steps 1-3 for the y direction.

5. Repeat step 1-2 for the z direction.

At the end of these steps, all particles will have reached their destination blocks, including those that move to a neighbor on the corner. Figure 8.9 illustrates the steps in getting a particle to its correct destination.

8.9.1.2 Point To Point Algorithm

As a part of the data cached by Paramesh, there is wealth of information about the neighborhood of all the blocks on a processor. This information includes the processor and block number of all neighbors (face and corners) if they are at the same refinement level. If those neighbors are at lower refinement level, then the neighbor block is in reality the current block’s parent’s neighbor, and the parent’s neighborhood information is part of the cached data. Similarly, if the neighbor is at a higher resolution then the current blocks neighbor is in reality the parent of the neighbor. The parent’s metadata is also cached, from which information about all of its children can be derived. Thus it is possible to determine the processor and block number of the destination block for each particle. The PointToPoint implementation finds out the destinations for every particles that is getting displaced from its block. Particles going to local destination blocks are moved first. The remaining particles are sorted based on their destination processor number, followed by a couple of global operations that allow every processor to determine the number of particles it is expected to receive from all of the other processors. A processor then posts asynchronous receives for every source processor that had at least one particle to send to it. In the next step, the processor cycles through the sorted list of particles and sends them to the appropriate destinations using synchronous mode of communication.
8.9.1.3 Sieve Algorithm

The Sieve algorithm does not concern itself with the configuration of the underlying mesh at any time. It also does not distinguish between data movements due to time evolution or regridding, and is therefore the only usable implementation when the particles are displaced as a consequence of mesh refinement. The sieve implementation works with two bins, one collects particles that have to be moved off-processor, and the other receives particles sent to it by other processors. The following steps broadly describe the algorithm:

1. For each particle, find if its current position is on the current block
2. If not, find if its current position is on another block on the same processor. If it is move the particle to that block, otherwise put it in the send bin.
3. Send contents of the send bin to the designated neighbor, and receive contents of another neighbor’s send bin into my receive bin.
4. Repeat step 2 on the contents of the receive bin, and step 3 until all particles are at their destination.
5. For every instance of step 3, the designated send and receive neighbors are different from any of the previous steps.

In this implementation, the trick is to use an algorithm to determine neighbors in such a way that all the particles reach their destination in minimal number of hops. Using $MyPE + n * (-1)^{n+1}$ as the destination processor and $MyPE + n * (-1)^n$ as the source processor in modulo $numProcs$ arithmetic meets the requirements. Here $MyPE$ is the local processor number and $numProcs$ is the number of processors.

8.9.2 GridParticlesMapToMesh

FLASH4 provides support for particles that can experience forces and contribute to the problem dynamics. These are termed active particles, and are described in detail in Chapter 19. As these particles may move independently of fluid flow, it is necessary to update the grid by mapping an attribute of the particles to the cells of the grid. We use these routines, for example, during the PM method to assign the particles’ mass to the particle density solution variable $pden$. The hybrid PIC method uses its own variation for mapping appropriate physical quantities to the mesh.

In general the mapping is performed using the grid routines in the GridParticlesMapToMesh directory and the particle routines in the ParticlesMapping directory. Here, the particle subroutines map the particles’ attribute into a temporary array which is independent of the current state of the grid, and the grid subroutines accumulate the mapping from the array into valid cells of the computational domain. This means the grid subroutines accumulate the data according to the grid block layout, the block refinement details, and the simulation boundary conditions. As these details are closely tied with the underlying grid, there are separate implementations of the grid mapping routines for UG and PARAMESH simulations.

The implementations are required to communicate information in a relatively non-standard way. Generally, domain decomposition parallel programs do not write to the guard cells of a block, and only use the guard cells to represent a sufficient section of the domain for the current calculation. To repeat the calculation for the next time step, the values in the guard cells are refreshed by taking updated values from the internal section of the relevant block. In FLASH4, PARAMESH refreshes the values in the guard cells automatically, and when instructed by a Grid_fillGuardCells call.

In contrast, the guard cell values are mutable in the particle mapping problem. Here, it is possible that a portion of the particle’s attribute is accumulated in a guard cell which represents an internal cell of another block. This means the value in the updated guard cell must be communicated to the appropriate block. Unfortunately, the mechanism to communicate information in this direction is not provided by PARAMESH or UG grid. As such, the relevant communication is performed within the grid mapping subroutines directly.

In both PARAMESH and UG implementations, the particles’ attribute is “smeared” across a temporary array by the generic particle mapping subroutine. Here, the temporary array represents a single leaf block from the local processor. In simulations using the PARAMESH grid, the temporary array represents each LEAF block from the local processor in turn. We assign a particle’s attribute to the temporary array when that
particle exists in the same space as the current leaf block. For details about the attribute assignment schemes available to the particle mapping sub-unit, please refer to Section 19.2.

After particle assignment, the Grid sub-unit applies simulation boundary conditions to those temporary arrays that represent blocks next to external boundaries. This may change the quantity and location of particle assignment in the elements of the temporary array. The final step in the process involves accumulating values from the temporary array into the correct cells of the computational domain. As mentioned previously, there are different strategies for UG and Paramesh grids, which are described in Section 8.9.2.1 and Section 8.9.2.2, respectively.

**FLASH Transition**

The particle mapping routines can be run in a custom debug mode which can help spot data errors (and even detect possible bugs). In this mode we inspect data for inconsistencies. To use, append the following line to the setup script:

```
-defines=DEBUG_GRIDMAPPARTICLES
```

### 8.9.2.1 Uniform Grid

The Uniform Grid algorithm for accumulating particles’ attribute on the grid is similar to the particle redistribution algorithm described in Section 8.9.1.1. We once again apply the concept of directional movement to minimize the number of communication steps:

1. Take the accumulated temporary array and iterate over all elements that correspond to the x-axis guard cells of the low block face. If a guard cell has been updated, determine its position in the neighboring block of the low block face. Copy the guard cell value and a value which encodes the destination cell into the send buffer.

2. Send the buffer to the low-side processor, and receive a buffer from the high-side processor. For processors next to a domain boundary assume periodic conditions because all processors must participate. If the simulation does not have periodic boundary conditions, there is still periodic communication at the boundaries, but the send buffers do not contain data.

3. Iterate over the elements in the receive buffer and accumulate the values into the local temporary array at the designated cells. It is possible to accumulate values in cells that represent internal cells and guard cells. A value accumulated in a guard cell will be repacked into the send buffer during the next directional (y or z) sweep.

4. Repeat steps 1-3 for the high block face.

5. Repeat steps 1-4 for the y-axis, and then the z-axis.

When the guard cell’s value is packed into the send buffer, a single value is also packed which is a 1-dimensional representation of the destination cell’s N-dimensional position. The value is obtained by using an array equation similar to that used by a compiler when mapping an array into contiguous memory. The receiving processor applies a reverse array equation to translate the value into N-dimensional space. The use of this communication protocol is designed to minimize the message size.

At the end of communication, each local temporary buffer contains accumulated values from guard cells of another block. The temporary buffer is then copied into the solution array.

### 8.9.2.2 Paramesh Grid

There are two implementations of the AMR algorithms for accumulating particles’ attribute on the grid. They are inspired by a particle redistribution algorithms Sieve and Point to Point described in Section 8.9.1.3 and Section 8.9.1.2 respectively.
The MoveSieve implementation of the mapping algorithm uses the same back and forth communication pattern as Sieve to minimize the number of message exchanges. That is, processor \( MyPE \) sends to \( MyPE + n \times (-1)^{n+1} \) and receives from \( MyPE + n \times (-1)^n \), where, \( MyPE \) is the local processor number and \( n \) is the count of the buffer exchange round. As such, this communication pattern involves a processor exchanging data with its nearest neighbor processors first. This is appropriate here because the block distribution generated by the space filling curve should be high in data locality, i.e., nearest neighbor blocks should be placed on the same processor or nearest neighbor processors.

Similarly, the Point to Point implementation of the mapping algorithm exploits the cached neighborhood knowledge, and uses a judicious combination of global communications with asynchronous receives and synchronous sends, as described in Section 8.9.1.2. Other than their communication patterns, the two implementations are very similar as described below.

1. Accumulate the temporary array values into the central section of the corresponding leaf block.

2. Divide the leaf block guard cells into guard cell regions. Determine whether the neighbor(s) to a particular guard cell region exist on the same processor.

3. If a neighbor exists on the same processor, the temporary array values are accumulated into the central cells of that leaf block. If the neighbor exists off processor, all temporary array values corresponding to a single guard cell region are copied into a send buffer. Metadata is also packed into the send buffer which describes the destination of the updated values.

4. Repeat steps 1-3 for each leaf block.

5. Carry out data exchange with off-processor destinations as described in the Section 8.9.1.3 or Section 8.9.1.2.

The guard cell region decomposition described in Step 2 is illustrated in Figure 8.10. Here, the clear regions correspond to guard cells and the gray region corresponds to internal cells. Each guard cell region contains cells which correspond to the internal cells of a single neighboring block at the same refinement.

```
   3,1   3,2   3,3
   2,1         2,3
   1,1   1,2   1,3
```

Figure 8.10: A single 2-D block showing how guard cells are divided into regions.

We use this decomposition as it makes it possible to query public PARAMESH data structures which contain the block and process identifier of the neighboring block at the same refinement. However, at times this is not enough information for finding the block neighbor(s) in a refined grid. We therefore categorize neighboring blocks as: Existing on the same processor, existing on another processor and the block and process ID are
known, and existing on another processor and the block and process ID are unknown. If the block and process identifier are unknown we use the FLASH4 corner ID. This is a viable alternative as the corner ID of a neighboring block can always be determined.

The search process also identifies the refinement level of the neighboring block(s). This is important as the guard cell values cannot be directly accumulated into the internal cells of another block if the blocks are at a different refinement levels. Instead the values must be operated on in processes known as restriction and prolongation (see Section 8.6.2). We perform these operations directly in the GridParticlesMapToMesh routines, and use quadratic interpolation during prolongation.

Guard cell data is accumulated in blocks existing on the same processor, or packed into a send buffer ready for communication. When packed into the send buffer, we keep values belonging to the same guard cell region together. This enables us to describe a whole region of guard cell data by a small amount of metadata. The metadata consists of: Destination block ID, destination processor ID, block refinement level difference, destination block corner ID (IAXIS, JAXIS, KAXIS) and start and end coordinates of destination cells (IAXIS, JAXIS, KAXIS). This is a valid technique because there are no gaps in the guard cell region, and is sufficient information for a receiving processor to unpack the guard cell data correctly.

We size the send / receive buffers according to the amount of data that needs to be communicated between processors. This is dependent upon how the PARAMESH library distributes blocks to processors. Therefore, in order to size the communication buffers economically, we calculate the number of guard cells that will accumulate on blocks belonging to another processor. This involves iterating over every single guard cell region, and keeping a running total of the number of off-processor guard cells. This total is added to the metadata total to give the size of the send buffer required on each processor. We use the maximum of the send buffer size across all processors as the local size for the send / receive buffer. Choosing the maximum possible size prevents possible buffer overflow when an intermediate processor passes data onto another processor.

After the point to point communication in step 6, the receiving processor scans the destination processor identifier contained in each metadata block. If the data belongs to this processor it is unpacked and accumulated into the central cells of the relevant leaf block. As mentioned earlier, it is possible that some guard cell sections do not have the block and processor identifier. When this happens, the receiving processor attempts to find the same corner ID in one of its blocks by performing a linear search over each of its leaf blocks. Should there be a match, the guard cells are copied into the matched block. If there is no match, the guard cells are copied from the receive buffer into the send buffer, along with any guard cell region explicitly designated for another processor. The packing and unpacking will continue until all send buffers are empty, as indicated by the result of the collective communication.

It may seem that the algorithm is unnecessarily complicated, however, it is the only viable algorithm when the block and process identifiers of the nearest block neighbors are unknown. This is the situation in FLASH3.0, in which some data describing block and process identifiers are yet to be extracted from PARAMESH. As an aside, this is different to the strategy used in FLASH2, in which the entire PARAMESH tree structure was copied onto each processor. Keeping a local copy of the entire PARAMESH tree structure on each processor is an unscalable approach because increase in the levels of resolution increases the meta-data memory overhead, which restricts the size of active particle simulations. Therefore, Point to Point method is a better option for larger simulations, and significantly, simulations that run on massively parallel processing (MPP) hardware architectures.

In FLASH3.1 we added a routine which searches non-public PARAMESH data to obtain all neighboring block and process identifiers. This discovery greatly improves the particle mapping performance because we no longer need to perform local searches on each processor for blocks matching a particular corner ID.

As another consequence of this discovery, we are able to experiment with alternative mapping algorithms that require all block and process IDs. From FLASH3.1 on we also provide a non-blocking point to point implementation in which off-processor cells are sent directly to the appropriate processor. Here, processors receive messages at incremented positions along a data buffer. These messages can be received in any order, and their position in the data buffer can change from run to run. This is very significant because the mass accumulation on a particular cell can occur in any order, and therefore can result in machine precision discrepancies. Please be aware that this can actually lead to slight variations in end-results between two runs of the exact same simulation.
8.10 GridSolvers

The GridSolvers unit groups together subunits that are used to solve particular types of differential equations. Currently, there are two types of solvers: a parallel Fast Fourier Transform package (Section 8.10.1) and various solvers for the Poisson equation (Section 8.10.2).

8.10.1 Pfft

Pfft is a parallel framework for computing a Fast Fourier Transform (FFT) on uniform grids. It can also be combined with the Multigrid solver described below in Section 8.10.2.6 to let the composite solver scale to thousands of processors.

Pfft has a layered architecture where the lower layer contains functions implementing basic tasks and primary data structures. The upper layer combines pieces from the lowest layer to interface with FLASH and create the parallel transforms. The computational part of Pfft is handled by sequential 1-dimensional FFT's, which can be from a native, vendor supplied scientific library, or from public domain packages. The current distribution of FLASH uses fftpack from NCAR for the 1-D FFTs, since that package also contains transforms that are useful with non-periodic boundary conditions.

The lowest layer has three distinct components. The first component redistributes data. It includes routines for distributed transposes for two and three dimensional data. The second component provides a uniform interface for FFT calls to hide the details of individual libraries. The third component is the data structures. There are global data structures to keep track of the type of transform, number of data dimensions, and physical and transform space information about each dimension. The dimensional information includes the start and end point of data (useful when the dimension is spread over more than one processor), the MPI communicator, the coordinates of the node in the processor grid etc. The structures also include pointers to the trigonometric tables and work space needed by the sequential FFT calls.

The upper layer of PFFT combines the lower layer routines to create end-to-end transforms in a variety of ways. The available one dimensional transforms include real-to-complex, complex-to-complex, sine and cosine transforms. They can be combined to create two or three dimensional transforms for different configuration of the domain. The two dimensional transforms support parallelization of one dimension (or a one dimensional grid of processors). The three dimensional transforms support one or two dimensional grid of processors. All transforms must have at least one dimension within the processor at all times. The data distribution changes during the computation. However, a complete cycle of forward and inverse transform restores the data distribution.

The computation of a forward three dimensional FFT in parallel involves following steps:

1. Compute 1-D transforms along x.
2. Reorder, or transpose from x-y-z to y-z-x

3. Compute 1-D transforms along y. If the transform along x was real-to-complex, this one must be a complex-to-complex transform.

4. Transpose from y-z-x to z-x-y

5. Compute 1-D FFTs along z. If the transform along x or y was real-to-complex, this must be a complex-to-complex transform.

The inverse transform can be computed by carrying out the steps described above in reverse order. The more commonly used domain decomposition in FFT based codes assumes a one dimensional processor grid:

\[ N_1 \times N_2 \times N_3 / P, \]  

(8.5)

where \( N_1 \times N_2 \times N_3 \) is the global data size and \( P \) is the number of processors. Here, the first transpose is local, while the second one is distributed. The internode communication is limited to one distributed transpose involving all the processors. However, there are two distinct disadvantages of this distribution of work:

- The size of the problem imposes an upper limit on the number of processors, in that the largest individual dimension is also the largest number of active processors. A three dimensional problem is forced to have modest individual dimensions to fit in the processor memory.

- As the machine size grows, the internode exchanges become long range, and the possibility of contention grows.

We have chosen a domain decomposition where each subdomain is a column of size

\[ N_1 \times N_2 / P_1 \times N_3 / P_2 \]

\[ P = P_1 \times P_2. \]  

(8.6)

With this distribution both the transposes are distributed in parallel. The data exchange along any one processor grid dimension is a collection of disjointed distributed transposes. Here, the contention and communication range is reduced, while the volume of data exchange is unaltered. The distributed transposes are implemented using collective MPI operation `alltoall`. In a slabwise distribution, the upper limit on the number of processors is determined by the smallest of \( < N_1, N_2, N_3 > \), where as in our distribution, the upper limit on the number of processors is the smallest of \( < N_1 \times N_2, N_2 \times N_3, N_1 \times N_3 > \).

### 8.10.1.1 Using Pfft

Pfft can only be used with a pencil grid, with the constraint that the number of processors along the IAXIS must be 1. This is because all one dimensional transforms are computed locally within a processor. However, FLASH contains a set of data movement subroutines that generate a usable pencil grid from any UG grid or any level of a PM grid. These routines are briefly explained in Section 8.10.1.2.

During the course of a simulation, Pfft can be used in two different modes. In the first mode, every instance of Pfft use will be exactly identical to every other instance in terms of domain size and the type of transforms. In this mode, the user can set the runtime parameter `pfft_setupOnce` to true, which enables the FLASH initialization process to also create and initialize all the data structures of Pfft. The finalization of the Pfft subunit is also done automatically by the FLASH shutdown process in this mode. However, if a simulation needs to use Pfft in different configurations at different instances of its use, then each set of calls to Grid.pfft for computing the transforms must be preceded by a call to Grid.pfftInit and followed by a call to Grid.pfftFinalize. In addition, the runtime parameter `pfft_setupOnce` should be set to false. A few other helper routines are available in the subunit to allow the user to query Pfft about the dimensioning of the domain, and also to map the Mesh variables from the unk array to and from Pfft compatible (single dimensional) arrays. Pfft also provides the location of wave numbers in the parallel domain; information that users can utilize to develop their own customized PDE solvers using FFT based techniques.
8.10.1.2 Pfft data movement subroutines

Mesh reconfiguration subroutines are available to generate a pencil grid for the Pfft unit from another mesh configuration. Two different implementations are available at Grid/GridSolvers/Pfft/MeshReconfiguration/PtToPt and Grid/GridSolvers/Pfft/MeshReconfiguration/Collective, with the PtToPt implementation being the default. Both implementations are able to generate an appropriate pencil grid in UG and PM mode. The pencil processor grid is automatically selected, but can be overridden by passing optional arguments to Grid_pfftInit. In UG mode they are invoked when the number of processors in the IAXIS of the FLASH grid is greater than one, and in PM mode they are always invoked. In PM mode they generate a pencil grid from a single level of the AMR grid, which may be manually specified or automatically selected as the maximum level that is fully-refined (i.e. has blocks that completely cover the computational domain at this level).

The pencil grid processor topology is stored in an MPI communicator, and the communicator may contain fewer processors than are used in the simulation. This is to ensure the pencil grid points are never distributed too finely over the processors, and naturally handles the case where the user may wish to obtain a pencil grid at a very coarse level in the AMR grid. If there are more blocks than processors then we are safe to distribute the pencil grid over all processors, otherwise we must remove a number of processors. Currently, we eliminate those processors that own zero FLASH blocks at this level, as this is a simple calculation that can be computed locally.

Both mesh reconfiguration implementations generate a map describing the data movement before moving any grid data. The map is retained between calls to the Pfft routines and is only regenerated when the grid changes. This avoids repeating the same global communications, but means communication buffers are left allocated between calls to Pfft.

In the Collective implementation, the map coordinates are used to specify where the FLASH data is copied into a send communication buffer. Two MPI_Alltoall calls then move data to the appropriate pencil processor at coordinates (J,K). Here, the first MPI_Alltoall moves data to processor (J,0), and the second MPI_Alltoall moves data to processor (J,K). The decision to use MPI_Alltoalls simplifies the MPI communication, but leads to very large send / receive communication buffers on each processor which consume:

\[
\text{Memory(bytes)} = \text{sizeof(REAL)} \times \text{total grid points at solve level} \times 2
\]

The PtToPt implementation consumes less memory compared to the Collective implementation, and communicates using point to point MPI messages. It is based upon using nodes in a linked list which contain metadata (a map) and a communication buffer for a single block fragment. There are two linked lists: one for the FLASH block fragments and one for Pfft block fragments. Metadata information about each FLASH block fragment is placed in separate messages and sent using MPI_Isend to the appropriate destination pencil grid processor.

Each destination pencil grid processor repeatedly invokes MPI_Iprobe using MPI_ANY_SOURCE, and creates a node in its Pfft list whenever it discovers a message. The MPI message is received into a metadata region of the freshly allocated node, and a communication buffer is also allocated according to the size specified in the metadata. The pencil processor continues probing for messages until the cumulative size of its node’s communication buffers is equal to the pencil grid points it has been assigned. At this stage, grid data is communicated by traversing the Pfft list and posting MPI_Irecvs, and then traversing the FLASH list and sending block fragment using MPI_Isends. After performing MPI_Waits, the received data in the nodes of the Pfft list is copied into internal Pfft arrays.

Note, the linked list is constructed using an include file stored at flashUtilities/datastructures/linkedlist. The file is named ut_listMethods.includeF90 and is meant to be included in any Fortran90 module to create lists with nodes of a user-defined type. Please see the README file, and the unit test example at flashUtilities/datastructures/linkedlist/UnitTest.

8.10.1.3 Unit Test

The unit test for Pfft solver solves the following equation:

\[
\nabla^2(F) = -13.0 \times \cos 2x \times \sin 3y 
\] (8.7)
The simplest analytical solution of this equation assuming no constants is

\[ F = \cos 2x \ast \sin 3y \]  

(8.8)

We discretize the domain by assuming \( x_{\text{min}}, y_{\text{min}}, z_{\text{min}} = 0 \), and \( x_{\text{max}}, y_{\text{max}}, z_{\text{max}} = 2\pi \). The equation satisfies periodic boundary conditions in this formulation and FFT based poisson solve techniques can be applied. In the unit test we initialize one variable of the solution data with the function \( F \), and another one with the right hand side of (8.7). We compute the forward real-to-complex transform of the solution data variable that is initialized with the right hand side of (8.7). This variable is then divided by \( k_i^2 + k_j^2 + k_k^2 \) where \( k_i, k_j \) and \( k_k \) are the wavenumbers at any point \( i,j,k \) in the domain. An inverse complex-to-real transform after the division should give the function \( F \) as result. Hence the unit test is considered successful if both the variables have matching values within the specified tolerance.

### 8.10.2 Poisson equation

The GridSolvers subunit contains several different algorithms for solving the general Poisson equation for a potential \( \phi(x) \) given a source \( \rho(x) \)

\[ \nabla^2 \phi(x) = \alpha \rho(x) . \]  

(8.9)

Here \( \alpha \) is a constant that depends upon the application. For example, when the gravitational Poisson equation is being solved, \( \rho(x) \) is the mass density, \( \phi(x) \) is the gravitational potential, and \( \alpha = 4\pi G \), where \( G \) is Newton’s gravitational constant.

#### 8.10.2.1 Multipole Poisson solver (original version)

This section describes the multipole Poisson solver that has been included in all the past releases of FLASH. It is included in the current release also, however, certain limitations found in this solver lead to a new implementation described in Section 8.10.2.2. This version is retained in FLASH4, because the new version is missing the ability to treat a non-zero minimal radius for spherical geometries and the ability to specify a point mass contribution to the potential. This will be implemented for the next coming release.

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. It currently works in 1D and 2D spherical, 2D axisymmetric cylindrical \((r,z)\), and 3D Cartesian and axisymmetric geometries. Because of the imposed symmetries, in the 1D spherical case, only the monopole term \((\ell = 0)\) makes sense, while in the axisymmetric and 2D spherical cases, only the \( m = 0 \) moments are used \((i.e., \) the basis functions are Legendre polynomials).

The multipole algorithm consists of the following steps. First, find the center of mass \( x_{\text{cm}} \)

\[ x_{\text{cm}} = \frac{\int d^3 x x \rho(x)}{\int d^3 x \rho(x)} . \]  

(8.10)

We will take \( x_{\text{cm}} \) as our origin. In integral form, Poisson’s ((8.9)) is

\[ \phi(x) = -\frac{\alpha}{4\pi} \int d^3 x' \frac{\rho(x')}{|x - x'|} . \]  

(8.11)

The Green’s function for this equation satisfies the relationship

\[ \frac{1}{|x - x'|} = 4\pi \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell + 1} Y_{\ell m}(\theta', \phi') Y_{\ell m}(\theta, \phi) , \]  

(8.12)

where the components of \( x \) and \( x' \) are expressed in spherical coordinates \((r, \theta, \phi)\) about \( x_{\text{cm}} \), and

\[ r_\lt \equiv \min\{|x|, |x'|\} \]  

\[ r_\gt \equiv \max\{|x|, |x'|\} . \]  

(8.13)
Here $Y_{\ell m}(\theta, \varphi)$ are the spherical harmonic functions

$$Y_{\ell m}(\theta, \varphi) \equiv (-1)^m \sqrt{\frac{2\ell + 1}{4\pi} \frac{(\ell - m)!}{(\ell + m)!}} P_{\ell m}(\cos \theta) e^{i m \varphi}. \quad (8.14)$$

$P_{\ell m}(x)$ are Legendre polynomials. Substituting (8.12) into (8.11), we obtain

$$\phi(\mathbf{x}) = -\alpha \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell + 1} \left\{ Y_{\ell m}(\theta, \varphi) \times \left[ r^{\ell} \int_{r<r'} d^3\mathbf{x}' \rho(\mathbf{x}') Y^*_{\ell m}(\theta', \varphi') + \frac{1}{r^{\ell+1}} \int_{r>r'} d^3\mathbf{x}' \rho(\mathbf{x}') Y^*_{\ell m}(\theta', \varphi') r^{\ell} \right] \right\}. \quad (8.15)$$

In practice, we carry out the first summation up to some limiting multipole $\ell_{\text{max}}$. By taking spherical harmonic expansions about the center of mass, we ensure that the expansions are dominated by low-multipole terms, so that for a given value of $\ell_{\text{max}}$, the error created by neglecting high-multipole terms is minimized. Note that the product of spherical harmonics in (8.15) is real-valued

$$\sum_{m=-\ell}^{\ell} Y^*_{\ell m}(\theta', \varphi') Y_{\ell m}(\theta, \varphi) = \frac{2\ell + 1}{4\pi} \left[ P_{\ell0}(\cos \theta)P_{\ell0}(\cos \theta') + 2 \sum_{m=1}^{\ell} \frac{(\ell - m)!}{(\ell + m)!} P_{\ell m}(\cos \theta)P_{\ell m}(\cos \theta') \cos (m(\varphi - \varphi')) \right]. \quad (8.16)$$

Using a trigonometric identity to split up the last cosine in this expression and substituting for the inner sums in (8.15), we obtain

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \sum_{\ell=0}^{\infty} P_{\ell0}(\cos \theta) \left[ r^{\ell} \mu_{\ell0}^{\ell}(r) + \frac{1}{r^{\ell+1}} \mu_{\ell0}^{\ell}(r) \right] - \frac{\alpha}{2\pi} \sum_{\ell=1}^{\infty} \sum_{m=1}^{\ell} P_{\ell m}(\cos \theta) \left[ (r^{\ell} \cos m\varphi) \mu_{\ell m}^{\ell}(r) + (r^{\ell} \sin m\varphi) \mu_{\ell m}^{\ell}(r) + \frac{\cos m\varphi}{r^{\ell+1}} \mu_{\ell m}^{\ell}(r) + \frac{\sin m\varphi}{r^{\ell+1}} \mu_{\ell m}^{\ell}(r) \right]. \quad (8.17)$$

The even (e)/odd (o), inner (i)/outer (o) source moments in this expression are defined to be

$$\mu_{\ell m}^{\ell}(r) \equiv \frac{(\ell - m)!}{(\ell + m)!} \int_{r>r'} d^3\mathbf{x}' r^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \cos m\varphi' \quad (8.18)$$

$$\mu_{\ell m}^{\ell}(r) \equiv \frac{(\ell - m)!}{(\ell + m)!} \int_{r>r'} d^3\mathbf{x}' r^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \sin m\varphi' \quad (8.19)$$

$$\mu_{\ell m}^{\ell}(r) \equiv \frac{(\ell - m)!}{(\ell + m)!} \int_{r<r'} d^3\mathbf{x}' r^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \cos m\varphi' \quad (8.20)$$

$$\mu_{\ell m}^{\ell}(r) \equiv \frac{(\ell - m)!}{(\ell + m)!} \int_{r<r'} d^3\mathbf{x}' r^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \sin m\varphi'. \quad (8.21)$$

The procedure is thus to compute the moment integrals ((8.18) – (8.21)) for a given source field $\rho(\mathbf{x})$, and then to use these moments in (8.17) to compute the potential.

In practice, the above procedure must take account of the fact that the source and the potential are assumed to be cell-averaged quantities discretized on a block-structured mesh with varying cell size. Also, because of the radial dependence of the multipole moments of the source function, these moments must be tabulated as functions of distance from $x_{cm}$, with an implied discretization. The solver allocates storage for moment samples spaced a distance $\Delta$ apart in radius

$$\mu_{\ell m, q}^{\ell}(r) \equiv \mu_{\ell m}^{\ell}(q\Delta) \quad \mu_{\ell m, q}^{\ell}(r) \equiv \mu_{\ell m}^{\ell}((q - 1)\Delta) \quad (8.22)$$

$$\mu_{\ell m, q}^{\ell}(r) \equiv \mu_{\ell m}^{\ell}(q\Delta) \quad \mu_{\ell m, q}^{\ell}(r) \equiv \mu_{\ell m}^{\ell}((q - 1)\Delta). \quad (8.23)$$
The sample index $q$ varies from 0 to $N_q$ $(\mu_{\ell_m,0}^{eo}$ and $\mu_{\ell_m,0}^{eo}$ are not used). The sample spacing $\Delta$ is chosen to be one-half the geometric mean of the $x$, $y$, and $z$ cell spacings at the highest level of refinement, and $N_q$ is chosen to be large enough to span the diagonal of the computational volume with samples.

Determining the contribution of individual cells to the tabulated moments requires some care. To reduce the error caused by the grid geometry, in each cell $ijk$ an optional subgrid can be established (see \texttt{mpole_subSample}) consisting of $N'$ points at the locations $\mathbf{x}_{i'j'k'}$, where

\begin{equation}
    x_{i'} = x_i + \left(i' - 0.5(N' - 1)\right)\frac{\Delta x_i}{N'} , \quad i' = 0 \ldots N' - 1 \tag{8.24}
\end{equation}

\begin{equation}
    y_{j'} = y_j + \left(j' - 0.5(N' - 1)\right)\frac{\Delta y_j}{N'} , \quad j' = 0 \ldots N' - 1 \tag{8.25}
\end{equation}

\begin{equation}
    z_{k'} = z_k + \left(k' - 0.5(N' - 1)\right)\frac{\Delta z_k}{N'} , \quad k' = 0 \ldots N' - 1 \tag{8.26}
\end{equation}

and where $\mathbf{x}_{ijk}$ is the center of cell $ijk$. (For clarity, we have omitted $ijk$ indices on $\mathbf{x}^r$ as well as all block indices.) For each subcell, we assume $\rho(\mathbf{x}_{i'j'k'}) \approx \rho_{ijk}$ and then apply

\begin{equation}
    \mu_{\ell_m, q \geq q'}^{ei} \leftarrow \mu_{\ell_m, q \geq q'}^{ei} + \frac{[(\ell - m)!]}{[(\ell + m)!]} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'\Delta} r_{i'j'k'}^\ell \rho(\mathbf{x}_{i'j'k'}) P_{\ell m}(\cos \theta_{i'j'k'}) \cos \varphi_{i'j'k'} \tag{8.27}
\end{equation}

\begin{equation}
    \mu_{\ell_m, q \geq q'}^{oi} \leftarrow \mu_{\ell_m, q \geq q'}^{oi} + \frac{[(\ell - m)!]}{[(\ell + m)!]} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'\Delta} r_{i'j'k'}^\ell \rho(\mathbf{x}_{i'j'k'}) P_{\ell m}(\cos \theta_{i'j'k'}) \sin \varphi_{i'j'k'} \tag{8.28}
\end{equation}

\begin{equation}
    \mu_{\ell_m, q \leq q'}^{eo} \leftarrow \mu_{\ell_m, q \leq q'}^{eo} + \frac{[(\ell - m)!]}{[(\ell + m)!]} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'\Delta} r_{i'j'k'}^\ell \rho(\mathbf{x}_{i'j'k'}) P_{\ell m}(\cos \theta_{i'j'k'}) \cos \varphi_{i'j'k'} \tag{8.29}
\end{equation}

\begin{equation}
    \mu_{\ell_m, q \leq q'}^{oo} \leftarrow \mu_{\ell_m, q \leq q'}^{oo} + \frac{[(\ell - m)!]}{[(\ell + m)!]} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'\Delta} r_{i'j'k'}^\ell \rho(\mathbf{x}_{i'j'k'}) P_{\ell m}(\cos \theta_{i'j'k'}) \sin \varphi_{i'j'k'} , \tag{8.30}
\end{equation}

where

\begin{equation}
    q' = \left\lfloor \frac{\|\mathbf{x}_{i'j'k'}\|}{\Delta} \right\rfloor + 1 \tag{8.31}
\end{equation}

is the index of the radial sample within which the subcell center lies. These expressions introduce (hopefully) small errors when compared to ((8.18)) – (8.21), because the subgrid volume elements are not spherical. These errors are greatest when $r' \sim \Delta x$; hence, using a subgrid reduces the amount of source affected by these errors. An error of order $\Delta^2$ is also introduced by assuming the source profile within each cell to be flat. Note that the total source computed by this method ($\mu_{\ell_m,N_q}^{ei}$) is exactly equal to the total implied by $\rho_{ijk}$.

Another way to reduce grid geometry errors when using the multipole solver is to modify the AMR refinement criterion to refine all blocks containing the center of mass (in addition to other criteria that may be used, such as the second-derivative criterion supplied with \texttt{PARAMESH}). This ensures that the center-of-mass point is maximally refined at all times, further restricting the volume which contributes errors to the moments because $r' \sim \Delta x$.

The default value of $N'$ is 1; note that large values of this parameter very quickly increase the amount of time required to evaluate the multipole moments (as $N'^3$). In order to speed up the moment summations, the sines and cosines in ((8.27)) – (8.30) are evaluated using trigonometric recurrence relations, and the factorials are pre-computed and stored at the beginning of the run.

When computing the cell-averaged potential, we again employ a subgrid, but here the subgrid points fall on cell boundaries to improve the continuity of the result. Using $N' + 1$ subgrid points per dimension, we have

\begin{equation}
    x_{i'} = x_i + \left(i' - 0.5N'\right)\frac{\Delta x_i}{N'} , \quad i' = 0 \ldots N' \tag{8.32}
\end{equation}

\begin{equation}
    y_{j'} = y_j + \left(j' - 0.5N'\right)\frac{\Delta y_j}{N'} , \quad j' = 0 \ldots N' \tag{8.33}
\end{equation}

\begin{equation}
    z_{k'} = z_k + \left(k' - 0.5N'\right)\frac{\Delta z_k}{N'} , \quad k' = 0 \ldots N' . \tag{8.34}
\end{equation}
The cell-averaged potential in cell $ijk$ is then

$$\phi_{ijk} = \frac{1}{N'} \sum_{i'j'k'} \phi(x_{i'j'k'}) , \tag{8.35}$$

where the terms in the sum are evaluated via (8.17) up to the limiting multipole order $\ell_{\text{max}}$.

### 8.10.2.2 Multipole Poisson solver (improved version)

The multipole Poisson solver is based on a multipolar expansion of the source (mass for gravity, for example) distribution around a conveniently chosen center of expansion. The angular number $L$ entering this expansion is a measure of how detailed the description of the source distribution will be on an angular basis. Higher $L$ values mean higher angular resolution with respect to the center of expansion. The multipole Poisson solver is thus appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. For problems which require high spatial resolution throughout the entire domain (like, for example, galaxy collision simulations), the multipole Poisson solver is less suited, unless one is willing to go to extremely (computationally unfeasible) high $L$ values. For stellar evolution, however, the multipole Poisson solver is the method of choice.

The new implementation of the multipole Poisson solver is located in the directory `source/Grid/GridSolvers/Multipole_new`.

This implementation improves upon the original implementation in many ways: i) efficient memory layout ii) elimination of numerical over- and underflow errors for large angular momenta when using astrophysical (dimensions $\approx 10^9$) domains iii) elimination of subroutine call overhead (1 call per cell), iv) minimization of error due to non-statistical distributions of moments near the multipolar origin. The following paragraphs explain the new approach to the multipole solver and an explanation of the above improvements. Details about the theory of the new implementation of the Poisson solver can be found in Couch (2013).

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. It currently works in 1D spherical, 2D spherical, 2D cylindrical, 3D Cartesian and 3D cylindrical. Symmetries can be specified for the 2D spherical and 2D cylindrical cases (a horizontal symmetry plane along the radial axis) and the 3D Cartesian case (assumed axisymmetric property). Because of the radial symmetry in the 1D spherical case, only the monopole term ($\ell = 0$) contributes, while for the 3D Cartesian axisymmetric, the 2D cylindrical and 2D spherical cases only the $m = 0$ moments need to be used (the other $m \neq 0$ moments effectively cancel out).

The multipole algorithm consists of the following steps. First, the center of the multipolar expansion $x_{\text{cen}}$ is determined via density-squared weighted integration over position:

$$x_{\text{cen}} = \frac{\int x \rho^2(x) \, dx}{\int \rho^2(x) \, dx} . \tag{8.36}$$

We will take $x_{\text{cen}}$ as our origin. In integral form, Poisson’s equation (8.9) becomes

$$\phi(x) = -\frac{\alpha}{4\pi} \int \frac{\rho(x')}{|x - x'|} \, dx' . \tag{8.37}$$

The inverse radial distance part can be expanded in terms of Legendre polynomials

$$\frac{1}{|x - x'|} = \sum_{\ell=0}^{\infty} \frac{x_{<}}{x_{>}} P_{\ell}(\cos \gamma) , \tag{8.38}$$

where $x_<$ ($x_>$) indicate the smaller (larger) of the magnitudes and $\gamma$ denotes the angle between $x$ and $x'$. Note, that this definition includes those cases where both magnitudes are equal. The expansion is always convergent if $\cos \gamma < 1$. Expansion of the Legendre polynomials in terms of spherical harmonics gives

$$P_{\ell}(\cos \gamma) = \frac{4\pi}{2\ell + 1} \sum_{m=-\ell}^{\ell} Y_{\ell m}^*(\theta', \phi') Y_{\ell m}(\theta, \phi) , \tag{8.39}$$
where $\theta$, $\phi$ and $\theta'$, $\phi'$ are the spherical angular components of $\mathbf{x}$ and $\mathbf{x}'$ about $\mathbf{x}_\text{cen}$. Defining now the regular $R_{\ell m}$ and irregular $I_{\ell m}$ solid harmonic functions

$$R_{\ell m}(x_<) = \sqrt{\frac{4\pi}{2\ell + 1}} x_<^\ell Y_{\ell m}(\theta, \phi)$$

$$I_{\ell m}(x_>) = \sqrt{\frac{4\pi}{2\ell + 1}} Y_{\ell m}(\theta, \phi) x_>^{\ell+1},$$

we can rewrite Eq. (8.37) in the form

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \sum_{\ell m} R_{\ell m}(x_<) I_{\ell m}(x_>) \rho(\mathbf{x}') \, d\mathbf{x'},$$

where the summation sign is a shorthand notation for the double sum over all the allowed $\ell$ and $m$ values. In FLASH both the source and the potential are assumed to be cell-averaged quantities discretized on a block-structured mesh with varying cell size. The integration must hence be replaced by a summation over all leaf cells

$$\phi(q) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}(q_<) I_{\ell m}(q_>) m(q'),$$

where $m$ denotes the cell’s mass. Note, that the symbol $q$ stands for cell index as well as its defining distance position from the expansion center in the computational domain. This discrete Poisson equation is incorrect. It contains the divergent $q' = q$ term on the rhs. The $q' = q$ contribution to the potential corresponds to the cell self potential $\phi_{\text{Self}}(q)$ and is divergent in our case because all the cell’s mass is assumed to be concentrated at the cell’s center. The value of this divergent term can easily be calculated from Eq. (8.38) by setting $\cos \gamma = 1$:

$$\phi_{\text{Self}}(q) = m(q) \frac{L + 1}{x_q},$$

where $m$ is the cell’s mass, $L$ the highest angular number considered in the expansion and $x_q$ the radial distance of the cell center from the expansion center. To avoid this divergence problem, we evaluate the potentials on each face of the cell and form the average of all cell face potentials to get the cell center potential. Eq. (8.43) will thus be replaced by

$$\phi(q) = \frac{1}{n_F} \sum_F \phi(\mathbf{x}_F)$$

and

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}([q', x_F]<) I_{\ell m}([q', x_F>]) m(q'),$$

where $\mathbf{x}_F$ is the cell face radial distance from the expansion center and $[q', x_F]<$ denotes the larger of the magnitudes between the cell center radial distance $q'$ and the cell face radial distance $x_F$. Splitting the summation over cells in two parts

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left\{ \sum_{q' \leq x_F} \sum_{\ell m} [R_{\ell m}(q') m(q')] I_{\ell m}^*(\mathbf{x}_F) + \sum_{q' > x_F} \sum_{\ell m} R_{\ell m}(\mathbf{x}_F) [I_{\ell m}^*(q') m(q')] \right\},$$

and defining the two moments

$$M_{\ell m}^R(\mathbf{x}_F) = \sum_{q' \leq x_F} R_{\ell m}(q') m(q')$$

$$M_{\ell m}^I(\mathbf{x}_F) = \sum_{q' > x_F} I_{\ell m}(q') m(q'),$$

we can rewrite Eq. (8.37) in the form

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \sum_{\ell m} R_{\ell m}(x_<) I_{\ell m}(x_>) \rho(\mathbf{x}') \, d\mathbf{x'},$$
we obtain
\[
\phi(x_F) = -\frac{\alpha}{4\pi} \left[ \sum_{\ell m} M^{R}_{\ell m}(x_F) J_{\ell m}(x_F) + \sum_{\ell m} M^{I}_{\ell m}(x_F) R_{\ell m}(x_F) \right] \quad (8.50)
\]
and using vector notation
\[
\phi(x_F) = -\frac{\alpha}{4\pi} \left[ [M^R(x_F) \cdot \mathbf{I}^c(x_F) + M^I(x_F) \cdot \mathbf{R}(x_F)] \right]. \quad (8.51)
\]

We now change from complex to real formulation. We state this for the regular solid harmonic functions, the same reasoning being applied to the irregular solid harmonic functions and all their derived moments. The regular solid harmonic functions can be split into a real and imaginary part
\[
R_{\ell m} = R^c_{\ell m} + i R^s_{\ell m}. \quad (8.52)
\]
The labels 'c' and 's' are shorthand notations for 'cosine' and 'sine', reflecting the nature of the azimuthal function of the corresponding real spherical harmonics. When inserting (8.52) into (8.51) all cosine and sine mixed terms of the scalar products cancel out. Also, due to the symmetry relations
\[
\begin{align*}
R^c_{\ell, -m} &= (-1)^m R^c_{\ell m} \\
R^s_{\ell, -m} &= (-1)^m R^s_{\ell m}
\end{align*} \quad (8.53) \quad (8.54)
\]
we can restrict ourselves to the following polar angle number ranges
\[
\begin{align*}
c & : \ell \geq 0 , \quad \ell \geq m \geq 0 \\
s & : \ell \geq 1 , \quad \ell \geq m \geq 1.
\end{align*} \quad (8.55) \quad (8.56)
\]
The real formulation of (8.51) becomes then
\[
\phi(x_F) = -\frac{\alpha}{4\pi} \left\{ \left[ \frac{M^{Rc}(x_F)}{M^{Rs}(x_F)} \right] \cdot \mathbf{\Delta} \left[ \mathbf{I}^c(x_F) \right] + \left[ \frac{M^{Ic}(x_F)}{M^{Is}(x_F)} \right] \cdot \mathbf{\Delta} \left[ \mathbf{R}^c(x_F) \right] \right\}, \quad (8.57)
\]
which, when compared to (8.51), shows, that all vectors now contain a cosine and a sine section. The \( \mathbf{\Delta} \) matrix is a diagonal matrix whose elements are equal to 2 for \( m \neq 0 \) and 1 otherwise, i.e.: \( \mathbf{\Delta} = \text{diag}(2 - \delta_{m0}). \) \quad (8.58)
The recursion relations for calculating the solid harmonic vectors are
\[
\begin{align*}
R^c_{00} &= 1 \quad (8.59) \\
R^c_{\ell \ell} &= -\frac{x R^c_{\ell -1, \ell -1} - y R^c_{\ell -1, \ell -1}}{2\ell} \quad (8.60) \\
R^s_{\ell \ell} &= -\frac{y R^s_{\ell -1, \ell -1} + x R^s_{\ell -1, \ell -1}}{2\ell} \quad (8.61) \\
R^{c/s}_{\ell m} &= \frac{(2\ell - 1)z R^{c/s}_{\ell -1, m} - r^2 R^c_{\ell -2, m}}{(\ell + m)(\ell - m)}, \quad 0 \leq m < \ell \quad (8.62)
\end{align*}
\]
and
\[
\begin{align*}
I^c_{00} &= \frac{1}{r} \quad (8.63) \\
I^c_{\ell \ell} &= -\frac{(2\ell - 1) x I^c_{\ell -1, \ell -1} - y I^c_{\ell -1, \ell -1}}{r^2} \quad (8.64) \\
I^s_{\ell \ell} &= -\frac{(2\ell - 1) y I^s_{\ell -1, \ell -1} + x I^s_{\ell -1, \ell -1}}{r^2} \quad (8.65) \\
I^{c/s}_{\ell m} &= \frac{(2\ell - 1)z I^{c/s}_{\ell -1, m} - [(\ell - 1)^2 - m^2] r^{c/s}_{\ell -2, m}}{r^2}, \quad 0 \leq m < \ell \quad (8.66)
\end{align*}
\]
in which \(x, y, z\) are the cartesian location coordinates of the cell face and \(r^2 = x^2 + y^2 + z^2\). For geometries depending on polar angles one must first calculate the corresponding cartesian coordinates for each cell before applying the recursions. In FLASH, the order of the two cosine and sine components for each solid harmonic vector is such that \(\ell\) precedes \(m\). This allows buildup of the vectors with maximum number of unit strides. The same applies of course for the assembly of the moments. For 2D cylindrical and 2D spherical geometries only the \(m = 0\) parts of both recursions are needed, involving only the cartesian \(z\) coordinate and \(r^2\). Symmetry along the radial axes of these 2D geometries inflicts only the sign change \(z \rightarrow -z\), resulting in the symmetry relations \(R_{\ell 0}^c \rightarrow R_{\ell 0}^s\) for even \(\ell\) and \(R_{\ell 0}^c \rightarrow -R_{\ell 0}^s\) for odd \(\ell\), the same holding for the irregular solid harmonic vector components. Thus symmetry in 2D can effectively be treated by halving the domain size and multiplying each even \(\ell\) moments by a factor of 2 while setting the odd \(\ell\) moments equal to 0. For 3D cartesian geometries introduction of symmetry is far more complicated since all \(m\) components need to be taken into account. It is not sufficient to simply reduce the domain to the appropriate size and multiply the moments by some factor, but rather one would have to specify the exact symmetry operations intended (generators of the symmetry group \(O_h\) or one of its subgroups) in terms of their effects on the \(x, y, z\) cartesian basis. The resulting complications in calculating symmetry adapted moments outweighs the computational gain that can be obtained from it. Options for 3D symmetry are thus no longer available in the improved FLASH multipole solver. The ‘octant’ symmetry option from the old multipole solver, using only the monopole \(\ell = 0\) term, was too restrictive in its applicability (exact only for up to angular momenta \(\ell = 3\) due to cancellation of the solid harmonic vector components).

From the above recursion relations (8.59-8.66), the solid harmonic vector components are functions of \(x^i y^j z^k\) monomials, where \(i + j + k = \ell\) for the \(R\) and (formally) \(i + j + k = -(\ell + 1)\) for the \(I\). For large astrophysical coordinates and large \(\ell\) values this leads to potential computational over- and underflow. To get independent of the size of both the coordinates and \(\ell\) we introduce a damping factor \(Dx, Dy, Dz\) for the coordinates and for each solid harmonic type before entering the recursions. \(D\) will be chosen such that for the highest specified \(\ell = L\) we will have approximately a value close to 1 for both solid harmonic components:

\[
R_{\ell m}^c/s \approx 1 \quad (8.67)
\]

\[
I_{\ell m}^c/s \approx 1. \quad (8.68)
\]

This ensures proper handling of size at the solid harmonic function evaluation level and one does not have to rely on size cancellations at a later stage when evaluating the potential via Eq.(8.57). We next state the evaluation of the damping factor \(D\). Due to the complicated nature of the recursions, the first step is to find solid harmonic components which have a simple structure. To do this, consider a cell face with \(x, y, z\) coordinates for each solid harmonic type before entering the recursions. \(D\) will be chosen such that for the highest specified \(\ell = L\) we will have approximately a value close to 1 for both solid harmonic components:

\[
|R_{\ell 0}| = \frac{r^\ell}{\ell!} \quad (8.69)
\]

\[
|I_{\ell 0}| = \frac{\ell!}{r^{\ell+1}}. \quad (8.70)
\]

Since \(r = \sqrt{x^2 + y^2 + z^2}\), damping of the coordinates with \(D\) results in a damped radial cell face distance \(Dr\). Inserting this result into (8.69) and (8.70) and imposing conditions (8.67) and (8.68) results in

\[
D_R = \frac{1}{r} \sqrt[L]{L!} \approx \frac{1}{r} \frac{L}{e} \sqrt{2\pi L} \quad (8.71)
\]

\[
D_I = \frac{1}{r} \frac{L+1}{L} \sqrt[L]{L!} \approx \frac{1}{r} \frac{L}{e} \sqrt{\frac{2\pi e^2}{L}} \quad (8.72)
\]

where the approximate forms are obtained by using Stirling’s factorial approximation formula for large \(L\). In FLASH only the approximate forms are computed for \(D_R\) and \(D_I\) to avoid having to deal with factorials of large numbers.

From the moment defining equations (8.48) and (8.49) we see, that the moments are sums over subsets of cell center solid harmonic vectors multiplied by the corresponding cell mass. From Eq.(8.57) it follows that for highest accuracy, the moments should be calculated and stored for each possible cell face. For high
refinement levels and/or 3D simulations this would result in an unmanageable request for computer memory. Several cell face positions have to be bundled into radial bins $Q$ defined by lower and upper radial bounds. Once a cell center solid harmonic vector pair $R(q)$ and $I(q)$ for a particular cell has been calculated, its radial bin location $q \to Q$ is determined and its contribution is added to the radial bin moments $M^R(Q)$ and $M^I(Q)$. The computational definition of the radial bin moments is

$$M^R(Q) = \sum_{q \leq Q} R(q)m(q)$$

$$M^I(Q) = \sum_{q \geq Q} I(q)m(q),$$

where $q \leq Q$ means including all cells belonging to $Q$ and all radial bins with lower radial boundaries than $Q$. The two basic operations of the multipole solver are thus: i) assembly of the radial bin moments and ii) formation of the scalar products via Eq. (8.75) to obtain the potentials. The memory declaration of the moment array should reflect the way the individual moment components are addressed and the most efficient layout puts the angular momentum indices in rows and the radial bin indices in columns.

How do we extract moments $M^R(x)$ and $M^I(x)$ at any particular position $x$ inside the domain (and, in particular, at the cell face positions $x_{Fr}$), which are ultimately needed for the potential evaluation at that location? Assume that the location $x$ corresponds to a particular radial bin $x \to Q$. Consider the three consecutive radial bins $Q - 1$, $Q$ and $Q + 1$, together with their calculated moments:

$$\begin{bmatrix}
M^R(Q - 1) & M^R(Q) & M^R(Q + 1) \\
M^I(Q - 1) & M^I(Q) & M^I(Q + 1)
\end{bmatrix}$$

(8.75)

Let us concentrate on the $Q$ bin, whose lower and upper radial limits are shown as solid vertical lines. The radial distance corresponding to $x$ splits the $Q$ bin into two parts: the left fractional part, denoted $R_{frac}$, and the right fractional part, denoted $I_{frac}$. Since both $M^R(Q - 1)$ and $M^I(Q + 1)$ are completely contained respectively in $M^R(Q)$ and $M^I(Q)$, the moments at $x$ be approximately evaluated as:

$$M^R(x) = M^R(Q - 1) + R_{frac} [M^R(Q) - M^R(Q - 1)]$$

$$M^I(x) = M^I(Q + 1) + I_{frac} [M^I(Q) - M^I(Q + 1)],$$

(8.76)

(8.77)

The extraction of the moments via (8.76) and (8.77) is of course an approximation that relies on the statistically dense distribution of the individual cell center moments inside each radial bin. For bins which are reasonably far away from the expansion center this statistical approximation is valid but for those close to the expansion center the statistical distribution does not hold and calculating the moments via the above scheme introduces a large statistical error. The way out of this problem is to move from a statistical radial bin description around the expansion center to a more discrete one, by constructing very narrow isolated radial bins. The code is thus forced to analyze the detailed structure of the geometrical domain grid surrounding the expansion center and to establish the inner radial zone of discrete distributed radial bins. The statistical radial bins are then referred to as belonging to the outer radial zone(s).

While the structure of the inner radial zone is fixed due to the underlying geometrical grid, the size of each radial bin in the outer radial zones has to be specified by the user. There is at the moment no automatic derivation of the optimum (accuracy vs memory cost) bin size for the outer zones. There are two types of radial bin sizes defined for the FLASH multipole solver: i) exponentially and/or ii) logarithmically growing:

$$\text{exponential bin size upper radial limit} = s \cdot \Delta r \cdot Q^i$$

$$\text{logarithmic bin size upper radial limit} = s \cdot \Delta r \cdot \frac{e^{tQ} - 1}{e^t - 1},$$

(8.78)

(8.79)

In these definitions, $\Delta r$ is a small distance ’atomic’ (basic unit) radial measure, defined as half the geometric mean of appropriate cell dimensions at highest refinement level, $s$ is a scalar factor to optionally increase or decrease the atomic unit radial measure and $Q = 1, 2, \ldots$ is a local bin index counter for each outer zone.
The atomic radial distance \( \Delta r \) is calculated for each individual domain geometry as follows:

<table>
<thead>
<tr>
<th>Domain Geometry</th>
<th>( \Delta r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D cartesian</td>
<td>( \frac{1}{2} \sqrt{\Delta x \Delta y \Delta z} )</td>
</tr>
<tr>
<td>3D cylindrical</td>
<td>( \frac{1}{2} \sqrt{\Delta R \Delta z} )</td>
</tr>
<tr>
<td>2D cylindrical</td>
<td>( \frac{1}{2} \Delta R )</td>
</tr>
<tr>
<td>2D spherical</td>
<td>( \frac{1}{2} \Delta R )</td>
</tr>
<tr>
<td>1D spherical</td>
<td>( \frac{1}{2} \Delta R )</td>
</tr>
</tbody>
</table>

where \( \Delta x, \Delta y, \Delta z \) are the usual cartesian cell dimensions and \( \Delta R \) is the radial cell dimension. Note, that since \( \Delta r \) measures a basic radial unit along the radial distance from the expansion center (which, for approximate spherical problems, is located close to the domain’s geometrical origin), only those cell dimensions for calculating each \( \Delta r \) are taken, which are directly related to radial distances from the geometrical domain origin. For 3D cylindrical domain geometries for example, only the radial cylindrical and z-coordinate cell dimensions determine the 3D radial distance from the 3D cylindrical domain origin. The angular coordinate is not needed. Likewise for spherical domains only the radial cell coordinate is of importance. Definitions (8.78) and (8.79) define the upper limit of the radial bins. Hence in order to obtain the true bin size for the \( Q \)-th bin one has to subtract its upper radial limit from the corresponding one of the \( (Q-1) \)-th bin:

\[
\begin{align*}
Q\text{-th exponential bin size} & = s \cdot \Delta r \cdot [Q^t - (Q - 1)^t] \\
Q\text{-th logarithmic bin size} & = s \cdot \Delta r \cdot e^{t(Q-1)}.
\end{align*}
\]

In principle the user can specify as many outer zone types as he/she likes, each having its own exponential or logarithmic parameter pair \( \{s, t\} \).

Multithreading of the code is currently enabled in two parts: 1) during moment evaluation and 2) during potential evaluation. The threading in the moment evaluation section is achieved by running multiple threads over separate, non-conflicting radial bin sections. Moment evaluation is thus organized as a single loop over all relevant radial bins on each processor. Threading over the potential evaluation is done over blocks, as these will address different non-conflicting areas of the solution vector.

The improved multipole solver was extensively tested and several runs have been performed using large domains (> \( 10^{10} \)) and extremely high angular numbers up to \( L = 100 \) for a variety of domain geometries. Currently, the following geometries can be handled: 3D cartesian, 3D cylindrical, 2D cylindrical, 2D spherical and 1D spherical. The structure of the code is such that addition of new geometries, should they ever be needed by some applications, can be done rapidly.

8.10.2.3 Multipole Poisson solver unit test (MacLaurin spheroid)

The first unit test for the multipole Poisson solver is based on the MacLaurin spheroid analytical gravitational solution given in section 25.3.4. The unit test sets up a spheroid with uniform unit density and determines both the analytical and numerical gravitational fields. The absolute relative error based on the analytical solution is formed for each cell and the maximum of all the absolute errors is compared to a predefined error tolerance value for a particular uniform refinement level. The multipole unit test runs in 2D cylindrical, 2D spherical, 3D cartesian and 3D cylindrical geometries and threaded multipole unit tests are also available.

8.10.2.4 Tree Poisson Solver

The tree solver is based on the Barnes & Hut (1986, Nature, 324, 446) tree code for calculation of gravity forces in N-body simulations. However, it is more general and it includes some more modern features described for instance in Salmon & Warren (1994, J. Comp. Phys, 136, 155), Springer (2005, MNRAS, 364, 1105) and other works. It builds a global octal tree over the whole computational domain, communicates its part to all processors and uses it for calculations of various physical problems provided as separate units in source/physics directory. The global tree is an extension of the AMR mesh octal tree down to individual cells. The communication of the tree is implemented so that only parts of the tree that are needed for the calculation of the potential on a given processor are sent to it. The calculation of physical problems is done by walking the tree for each grid cell (hereafter point-of-calculation) and evaluating whether the tree node should be used for calculation or whether its children should be open.
The tree solver is connected to physical units by several wrapper subroutines that are called at specific places of the tree build and tree walk, and that call corresponding subroutines of physical units. In this way, physical units can include arbitrary quantity into the tree, and then, use it to calculate some other physical quantity by integrating contributions of all tree nodes during the tree walk. In this version, the only working unit is physics/Gravity/GravityMain/Poisson/BHTree which calculates the gravitational potential.

The tree solver algorithm consists of four parts. The first one, communication of block properties, is called only if the AMR grid changes. The other three, building of the tree, communication of the tree and calculation of the potential, are called in each time-step.

Communication of block properties. In recent version, each processor needs to know some basic information about all blocks in the simulation. It includes arrays: modetype, lrefine and child. These arrays are distributed from each processor to all the other processors. They can occupy a substantial amount of memory on large number of processors (memory required for statically allocated arrays of the tree solver can be calculated by a script tree_mem_use.py).

Building the tree. The global tree is constructed from bottom and the process consists of three steps. In the first one, the so-called block-tree is constructed in each leaf block on a given processor. The block-trees are 1-dimensional dynamically allocated arrays (see Figure 8.12) and pointers to them are stored in array gr_bhTreeArray. In the second step, top nodes of block-trees (corresponding to whole blocks) are distributed to all processors and stored in array gr_bhTreeParentTree. In the last step, higher nodes of the parent tree are calculated by each processor and stored in the gr_bhTreeParentTree array. At the end, each processor holds information about the global tree down to the level of leaf blocks. During the whole process of tree building, 5 subroutines providing the interface to physical units are called: gr_bhFillBotNode, gr_bhAccBotNode, gr_bhAccNode, gr_bhNormalizeNode and gr_bhPostprocNode (see their auto-documentation and source code for details). Each of them calls a corresponding subroutine of all physical units with a name where the first two letters 'gr' are replaced with the name of the unit (e.g. Gravity_bhFillBotNode).

Communication of the tree. Most of the tree data is contained on the bottom levels in individual block-trees. In order to save memory and communication time, only parts of block-trees that are needed on a given processor are sent to it. The procedure consists of three steps. In the first one, a level down to which each block-tree has to be sent to each processor is determined. For a given block-tree, it is done by evaluating the criterion for the node acceptance (traditionally called multipole acceptance criterion, shortly MAC) for all blocks on a remote processor, searching for the maximum level down to which the evaluated node will be needed on a given remote processor. In the second step, information about the block-tree levels which are going to be communicated is sent to all processors. This information is needed for allocation of arrays in which block-trees are stored on remote processors. In the third step, the block-tree arrays are allocated, all

---

**Figure 8.12:** Example of a block-tree in case of nxb=nyb=nzb=8 and in case physical units do not store any further information to tree nodes (masses and mass centre positions are included by the tree solver itself).
block-trees for a given processor are packed into a single message and the messages are communicated.

The MAC is implemented in subroutine `gr_bhMAC` which includes only a simple geometrical MAC used also by Barnes & Hut. The node is accepted for calculation if

\[ \frac{S_{node}}{D} < \text{gr\_bhTreeLimAngle}, \]

where \( S_{node} \) is the node size (defined as the largest edge of the corresponding cuboid) and \( D \) is the distance between the node and the point-of-calculation. Additionally, `gr_bhMAC` checks that the point-of-calculation is not located within the node itself enlarged by factor `gr_bhTreeSafeBox`. On the top of that, `gr_bhMAC` calls MACs of physical units and the node is accepted only if all criteria are fulfilled.

**Tree walk.** The tree is traversed from the top to the bottom, evaluating MAC of each node and in case it is not fulfilled, continuing the tree walk with its children. If the node’s MAC is fulfilled, the node is accepted for the calculation and subroutine `gr_bhBotNodeContrib` or `gr_bhNodeContrib` is called, depending on whether it is a bottom-most node (i.e. a single grid cell) or higher node, respectively. These subroutines only call the corresponding subroutines of physical units (e.g. `Gravity_bhNodeContrib`). This is the most CPU-intensive part of the tree solver, it usually takes more than 90% of the total tree solver time. It is completely parallel and it does not include any communication (apart from sending some statistics to the main processor at the end).

The tree solver includes several implementations of the tree walk. In the default (“standard”) algorithm, instead of walking the whole tree for each cell individually, MAC is at first evaluated for whole mesh block (interacting with some node). If the node is accepted and if the node is a parent node (i.e. corresponding to whole mesh block), the node is accepted for all cells of the block and the contribution of the node is added to them. However, the node contribution is calculated separately for each cell, because the distance between the node and individual cells differs. Another possibility is so called unified tree walk, in which the whole tree is traversed from the top down to nodes fulfilling MAC for each cell separately. This can be enabled by setting the runtime parameter `gr_bhUseUnifiedTW` to true. In future, we will include tree walk algorithm based on priority queues needed by so called SumSquare MAC described by Salmon & Warren (1994) which provides much better error control.

The tree solver supports isolated and periodic boundary conditions that can be set independently in each direction. In the latter case, when a node is considered for MAC evaluation and eventually calculation by calling `Gravity_bhNodeContrib` or `Gravity_bhBotNodeContrib`, periodic copies of the node are checked, and the minimum distance among the node periodic copies is taken in account. This allows for instance to calculate gravitational potential with periodic boundary conditions using the Ewald method (see description of the Gravity unit).

### 8.10.2.5 Tree Poisson solver unit test

The unit test for the tree gravity solver calculates the gravitational potential of the Bonnor-Ebert sphere (Bonnor, W. B., 1956, MNRAS, 116, 351) and compares it to the analytical potential. The density distribution and the analytical potential are calculated by the python script `bes-generator.py`. The simulation setup only reads the file with radial profiles of these quantities and sets it on the grid. It also normalizes the analytical potential (adds a constant to it) so that the minimum values of the analytical and numerical potential are the same. The error of the gravitational potential calculated by the tree code is stored in the field array `PERR` (written into the PlotFile). The maximum absolute and relative errors are written into the log file.

### 8.10.2.6 Multigrid Poisson solver

This section of the User’s Guide is taken from a paper by Paul Ricker, “A Direct Multigrid Poisson Solver for Oct-Tree Adaptive Meshes” (2007). Dr. Ricker wrote an original version of this multigrid algorithm for FLASH2. The Flash Center adapted it to FLASH3.

Structured adaptive mesh refinement provides some challenges for the implementation of effective, parallel multigrid methods. In the case of patch-based meshes, Huang & Greengard (2000) presents an algorithm which works by using the coarse-grid solution to impose boundary values on the fine grid. Discontinuities
in the solution caused by jumps in refinement are resolved through iterative calculation of the residual and subsequent correction. While this is not a multigrid method in the standard sense, it still provides significant convergence acceleration.

The adaptation of this method to the FLASH grid structure (Ricker, 2007) requires a few modifications. The original formulation required that there be shared points between the coarse and fine patches. Contrast this with finite-volume, nested-cell, cell-averaged grids as used in FLASH.

Another difference between the method of (Ricker 2007) and Huang & Greengard is that an oct-tree undoubtedly has neighboring blocks of the same refinement, while a patch-based mesh would not. This problem is solved through uniform prolongation of boundaries from coarse-to-fine, with simple relaxation done to eliminate the slight error introduced between adjacent cells.

One final change between the two methods is that the original computes new sources at the boundary between corrections, while the propagation here is done through nested solves on various levels.

The entire algorithm requires that the PARAMESH grid be reset such that all blocks at refinement above some level $\ell$ are set as temporarily nonexistent. This is required so that guardcell filling can occur at only that level, neglecting blocks at a higher level of refinement. This requires some global communication by PARAMESH.

The method requires three basic operators over the solution $\phi$ on the grid: taking the residual, restricting a fine-level function to coarser-level blocks, and prolonging values from the coarse level to the faces of fine level blocks in order to impose boundary values for the fine mesh problems.

The residual is calculated such that:

$$ R(x) \equiv 4\pi G \rho(x) - \nabla^2 \tilde{\phi}(x) . $$

This is accomplished through the application of the finite difference laplacian, defined on level $\ell$ with length-scales $\Delta x_\ell$, $\Delta y_\ell$ and $\Delta z_\ell$.

$$ D_\ell \tilde{\phi}^{bl}_{ijk} = \frac{1}{\Delta x_\ell^2} \left( \tilde{\phi}_{i+1,j,k} - 2\tilde{\phi}_{ij,k} + \tilde{\phi}_{i-1,j,k} \right) + \frac{1}{\Delta y_\ell^2} \left( \tilde{\phi}_{i,j+1,k} - 2\tilde{\phi}_{ij,k} + \tilde{\phi}_{i,j-1,k} \right) + \frac{1}{\Delta z_\ell^2} \left( \tilde{\phi}_{i,j,k+1} - 2\tilde{\phi}_{ij,k} + \tilde{\phi}_{i,j,k-1} \right) . $$

The restriction operator $R_\ell$ for block interior zones $(i, j, k)$ is:

$$ (R_\ell \phi^{\mathcal{P}(c)})_{ijk} \equiv \frac{1}{2^d} \sum_{i'j'k'} \tilde{\phi}^{\mathcal{P}(c),\ell+1}_{i'j'k'} , $$

where the indices $(i', j', k')$ refer to the zones in block $c$ that lie within zone $(i, j, k)$ of block $\mathcal{P}(c)$. We apply the restriction operator throughout the interiors of blocks, but its opposite, the prolongation operator $I_\ell$,
need only be defined on the edges of blocks, because it is only used to set boundary values for the direct single-block Poisson solver:

$$\mathcal{I}_l \tilde{\phi}_{l,j',k'}^\ell,\ell+1 \equiv \sum_{p,q,r=-2}^2 \alpha_{i,j',k'pq}\tilde{\phi}_{l+1,p,j+q,k+r}^{P(c),\ell} \quad (8.88)$$

When needed, boundary zone values are set as for the difference operator. We use conservative quartic interpolation to set edge values, then solve with homogeneous Dirichlet boundary conditions after using second-order boundary-value elimination. The coefficients $\alpha$ determine the interpolation scheme. For the $-x$ face in 3D,

$$\alpha_{1/2,j'k'pq} = \beta_p \gamma_{j'q} \gamma_{k'r} \quad (8.89)$$

$$(\beta_p) = \left(\begin{array}{cccc}
-\frac{1}{12} & \frac{7}{12} & \frac{7}{12} & 0
\end{array}\right)$$

$$(\gamma_{j'q}) = \left\{\begin{array}{ll}
\left(\begin{array}{cccc}
-\frac{3}{128} & \frac{11}{64} & 1 & -\frac{11}{64} & \frac{3}{128}
\end{array}\right) & j' \text{ odd} \\
\left(\begin{array}{cccc}
\frac{3}{128} & -\frac{11}{64} & 1 & \frac{11}{64} & -\frac{3}{128}
\end{array}\right) & j' \text{ even}
\end{array}\right.$$  

Interpolation coefficients are defined analogously for the other faces. Note that we use half-integer zone indices to refer to averages over the faces of a zone; integer zone indices refer to zone averages.

### 8.10.2.7 The direct solver

In the case of problems with Dirichlet boundary conditions, a $d$-dimensional fast sine transform is used. The transform-space Green’s Function for this is:

$$G_{ijk}^\ell = -\frac{1}{16\pi G} \left[ \frac{1}{\Delta x_\ell^2} \sin^2 \left(\frac{i\pi}{2n_x}\right) + \frac{1}{\Delta y_\ell^2} \sin^2 \left(\frac{j\pi}{2n_y}\right) + \frac{1}{\Delta z_\ell^2} \sin^2 \left(\frac{k\pi}{2n_z}\right) \right]^{-1}. \quad (8.90)$$

However, to be able to use the block solver in a general fashion, we must be able to impose arbitrary boundary conditions per-block. In the case of nonhomogeneous Dirichlet boundary values, boundary value elimination may be used to generalize the solver. For instance, at the $-x$ boundary:

$$\rho_{1jk} \rightarrow \rho_{1jk} - \frac{2}{\Delta x^2_\ell} \phi(x_{1/2}, y_j, z_k) \quad . \quad (8.91)$$

For periodic problems only the coarsest block must be handled differently; block adjacency for finer levels is handled naturally. The periodic solver uses a real-to-complex FFT with the Green’s function:

$$G_{ijk}^\ell = \left\{\begin{array}{ll}
-\frac{1}{16\pi G} \left[ \frac{1}{\Delta x_\ell^2} \sin^2 \left(\frac{(i-1)\pi}{n_x}\right) + \frac{1}{\Delta y_\ell^2} \sin^2 \left(\frac{(j-1)\pi}{n_y}\right) + \frac{1}{\Delta z_\ell^2} \sin^2 \left(\frac{(k-1)\pi}{n_z}\right) \right]^{-1} & i,j,k \neq 1 \\
0 & i = j = k = 1
\end{array}\right.$$  

$$(8.92)$$

This solve requires that the source be zero-averaged; otherwise the solution is non-unique. Therefore the source average is subtracted from all blocks. In order to decimate error across same-refinement-level boundaries, Gauss-Seidel relaxations to the outer two layers of zones in each block are done after applying the direct solver to all blocks on a level. With all these components outlined, the overall solve may be described by the following algorithm:

1. Restrict the source function $4\pi G \rho$ to all levels. Subtract the global average for the periodic case.

2. Interpolation step: For $\ell$ from 1 to $\ell_{\text{max}}$. 

(a) Reset the grid so that $\ell$ is the maximum refinement level

(b) Solve $D_\ell \tilde{\phi}^{b\ell}_{ijk} = 4\pi G \rho^{b\ell}_{ijk}$ for all blocks $b$ on level $\ell$.

(c) Compute the residual $R^{b\ell}_{ijk} = 4\pi G \rho^{b\ell}_{ijk} - D_\ell \tilde{\phi}^{b\ell}_{ijk}$.

(d) For each block $b$ on level $\ell$ that has children, prolong face values for $\tilde{\phi}^{b\ell}_{ijk}$ onto each child block.

3. Residual propagation step: Restrict the residual $R^{b\ell}_{ijk}$ to all levels.

4. Correction step: Compute the discrete $L_2$ norm of the residual over all leaf-node blocks and divide it by the discrete $L_2$ norm of the source over the same blocks. If the result is greater than a preset threshold value, proceed with a correction step: for each level $\ell$ from 1 to $\ell_{\text{max}}$,

(a) Reset the grid so that $\ell$ is the maximum refinement level

(b) Solve $D_\ell C^{b\ell}_{ijk} = R^{b\ell}_{ijk}$ for all blocks $b$ on level $\ell$.

(c) Overwrite $R^{b\ell}_{ijk}$ with the new residual $R^{b\ell}_{ijk} - D_\ell C^{b\ell}_{ijk}$ for all blocks $b$ on level $\ell$.

(d) Correct the solution on all leaf-node blocks $b$ on level $\ell$: $\tilde{\phi}^{b\ell}_{ijk} \rightarrow \tilde{\phi}^{b\ell}_{ijk} + C^{b\ell}_{ijk}$.

(e) For each block $b$ on level $\ell$ that has children, interpolate face boundary values of $C^{b\ell}_{ijk}$ for each child.

5. If a correction step was performed, return to the residual propagation step.

The above procedure requires storage for $\tilde{\phi}$, $C$, $R$, and $\rho$ on each block, for a total storage requirement of $4n_x n_y n_z$ values per block. Global communication is required in computing the tolerance-based stopping criterion.

8.10.2.8 A Hybrid Poisson Solver: Interfacing PFFT with Multigrid

We can improve the performance of the Multigrid solver in Section 8.10.2.6 by replacing single block FFTs with a parallel FFT at a specified coarse level, where, the coarse level is any level which is fully refined, i.e. containing blocks that completely cover the computational domain. Currently, we automatically select the maximum refinement level that is fully refined.

There is load imbalance in the Multigrid solver because each processor performs single block FFTs on the blocks it owns. At the coarse levels there are relatively few blocks compared to available processors which means many processors are effectively idle during the coarse level solves. The introduction of PFFT, and creation of a hybrid solver, eliminates some of the coarse level solves.

The performance characteristics of the hybrid solver are described in “Optimization of multigrid based elliptic solver for large scale simulations in the FLASH code” (2012) which is available online at http://onlinelibrary.wiley.com/doi/10.1002/cpe.2821/pdf. Performance results are obtained using the PFFT_PoissonFD unit test.

8.10.3 Using the Poisson solvers

The GridSolvers subunit solves the Poisson equation ((8.9)). Two different elliptic solvers are supplied with FLASH: a multipole solver, suitable for approximately spherical source distributions, and a multigrid solver, which can be used with general source distributions. The multipole solver accepts only isolated boundary conditions, whereas the multigrid solver supports Dirichlet, given-value, Neumann, periodic, and isolated boundary conditions. Boundary conditions for the Poisson solver are specified using an argument to the Grid_solvePoisson routine which can be set from different runtime parameters depending on the physical context in which the Poisson equation is being solved. The Grid_solvePoisson routine is the primary entry point to the Poisson solver module and has the following interface

\[
\text{call Grid\_solvePoisson (iSoln, iSrc, bcTypes(6), bcValues(2,6), poisfact),}
\]
Table 8.3: Runtime parameters used with poisson/multipole.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpole_lmax</td>
<td>integer</td>
<td>10</td>
<td>Maximum multipole moment</td>
</tr>
<tr>
<td>quadrant</td>
<td>logical</td>
<td>.false.</td>
<td>Use symmetry to solve a single quadrant in 2D axisymmetric cylindrical ((r,z)) coordinates, instead of a half domain.</td>
</tr>
</tbody>
</table>

where \(iSolu\) and \(iSrc\) are the integer-valued indices of the solution and source (density) variables, respectively. 

\(bcTypes(6)\) is an integer array specifying the type of boundary conditions to employ on each of the (up to) 6 sides of the domain. Index 1 corresponds to the \(-x\) side of the domain, 2 to \(+x\), 3 to \(-y\), 4 to \(+y\), 5 to \(-z\), and 6 to \(+z\). The following values are accepted in the array:

<table>
<thead>
<tr>
<th>(bcTypes)</th>
<th>Type of boundary condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Isolated boundaries</td>
</tr>
<tr>
<td>1</td>
<td>Periodic boundaries</td>
</tr>
<tr>
<td>2</td>
<td>Dirichlet boundaries</td>
</tr>
<tr>
<td>3</td>
<td>Neumann boundaries</td>
</tr>
<tr>
<td>4</td>
<td>Given-value boundaries</td>
</tr>
</tbody>
</table>

Not all boundary types are supported by all solvers. In this release, \(bcValues(2,6)\) is not used and can be filled arbitrarily. Given-value boundaries are treated as Dirichlet boundaries with the boundary values subtracted from the outermost interior cells of the source; for this case the solution variable should contain the boundary values in its first layer of boundary cells on input to \(Grid\_solvePoisson\). It should be noted that if \(PARAMESH\) is used, the values must be set for all levels. Finally, \(poisfact\) is real-valued and indicates the value of \(\alpha\) multiplying the source function in (8.9).

When solutions found using the Poisson solvers are to be differenced (\(e.g.,\) in computing the gravitational acceleration), it is strongly recommended that for AMR meshes, quadratic (or better) spatial interpolation at fine-coarse boundaries is chosen. (For PARAMESH, this is automatically the case by default, and is handled correctly for Cartesian as well as the supported curvilinear geometries. But note that the default interpolation implementation may be changed at configuration time with the ‘\(-gridinterpolation=\ldots\)’ setup option; and with the default implementation, the interpolation order may be lowered with the \(interpol\_order\) runtime parameter.) If the order of the gridinterpolation of the mesh is not of at least the same order as the differencing scheme used in places like \(Gravity\_accelOneRow\), unphysical forces will be produced at refinement boundaries. Also, using constant or linear grid interpolation may cause the multigrid solver to fail to converge.

8.10.3.1 Multipole (original version)

The \(poisson/multipole\) sub-module takes two runtime parameters, listed in Table 8.3. Note that storage and CPU costs scale roughly as the square of \(mpole\_lmax\), so it is best to use this module only for nearly spherical matter distributions.

8.10.3.2 Multipole (improved version)

To include the new multipole solver in a simulation, the best option is to use the shortcut \(+newMpole\) at setup command line, effectively replacing the following setup options:

- \(-with-unit=Grid/GridSolvers/Multipole\_new\)
- \(-with-unit=physics/Gravity/GravityMain/Poisson/Multipole\)
- \(-without-unit=Grid/GridSolvers/Multipole\)

The improved multipole solver currently accepts only two setup parameters, either one switching on multi-threading:
8.10. GRIDSOLVERS

- **threadBlockList**: enables multithreaded compilation and execution.
- **threadWithinBlock**: enables multithreaded compilation and execution.

The names of these two setup parameters are misleading, since there is only one universal threading strategy used. The use of these two setup parameters is a temporary solution and will be replaced in near future by only one setup parameter.

The improved multipole solver takes several runtime parameters, whose functions are explained in detail below, together with comments about expected time and memory scaling.

- **mpole\_Lmax**: The maximum angular moment $L$ to be used for the multipole Poisson solver. Depending on the domain geometry, the memory and time scaling factors due to this variable alone are: i) 3D cartesian, 3D cylindrical $\rightarrow (L +1)(L +1)$, ii) 3D cartesian axisymmetric, 2D cylindrical, 2D spherical $\rightarrow (L +1)$, iii) 1D spherical $\rightarrow 1$. Assuming no memory limitations, the multipole solver is numerically stable for very large $L$ values. Runs up to $L = 100$ for 3D cartesian domains have been performed. For 2D geometries, $L = 1000$ was the maximum tested.

- **mpole\_2DSymmetryPlane**: In 2D coordinates, this runtime parameter enables the user to specify a plane of symmetry along the radial part of the domain coordinates. In effect, this allows a reduction of the computational domain size by one half. The code internally computes the multipole moments as if the other symmetric part is present, i.e. no memory or execution time savings can be achieved by this runtime parameter.

- **mpole\_3DAxisymmetry**: Forces rotational invariance around the main ($z$) axis in 3D cartesian domains. The assumed rotational invariance in the ($x, y$) plane effectively cancels all $m \neq 0$ multipole moments and one can restrict the calculation to the $m = 0$ multipole moments only. The time and memory savings compared to a asymmetric 3D cartesian run is thus about a factor of $(L +1)$. For 3D cylindrical domains, rotational invariance in the ($x, y$) plane is equivalent of setting up the corresponding 2D cylindrical domain, hence this runtime parameter is not honored for 3D cylindrical domains, and the user is informed about the 3D to 2D cylindrical domain reduction possibility.

- **mpole\_DumpMoments**: This parameter is meant mainly for debugging purposes. It prints the entire moment array for each radial bin for each time step. This option should be used with care and for small problems only. The output is printed to a text file named ‘<basenm>_dumpMoments.txt’, where `<basenm>` is the base name given for the output files.

- **mpole\_PrintRadialInfo**: This parameter enables showing all detailed radial bin information at each time step. This option is especially useful for optimizing the radial bin sizes. The output is written to the text file ‘<basenm>\_printRadialInfo.txt’.

- **mpole\_IgnoreInnerZone**: Controls switching on/off the radial inner zone. If it is set ‘.true.’, the inner zone will not be recognized and all inner zone radii will be treated statistically. This parameter is meant only for performing some error analysis. For production runs it should always be at its default value of false. Otherwise errors will be introduced in calculating the moments near the expansion center.

- **mpole\_InnerZoneSize**: The size defining the discrete inner zone. The size is given in terms of the inner zone smallest (atomic) radius, which is determined at each time step by analyzing the domain grid structure around the multipolar origin (expansion center). Only very rarely will this value ever have to be changed. The default setting is very conservative and only under unusual circumstances (ex: highly nonuniform grid around the expansion center) this might be necessary. This value needs to be an integer, as it is used by the code to define dimensions of certain arrays. Note, that by giving this runtime parameter a large integer value ($\geq 1000$ for domain refinement levels up to 5) one can enforce the code to use only non-statistical radial bins.

- **mpole\_InnerZoneResolution**: Defines the inner zone radial bin size for the inner zone in terms of the inner zone smallest (atomic) radius. Two inner zone radii will be considered different, if they are more
than this resolution value apart. A very tiny number (for example $10^{-8}$) will result in a complete separation of all inner zone radii into separate radial bins. The default value of 0.1 should never be surpassed, and any attempt to do so will stop the program with the appropriate information to the user. Likewise with a meaningless resolution value of 0.

- **mpole\_MaxRadialZones**: The maximum number of outer radial zones to be used. In contrast to the inner radial zone, the outer radial zones are much more important for the user. Their layout defines the performance of the multipole solver both in cpu time spent and accuracy of the potential obtained at each cell. The default value of 1 outer radial zone at maximum refinement level leads to high accuracy, but at the same time can consume quite a bit of memory, especially for full 3D runs. In these cases the user can specify several outer radial zones each having their own radial bin size determination rule.

- **mpole\_ZoneRadiusFraction\_n**: The fraction of the maximum domain radius defining the n-th outer zone maximum radial value. The total number of fractions given must match the maximum number of outer radial zones specified and the fractions must be in increasing order and less than unity as we move from the 1st outer zone upwards. The last outer zone must always have a fraction of exactly 1. If not, the code will enforce it.

- **mpole\_ZoneType\_n**: String value containing the outer radial zone type for the n-th outer zone. If set to 'exponential', the radial equation $r(Q) = s \cdot \Delta r \cdot Q^t$, defining the upper bound radius of the Q-th radial bin in the n-th outer zone, is used. If set to 'logarithmic', the radial equation $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$ is used. In these equations $Q$ is a local radial bin counting index for each outer zone and $s, t$ are parameters defining size and growth of the outer zone radial bins (see below).

- **mpole\_ZoneScalar\_n**: The scalar value $s$ in the n-th outer radial zone equation $r(Q) = s \cdot \Delta r \cdot Q^t$ or $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$. The scalar is needed to be able to increase (or decrease) the size of the first radial bin with respect to the default smallest outer zone radius $\Delta r$.

- **mpole\_ZoneExponent\_n**: The exponent value $t$ in the n-th outer radial zone equations $r(Q) = s \cdot \Delta r \cdot Q^t$ or $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$. The exponent controls the growth (shrinkage) in size of each radial bin with increasing bin index. For the first equation, growing will occur for $t > 1$, shrinking for $t < 1$ and same size for $t = 1$. For the logarithmic equation, growing will occur for $t > 0$, shrinking for $t < 0$, but the same size option $t = 0$ will not work because the denominator becomes undefined. The same size option must hence be treated using the exponential outer zone type choice.

### Runtime parameter types, defaults and options:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpole_Lmax</td>
<td>integer</td>
<td>0</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>mpole_2DSymmetryPlane</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_3DAxisymmetry</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_DumpMoments</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_PrintRadialInfo</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_IgnoreInnerZone</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_InnerZoneSize</td>
<td>integer</td>
<td>16</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>mpole_InnerZoneResolution</td>
<td>real</td>
<td>0.1</td>
<td>less than 0.1 and &gt; 0.0</td>
</tr>
<tr>
<td>mpole_MaxRadialZones</td>
<td>integer</td>
<td>1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>mpole_ZoneRadiusFraction_n</td>
<td>real</td>
<td>1.0</td>
<td>less than 1.0 and &gt; 0.0</td>
</tr>
<tr>
<td>mpole_ZoneType_n</td>
<td>string</td>
<td>exponential</td>
<td>logarithmic</td>
</tr>
<tr>
<td>mpole_ZoneScalar_n</td>
<td>real</td>
<td>1.0</td>
<td>&gt; 0.0 (exponential)</td>
</tr>
<tr>
<td>mpole_ZoneExponent_n</td>
<td>real</td>
<td>-</td>
<td>any $\neq$ 0 (logarithmic)</td>
</tr>
</tbody>
</table>

### 8.10.3.3 Tree Poisson solver

The tree gravity solver can be included by **setup** or **Config** by
Recent implementation works only in 3D Cartesian coordinates and blocks have to be logically cubic (i.e., $n_{xb}=n_{yb}=n_{zb}$). Physical dimensions of blocks can be arbitrary, however, some multipole acceptance criteria can produce inaccurate error estimates with non-cubic blocks. The computational domain can have arbitrary dimensions, and there can be more blocks with $l_{refine}=1$ (i.e., $n_{blockx}$, $n_{blocky}$ and $n_{blockz}$ can have different values).

Runtime parameters $gr_{bhPhysMACTW}$ and $gr_{bhPhysMACComm}$ control whether MACs of physical units are used in tree walk and communication, respectively. If one of them (or both) is set .false., only purely geometric MAC is used for a corresponding part of the tree solver. It is not allowed to set $gr_{bhPhysMACTW} = .false.$ and $gr_{bhPhysMACComm} = .true.$.

Runtime parameter $gr_{bhTreeLimAngle}$ allows to set the limit opening angle for the purely geometrical MAC. Another condition controlling the acceptance of the node for the calculations is that the point-of-calculation must lie out of the box obtained by increasing the considered node by factor $gr_{bhTreeSafeBox}$. There are two types of tree walk algorithm implemented: the default "standard" tree walk and the unified tree walk that can be enabled by setting $gr_{bhUseUnifiedTw}$ (see 8.10.2.4 for their description).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gr_{bhPhysMACTW}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether physical MAC should be used in tree walk</td>
</tr>
<tr>
<td>$gr_{bhPhysMACComm}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether physical MAC should be used in communication</td>
</tr>
<tr>
<td>$gr_{bhTreeLimAngle}$</td>
<td>real</td>
<td>0.5</td>
<td>limiting opening angle</td>
</tr>
<tr>
<td>$gr_{bhTreeSafeBox}$</td>
<td>real</td>
<td>1.2</td>
<td>relative size of restricted volume around node where the point-of-calculation is not allowed to be</td>
</tr>
<tr>
<td>$gr_{bhUseUnifiedTw}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether Unified tree walk should be used</td>
</tr>
</tbody>
</table>

### 8.10.3.4 Multigrid

The Grid/GridSolvers/Multigrid module is appropriate for general source distributions. It solves Poisson’s equation for 1, 2, and 3 dimensional problems with Cartesian geometries. It only supports the PARAMESH Grid with one block at the coarsest level. For any other mesh configuration it is advisable to use the hybrid solver, which switches to a uniform grid exact solve when the specified level of coarsening has been achieved. In most use cases for FLASH, the multigrid solver will be used to solve for Gravity (see: Chapter 18). It may be included by setup or Config by including:

```
physics/Gravity/GravityMain/Poisson/Multigrid
```

The multigrid solver may also be included stand-alone using:

```
Grid/GridSolvers/Multigrid
```

In which case the interface is as described above. The supported boundary conditions for the module are periodic, Dirichlet, given-value, and isolated. Due to the nature of the FFT block solver, the same type of boundary condition must be used in all directions. Therefore, only the value of $bcTypes(1)$ will be considered in the call to Grid_solvePoisson.

The multigrid solver requires the use of two internally-used grid variables: $isls$ and $icor$. These are used to store the calculated residual and solved-for correction, respectively. If it is used as a Gravity solver with isolated boundary conditions, then two additional grid variables, $imgm$ and $imgp$, are used to store the image mass and image potential.

```
Table 8.4: Runtime parameters used with Grid/GridSolvers/Multigrid.
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gr_{bhPhysMACTW}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether physical MAC should be used in tree walk</td>
</tr>
<tr>
<td>$gr_{bhPhysMACComm}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether physical MAC should be used in communication</td>
</tr>
<tr>
<td>$gr_{bhTreeLimAngle}$</td>
<td>real</td>
<td>0.5</td>
<td>limiting opening angle</td>
</tr>
<tr>
<td>$gr_{bhTreeSafeBox}$</td>
<td>real</td>
<td>1.2</td>
<td>relative size of restricted volume around node where the point-of-calculation is not allowed to be</td>
</tr>
<tr>
<td>$gr_{bhUseUnifiedTw}$</td>
<td>logical</td>
<td>.false.</td>
<td>whether Unified tree walk should be used</td>
</tr>
</tbody>
</table>
CHAPTER 8. GRID UNIT

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mg.MaxResidualNorm</td>
<td>real</td>
<td>$1 \times 10^{-6}$</td>
<td>Maximum ratio of the norm of the residual to that of the right-hand side</td>
</tr>
<tr>
<td>mg.maxCorrections</td>
<td>integer</td>
<td>100</td>
<td>Maximum number of iterations to take</td>
</tr>
<tr>
<td>mg_printNorm</td>
<td>real</td>
<td>.true.</td>
<td>Print the norm ratio per-iteration</td>
</tr>
<tr>
<td>mpole_lmax</td>
<td>integer</td>
<td>4</td>
<td>The number of multipole moments used in the isolated case</td>
</tr>
</tbody>
</table>

### 8.10.3.5 Hybrid (Multigrid with PFFT)

The hybrid solver can be used in place of the Multigrid solver for problems with

- all-periodic
- 2 periodic and 1 Neumann
- 1 periodic and 2 Neumann

boundary conditions, if the default PFFT solver variant (called DirectSolver) is used. To use the hybrid solver in this way, add `Grid/GridSolvers/Multigrid/PfftTopLevelSolve` to your setup line or request the solver in your Simulation Config file (see e.g. `unitTest/PFFT_PoissonFD`). The following setup lines create a unit test that uses first the hybrid solver and then the standard Multigrid solver.

```
./setup unitTest/PFFT_PoissonFD -auto -3d -parfile=flash_pm_3d.par -maxblocks=800 +noio
./setup unitTest/PFFT_PoissonFD -auto -3d -parfile=flash_pm_3d.par -maxblocks=800 +noio
  --without-unit=Grid/GridSolvers/Multigrid/PfftTopLevelSolve
  --with-unit=Grid/GridSolvers/Multigrid
```

It is also possible to select a different PFFT solver variant. In that case, different combinations of boundary conditions for the Poisson problem may be supported. The `HomBcTrigSolver` variant supports the largest set of combinations of boundary conditions. Use the `PfftSolver` setup variable to choose a variant. Thus, appending `PfftSolver=HomBcTrigSolver` to the setup chooses the `HomBcTrigSolver` variant. When using the hybrid solver with the PFFT variants `HomBcTrigSolver` or `SimplePeriodicSolver`, the runtime parameter `gr.pfftDiffOpDiscretize` should be set to 1.

The `Multigrid` runtime parameters given in the previous section also apply.

### 8.10.4 HYPRE

As a part of implicit time advancement we end up with a system of equations that needs to solved at every time step. In FLASH4 the HYPRE linear algebra package is used to solve these system of equations. Therefore it is necessary to install Hypre if this capability of FLASH is to be used.

`Grid.advanceDiffusion` is the API function which solves the system of equations. This API is provided by both the split and unsplit solvers. The unsplit solver uses HYPRE to solve the system of equations and split solver does a direct inverse using Thomas algorithm. Note that the split solver relies heavily on PFFT infrastructure for data exchange and significant portion of work in split `Grid.advanceDiffusion` involves PFFT routines. In the unsplit solver the data exchange is implicitly done within HYPRE and is hidden.

The steps in unsplit `Grid.advanceDiffusion` are as follows:

- Setup HYPRE grid object
- Exchange Factor B
• Set initial guess
• Compute HYPRE Matrix $M$ such that $B = MX$
• Compute RHS Vector $B$
• Compute matrix $A$
• Solve system $AX = B$
• Update solution (in FLASH4)

Mapping UG grid to HYPRE matrix is trivial, however mapping PARAMESH grid to a HYPRE matrix can be quite complicated. The process is simplified using the grid interfaces provided by HYPRE.

• Struct Grid interface
• SStruct Grid interface
• IJ System interface

The choice of an interface is tightly coupled to the underlying grid on which the problem is being solved. We have chosen the SSTRUCT interface as it is the most compatible with the block structured AMR mesh in FLASH4. Two terms commonly used in HYPRE terminology are part and box. We define these terms in equivalent FLASH4 terminology. A HYPRE box object maps directly to a leaf block in FLASH4. The block is then defined by its extents. In FLASH4 this information can be computed easily using a combination of `Grid_getBlkCornerID` and `Grid_getBlkIndexLimits`. All leaf blocks at a given refinement level form a HYPRE part. So number of parts in a typical FLASH4 grid would be given by,

$$nparts = leaf\_block(lrefine\_max) - leaf\_block(lrefine\_min) + 1$$

So, if a grid is fully refined or UG, $nparts = 1$. However, there could still be more than one box object.

Setting up the HYPRE grid object is one of the most important steps of the solution process. We use the SSTRUCT interface provided in HYPRE to setup the grid object. Since the HYPRE Grid object is mapped directly with FLASH4 grid, whenever the FLASH4 grid changes the HYPRE grid object needs to be updated. Consequently with AMR the HYPRE grid setup might happen multiple times.

Setting up a HYPRE grid object is a two step process,

• Creating stenciled relationships.
• Creating Graph relationships.

Stenciled relationships typically exist between leaf blocks at same refinement level (intra part) and graph relationships exist between leaf blocks at different refinement levels (inter part). The fine-coarse boundary is handled in such a way that fluxes are conserved at the interface (see Chapter 17 for details). UG does not require any graph relationships.

Whether a block needs a graph relationship depends on the refinement level of its neighbor. While this information is not directly available in PARAMESH, it is possible to determine whether the block neighbor is coarser or finer. Combining this information with the constraint of at best a factor of two jump in refinement at block boundaries, it is possible to compute the part number of a neighbor block, which in turn determines whether we need a graph. Creating a graph involves creating a link between all the cells on the block boundary.

Once the grid object is created, the matrix and vector objects are built on the grid object. The diffusion solve needs uninterpolated data from neighbor blocks even when there is a fine-coarse boundary, therefore it cannot rely upon the guardcell fill process. A two step process is used to handle this situation,
• Since HYPRE has access to X(at n, i.e., initial guess), the RHS vector B can be computed as MX where M is a modified Matrix.

• Similarly the value of Factor B can be shared across the fine-coarse boundary by using Grid_conserveFluxes, the fluxes need to be set in an intuitive way to way to achieve the desired effect.

With the computation of Vector B (RHS), the system can be solved using HYPRE and UNK can be updated.

8.10.4.1 HYPRE Solvers

In FLASH4 we use the HYPRE PARCSR storage format as this exposes the maximum number of iterative solvers.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Preconditioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCG</td>
<td>AMG, ILU(0)</td>
</tr>
<tr>
<td>BICGSTAB</td>
<td>AMG, ILU(0)</td>
</tr>
<tr>
<td>GMRES</td>
<td>AMG, ILU(0)</td>
</tr>
<tr>
<td>AMG</td>
<td>-</td>
</tr>
<tr>
<td>SPLIT</td>
<td>-</td>
</tr>
</tbody>
</table>

Parallel runs: One issue that has been observed is that there is a difference in the results produced by HYPRE using one or more processors. This would most likely be due to use of CG (krylov subspace methods), which involves an MPI_SUM over the dot product of the residue. We have found this error to be dependent on the type of problem. One way to get across this problem is to use direct solvers in HYPRE like SPLIT. However we have noticed that direct solvers tend to be slow. In this release there is an option to use SPLIT solver, but this solver has not been extensively tested and was used only for internal debugging purposes and the usage of HYPRE SPLIT solver in FLASH4 is purely experimental.

Customizing solvers: HYPRE exposes a lot more parameters to tweak the solvers and preconditioners mentioned above. We have only used those which are applicable to general diffusion problems. Although in general these settings might be good enough it is by no means complete and might not be applicable to all class of problems. Use of additional HYPRE parameters might require direct manipulation of FLASH4 code.

Symmetric Positive Definite (SPD) Matrix: PCG has been noticed to have convergence issues which might be related to (not necessarily),

• A non-SPD matrix generated due to round of errors (only).

• Use of BoomerAMG as PC (refer to HYPRE manual).

The default settings use PCG as the solver with AMG as preconditioner. The following parameters can be tweaked at run time,
8.11 Grid Geometry

FLASH can use various kinds of coordinates ("geometries") for modeling physical problems. The available geometries represent different (orthogonal) curvilinear coordinate systems.

The geometry for a particular problem is set at runtime (after an appropriate invocation of setup) through the geometry runtime parameter, which can take a value of "cartesian", "spherical", "cylindrical", or "polar". Together with the dimensionality of the problem, this serves to completely define the nature of the problem’s coordinate axes (Table 8.7). Note that not all Grid implementations support all geometry/dimension combinations. Physics units may also be limited in the geometries supported, some may only work for cartesian coordinates.

The core code of a Grid implementation is not concerned with the mapping of cell indices to physical coordinates; this is not required for under-the-hood Grid operations such as keeping track of which blocks are neighbors of which other blocks, which cells need to be filled with data from other blocks, and so on. Thus the physical domain can be logically modeled as a rectangular mesh of cells, even in curvilinear coordinates.

There are, however, some areas where geometry needs to be taken into consideration. The correct implementation of a given geometry requires that gradients and divergences have the appropriate area factors and that the volume of a cell is computed properly for that geometry. Initialization of the grid as well as AMR operations (such as restriction, prolongation, and flux-averaging) must respect the geometry also. Furthermore, the hydrodynamic methods in FLASH are finite-volume methods, so the interpolation must also be conservative in the given geometry. The default mesh refinement criteria of FLASH4 also currently take geometry into account, see Section 8.6.3 above.

A convention: in this section, Small letters x, y, and z are used with their usual meaning in designating coordinate directions for specific coordinate systems: i.e., x and y refer to directions in cartesian coordinates, and z may refer to a (linear) direction in either cartesian or cylindrical coordinates.

On the other hand, capital symbols X, Y, and Z are used to refer to the (up to) three directions of any coordinate system, i.e., the directions corresponding to the JAXIS, JAXIS, and KAXIS dimensions in FLASH4, respectively. Only in the cartesian case do these correspond directly to their small-letter counterparts. For other geometries, the correspondence is given in Table 8.7.

8.11.1 Understanding Curvilinear Coordinates

In the context of FLASH, curvilinear coordinates are most useful with 1-d or 2-d simulations, and this is how they are commonly used. But what does it mean to apply curvilinear coordinates in this way? Physical
Table 8.7: Different geometry types. For each geometry/dimensionality combination, the “support” column lists the Grid implementations that support it: pm4 stands for PARAMESH 4.0 and PARAMESH 4dev, pm2 for PARAMESH 2, UG for Uniform Grid implementations, respectively.

<table>
<thead>
<tr>
<th>name</th>
<th>dimensions</th>
<th>support</th>
<th>axisymmetric</th>
<th>X-coord</th>
<th>Y-coord</th>
<th>Z-coord</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian</td>
<td>1</td>
<td>pm4,pm2,UG</td>
<td>n</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cartesian</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>n</td>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>cartesian</td>
<td>3</td>
<td>pm4,pm2,UG</td>
<td>n</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>cylindrical</td>
<td>1</td>
<td>pm4,UG</td>
<td>y</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cylindrical</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>y</td>
<td>r</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>cylindrical</td>
<td>3</td>
<td>pm4,UG</td>
<td>n</td>
<td>r</td>
<td>z</td>
<td>φ</td>
</tr>
<tr>
<td>spherical</td>
<td>1</td>
<td>pm4,pm2,UG</td>
<td>y</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spherical</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>y</td>
<td>r</td>
<td>θ</td>
<td></td>
</tr>
<tr>
<td>spherical</td>
<td>3</td>
<td>pm4,pm2,UG</td>
<td>n</td>
<td>r</td>
<td>θ</td>
<td>φ</td>
</tr>
<tr>
<td>polar</td>
<td>1</td>
<td>pm4,UG</td>
<td>y</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>polar</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>n</td>
<td>r</td>
<td>φ</td>
<td></td>
</tr>
<tr>
<td>&quot;polar + z&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cylindrical with a different ordering of coordinates)</td>
<td>3</td>
<td>—</td>
<td>n</td>
<td>r</td>
<td>φ</td>
<td>z</td>
</tr>
</tbody>
</table>

reality has three spatial dimensions (as far as the physical problems simulated with FLASH are concerned). In cartesian coordinates, it is relatively straightforward to understand what a 2-d (or 1-d) simulation means: “Just leave out one (or two) coordinates.” This is less obvious for other coordinate systems, therefore some fundamental discussion follows.

A reduced dimensionality (RD) simulation can be naively understood as taking a cut (or, for 1-d, a linear probe) through the real 3-d problem. However, there is also an assumption, not always explicitly stated, that is implied in this kind of simulation: namely, that the cut (or line) is representative of the 3-d problem. This can be given a stricter meaning: it is assumed that the physics of the problem do not depend on the omitted dimension (or dimensions). A RD simulation can be a good description of a physical system only to the degree that this assumption is warranted. Depending on the nature of the simulated physical system, non-dependence on the omitted dimensions may mean the absence of force and/or momenta vector components in directions of the omitted coordinate axes, zero net mass and energy flow out of the plane spanned by the included coordinates, or similar.

For omitted dimensions that are lengths — z and possibly y in cartesian, and z in cylindrical and polar RD simulations — one may think of a 2-d cut as representing a (possibly very thin) layer in 3-d space sandwiched between two parallel planes. there is no a priori definition of the thickness of the layer, so it is not determined what 3-d volume should be assigned to a 2-d cell in such coordinates. We can thus arbitrarily assign the length “1” to the edge length of a 3-d cell volume, making the volume equal to the 2-d area. We can understand generalizations of “volume” to 1-d, and of “face areas” to 2-d and 1-d RD simulations with omitted linear coordinates, in an equivalent way: just set the length of cell edges along omitted dimensions to 1.

For omitted dimensions that are angles — the θ and φ coordinates on spherical, cylindrical, and polar geometries — it is easier to think of omitting an angle as the equivalent of integrating over the full range of that angle coordinate (under the assumption that all physical solution variables are independent of that angle). Thus omitting and angle φ in these geometries implies the assumption of axial symmetry, and this is noted in Table 8.7. Similarly, omitting both φ and θ in spherical coordinates implies an assumption of complete spherical symmetry. When φ is omitted, a 2-d cell actually represents the 3-d object that is generated by rotating the 2-d area around a z-axis. Similarly, when only r is included, 1-d cells (i.e., r intervals) represent hollow spheres or cylinders. (If the coordinate interval begins at \( r_l = 0.0 \), the sphere or cylinder is massive instead of hollow.)

As a result of these considerations, the measures for cell (and block) volumes and face areas in a simulation
depends on the chosen geometry. Formulas for the volume of a cell dependent on the geometry are given in the geometry-specific sections further below.

As discussed in Figure 8.6, to ensure conservation at a jump in refinement in AMR grids, a flux correction step is taken. The fluxes leaving the fine cells adjacent to a coarse cell are used to determine more accurately the flux entering the coarse cell. This step takes the coordinate geometry into account in order to accurately determine the areas of the cell faces where fine and coarse cells touch. By way of example, an illustration is provided below in the section on cylindrical geometry.

### 8.11.2 Choosing a Geometry

The user chooses a geometry by setting the `geometry` runtime parameter in `flash.par`. The default is "cartesian" (unless overridden in a simulation’s Config file). Depending on the Grid implementation used and the way it is configured, the geometry may also have to be compiled into the program executable and thus may have to be specified at configuration time; the `setup` flag `geometry` should be used for this purpose, see Section 5.2.

The `geometry` runtime parameter is most useful in cases where the geometry does not have to be specified at compile-time, in particular for the Uniform Grid. The runtime parameter will, however, always be considered at run-time during Grid initialization. If the `geometry` runtime parameter is inconsistent with a geometry specified at setup time, FLASH will then either override the geometry specified at setup time (with a warning) if that is possible, or it will abort.

This runtime parameter is used by the Grid unit and also by hydrodynamics solvers, which add the necessary geometrical factors to the divergence terms. Next we describe how user code can use the runtime parameter’s value.

### 8.11.3 Getting Geometry Information in Program Code

The Grid unit provides an accessor `Grid.getGeometry` property that returns the geometry as an integer, which can be compared to the symbols `{CARTESIAN, SPHERICAL, CYLINDRICAL, POLAR}` defined in "constants.h" to determine which of the supported geometries we are using. A unit writer can therefore determine flow-control based on the geometry type (see Figure 8.14). Furthermore, this provides a mechanism for a unit to determine at runtime whether it supports the current geometry, and if not, to abort.

Coordinate information for the mesh can be determined via the `Grid.getCellCoords` routine. This routine can provide the coordinates of cells at the left edge, right edge, or center. The width of cells can be determined via the `Grid.getDeltas` routine. Angle values and differences are given in radians. Coordinate information for a block of cells as a whole is available through `Grid.getBlkCenterCoords` and `Grid.getBlkPhysicalSize`.

The volume of a single cell can be obtained via the `Grid.getSingleCellVol` or the `Grid.getPointData` routine. Use the `Grid.getBlkData`, `Grid.getPlaneData`, or `Grid.getRowData` routines with argument `dataType=CELL_VOLUME` To retrieve cell volumes for more than one cell of a block. To retrieve cell face areas, use the same `Grid.get*Data` interfaces with the appropriate `dataType` argument.

Note the following difference between the two groups of routines mentioned in the previous two paragraphs: the routines for volumes and areas take the chosen geometry into account in order to return geometric measures of physical volumes and faces (or their RD equivalents). On the other hand, the routines for coordinate values and widths return values for \( X \), \( Y \), and \( Z \) directly, without converting angles to (arc) lengths.

### 8.11.4 Available Geometries

Currently, all of FLASH’s physics support one-, two-, and (with a few exceptions explicitly stated in the appropriate chapters) three-dimensional Cartesian grids. Some units, including the FLASH Grid unit and PPM hydrodynamics unit (Chapter 14), support additional geometries, such as two-dimensional cylindrical \((r, z)\) grids, one/two-dimensional spherical \((r)/(r, \theta)\) grids, and two-dimensional polar \((r, \phi)\) grids. Some specific considerations for each geometry follow.

The following tables use the convention that \( r_l \) and \( r_r \) stand for the values of the \( r \) coordinate at the “left” and “right” end of the cell’s \( r \)-coordinate range, respectively (i.e., \( r_l < r_r \) is always true), and \( \Delta r = r_r - r_l \);


```fortran
#include "constants.h"

integer :: geometry

call Grid_getGeometry(geometry)

select case (geometry)
  case (CARTESIAN)
    ! do Cartesian stuff here ...
  case (SPHERICAL)
    ! do spherical stuff here ...
  case (CYLINDRICAL)
    ! do cylindrical stuff here ...
  case (POLAR)
    ! do polar stuff here ...
end select
```

Figure 8.14: Branching based on geometry type

and similar for the other coordinates.

8.11.4.1 Cartesian geometry

FLASH uses Cartesian (plane-parallel) geometry by default. This is equivalent to specifying

```fortran
geometry = "cartesian"
```

in the runtime parameter file.

<table>
<thead>
<tr>
<th>Cell Volume in Cartesian Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
</tr>
<tr>
<td>2-d</td>
</tr>
<tr>
<td>3-d</td>
</tr>
</tbody>
</table>

8.11.4.2 Cylindrical geometry

To run FLASH with cylindrical coordinates, the `geometry` parameter must be set thus:

```fortran
geometry = "cylindrical"
```
8.11. GRID GEOMETRY

Figure 8.15: Diagram showing two fine cells and a coarse cell at a jump in refinement in the cylindrical ‘z’ direction. The block boundary has been cut apart here for illustrative purposes. The fluxes out of the fine blocks are shown as $f_1$ and $f_2$. These will be used to compute a more accurate flux entering the coarse flux $f_3$. The area that the flux passes through is shown as the annuli at the top of each fine cell and the annulus below the coarse cell.

**Cell Volume in Cylindrical Coordinates**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>$\pi(r_2^2 - r_1^2)$</td>
</tr>
<tr>
<td>2-d</td>
<td>$\pi(r_2^2 - r_1^2)\Delta z$</td>
</tr>
<tr>
<td>3-d</td>
<td>$\frac{1}{2}(r_2^2 - r_1^2)\Delta z\Delta \phi$</td>
</tr>
</tbody>
</table>

As in other non-cartesian geometries, if the minimum radius is chosen to be zero ($x_{\text{min}} = 0.$), the left-hand boundary type should be reflecting. Of all supported non-cartesian geometries, the cylindrical is in 2-d most similar to a 2-d coordinate system: it uses two linear coordinate axes ($r$ and $z$) that form a rectangular grid physically as well as logically.

As an illustrative example of the kinds of considerations necessary in curved coordinates, Figure 8.15 shows a jump in refinement along the cylindrical ‘z’ direction. When performing the flux correction step at a jump in refinement, we must take into account the area of the annulus through which each flux passes to do the proper weighting. We define the cross-sectional area through which the $z$-flux passes as

$$A = \pi(r_2^2 - r_1^2).$$  \hspace{1cm} (8.93)

The flux entering the coarse cell above the jump in refinement is corrected to agree with the fluxes leaving the fine cells that border it. This correction is weighted according to the areas

$$f_3 = \frac{A_1 f_1 + A_2 f_2}{A_3}. \hspace{1cm} (8.94)$$

For fluxes in the radial direction, the cross-sectional area is independent of the height, $z$, so the corrected flux is simply taken as the average of the flux densities in the adjacent finer cells.

8.11.4.3 Spherical geometry

One or two dimensional spherical problems can be performed by specifying

```
game = "spherical"
```

in the runtime parameter file.
CHAPTER 8. GRID UNIT

Cell Volume in Spherical Coordinates

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Volume Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>( \frac{4}{3}\pi(r_2^3 - r_1^3) )</td>
</tr>
<tr>
<td>2-d</td>
<td>( \frac{2}{3}\pi(r_2^3 - r_1^3)(\cos(\theta_2) - \cos(\theta_1)) )</td>
</tr>
<tr>
<td>3-d</td>
<td>( \frac{1}{3}(r_2^3 - r_1^3)(\cos(\theta_2) - \cos(\theta_1))\Delta\phi )</td>
</tr>
</tbody>
</table>

If the minimum radius is chosen to be zero \((x_{min} = 0.)\), the left-hand boundary type should be reflecting.

8.11.4.4 Polar geometry

Polar geometry is a 2-D subset of 3-D cylindrical configuration without the “z” coordinate. Such geometry is natural for studying objects like accretion disks. This geometry can be selected by specifying

\[ \text{geometry = "polar"} \]

in the runtime parameter file.

Cell Volume in Polar Coordinates

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Volume Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>( \pi(r_2^2 - r_1^2) )</td>
</tr>
<tr>
<td>2-d</td>
<td>( \frac{1}{2}(r_2^2 - r_1^2)\Delta\phi )</td>
</tr>
<tr>
<td>3-d</td>
<td>( \frac{1}{2}(r_2^2 - r_1^2)\Delta\phi\Delta z ) (not supported)</td>
</tr>
</tbody>
</table>

As in other non-cartesian geometries, if the minimum radius is chosen to be zero \((x_{min} = 0.)\), the left-hand boundary type should be reflecting.

8.11.5 Conservative Prolongation/Restriction on Non-Cartesian Grids

When blocks are refined, we need to initialize the child data using the information in the parent cell in a manner which preserves the cell-averages in the coordinate system we are using. When a block is derefined, the parent block (which is now going to be a leaf block) needs to be filled using the data in the child blocks (which are soon to be destroyed). The first procedure is called prolongation. The latter is called restriction. Both of these procedures must respect the geometry in order to remain conservative. Prolongation and restriction are also used when filling guard cells at jumps in refinement.

8.11.5.1 Prolongation

When using a supported non-Cartesian geometry, FLASH has to use geometrically correct prolongation routines. These are located in:

- source/Grid/GridMain/paramesh/Paramesh2/monotonic (for PARAMESH 2)
- source/Grid/GridMain/paramesh/interpolation/Paramesh4/prolong (for PARAMESH 4)

These paths will be be automatically added by the setup script when the \(-\text{gridinterpolation=monotonic}\) option is in effect (which is the case by default, unless \(-\text{gridinterpolation=native}\) was specified). The “monotonic” interpolation scheme used in both cases is taking geometry into consideration and is appropriate for all supported geometries.
Some more specific PARAMESH 2 interpolation schemes are included in the distribution and might be useful for compatibility with FLASH2:

- source/Grid/GridMain/paramesh/Paramesh2/quadratic_cartesian (for cartesian coordinates)
- source/Grid/GridMain/paramesh/Paramesh2/quadratic_spherical (for spherical coordinates)

Other geometry types and prolongation schemes can be added in a manner analogous to the ones implemented here. These routines could be included by specifying the correct path in your Units file, or by using appropriate -unit= flags for setup. However, their use is not recommended.

8.11.5.2 Restriction

The default restriction routines understand the supported geometries by default. A cell-volume weighted average is used when restricting the child data up to the parent. For example, in 2-d, the restriction would look like

$$\langle f \rangle_{i,j} = \frac{V_{ic,jc} \langle f \rangle_{ic,jc} + V_{ic+1,jc} \langle f \rangle_{ic+1,jc} + V_{ic,jc+1} \langle f \rangle_{ic,jc+1} + V_{ic+1,jc+1} \langle f \rangle_{ic+1,jc+1}}{V_{i,j}},$$

(8.95)

where $V_{i,j}$ is the volume of the cell with indices, $i, j$, and the $ic, jc$ indices refer to the children.

8.12 Unit Test

The Grid unit test has implementations to test Uniform Grid and PARAMESH. The Uniform Grid version of the test has two parts; the latter portion is also tested in PARAMESH. The test initializes the grid with a sinusoid function $\sin(x) \times \cos(y) \times \cos(z)$, distributed over a number of processors. Knowing the configuration of processors, it is possible to determine the part of the sinusoid on each processor. Since guardcells are filled either from the interior points of the neighboring processor, or from boundary conditions, it is also possible to predict the values expected in guard cells on each processor. The first part of the UG unit test makes sure that the actual received values of guard cell match with the predicted ones. This process is carried out for both cell-centered and face-centered variables.

The second part of the UG test, and the only part of the PARAMESH test, exercises the get and put data functions. Since the Grid unit has direct access to all of its own data structures, it can compare the values fetched using the getData functions against the directly accessible values and report an error if they do not match. The testing of the Grid unit is not exhaustive, and given the complex nature of the unit, it is difficult to devise tests that would do so. However, the more frequently used functions are exercised in this test.
Chapter 9

IO Unit

Figure 9.1: The IO unit: IOMain subunit directory tree.
FLASH uses parallel input/output (IO) libraries to simplify and manage the output of the large amounts of data usually produced. In addition to keeping the data output in a standard format, the parallel IO libraries also ensure that files will be portable across various platforms. The mapping of FLASH data-structures to records in these files is controlled by the FLASH IO unit. FLASH can output data with two parallel IO libraries, HDF5 and Parallel-NetCDF. The data layout is different for each of these libraries. Since FLASH3 we also offer direct serial FORTRAN IO, which can be used as a last resort if no parallel library is available. However, FLASH post-processing tools such as fidlr (Chapter 29) and sfocu (Chapter 27) do not support the direct IO format.

Note:
This release supports both HDF5 and Parallel-NetCDF, including particle IO for both implementations.

Various techniques can be used to write the data to disk when running a parallel simulation. The first is to move all the data to a single processor for output; this technique is known as serial IO. Secondly, each processor can write to a separate file, known as direct IO. As a third option, each processor can use parallel access to write to a single file in a technique known as parallel IO. Finally, a hybrid method can be used where clusters of processors write to the same file, though different clusters of processors output to different files. In general, parallel access to a single file will provide the best parallel IO performance unless the number of processors is very large. On some platforms, such as Linux clusters, there may not be a parallel file system, so moving all the data to a single processor is the only solution. Therefore FLASH supports HDF5 libraries in both serial and parallel forms, where the serial version collects data to one processor before writing it, while the parallel version has every processor writing its data to the same file.
9.1 IO Implementations

FLASH4 supports multiple IO implementations: direct, serial and parallel implementations as well as support for different parallel libraries. In addition, FLASH4 also supports multiple (Chapter 8) Grid implementations. As a consequence, there are many permutations of the IO API implementation, and the selected implementation must match not only the correct IO library, but also the correct grid. Although there are many IO options, the setup script in FLASH4 is quite ‘smart’ and will not let the user setup a problem with incompatible IO and Grid unit implementations. Table 9.1 summarizes the different implementation of the FLASH IO unit in the current release.

Table 9.1: IO implementations available in FLASH. All implementations begin at the /source directory.

<table>
<thead>
<tr>
<th>Implementation Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO/IOMain/HDF5/parallel/PM</td>
<td>Hierarchical Data Format (HDF) 5 output. A single HDF5 file is created, with each processor writing its data to the same file simultaneously. This relies on the underlying MPIIO layer in HDF5. This particular implementation only works with the PARAMESH grid package.</td>
</tr>
<tr>
<td>IO/IOMain/hdf5/parallel/UG</td>
<td>Hierarchical Data Format (HDF) 5 output. A single HDF5 file is created, with each processor writing its data to the same file simultaneously. This relies on the underlying MPIIO layer in HDF5. This particular implementation only works with the Uniform Grid.</td>
</tr>
<tr>
<td>IO/IOMain/hdf5/parallel/NoFbs</td>
<td>Hierarchical Data Format (HDF) 5 output. A single HDF5 file is created, with each processor writing its data to the same file simultaneously. All data is written out as one block. This relies on the underlying MPIIO layer in HDF5. This particular implementation only works in non-fixedblocksize mode.</td>
</tr>
<tr>
<td>IO/IOMain/hdf5/serial/PM</td>
<td>Hierarchical Data Format (HDF) 5 output. Each processor passes its data to processor 0 through explicit MPI sends and receives. Processor 0 does all of the writing. The resulting file format is identical to the parallel version; the only difference is how the data is moved during the writing. This particular implementation only works with the PARAMESH grid package.</td>
</tr>
<tr>
<td>IO/IOMain/hdf5/serial/UG</td>
<td>Hierarchical Data Format (HDF) 5 output. Each processor passes its data to processor 0 through explicit MPI sends and receives. Processor 0 does all of the writing. The resulting file format is identical to the parallel version; the only difference is how the data is moved during the writing. This particular implementation only works with the Uniform Grid.</td>
</tr>
<tr>
<td>IO/IOMain/pnetcdf/PM</td>
<td>ParallelNetCDF output. A single file is created with each processor writing its data to the same file simultaneously. This relies on the underlying MPI-IO layer in PNetCDF. This particular implementation only works with the PARAMESH grid package.</td>
</tr>
</tbody>
</table>
Table 9.1: FLASH IO implementations (continued).

<table>
<thead>
<tr>
<th>Implementation path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO/IOMain/pnetcdf/UG</td>
<td>ParallelNetCDF output. A single file is created with each processor writing its data to the same file simultaneously. This relies on the underlying MPI-IO layer in PNetCDF. This particular implementation only works with the Uniform Grid.</td>
</tr>
<tr>
<td>IO/IOMain/direct/UG</td>
<td>Serial FORTRAN IO. Each processor writes its own data to a separate file. Warning! This choice can lead to many many files! Use only if neither HDF5 or Parallel-NetCDF is available. The FLASH tools are not compatible with the direct IO unit.</td>
</tr>
<tr>
<td>IO/IOMain/direct/PM</td>
<td>Serial FORTRAN IO. Each processor writes its own data to a separate file. Warning! This choice can lead to many many files! Use only if neither HDF5 or Parallel-NetCDF is available. The FLASH tools are not compatible with the direct IO unit.</td>
</tr>
<tr>
<td>IO/IOMain/chombo</td>
<td>Hierarchical Data Format (HDF) 5 output. The subroutines provide an interface to the Chombo I/O routines that are part of the standard Chombo release. This implementation can only be used in applications built with Chombo Grid. The Chombo file layout is incompatible with tools, such as fidlr and sfocu, that depend on FLASH file layout.</td>
</tr>
</tbody>
</table>

FLASH4 also comes with some predefined setup shortcuts which make choosing the correct IO significantly easier; see Chapter 5 for more details about shortcuts. In FLASH4 HDF5 serial IO is included by default. Since PARAMESH 4.0 is the default grid, the included IO implementations will be compatible with PARAMESH 4.0. For clarity, a number or examples are shown below.

An example of a basic setup with HDF5 serial IO and the PARAMESH grid, (both defaults) is:

```
./setup Sod -2d -auto
```

To include a parallel implementation of HDF5 for a PARAMESH grid the `setup` syntax is:

```
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
```

using the already defined shortcuts the `setup` line can be shortened to

```
./setup Sod -2d -auto +parallelio
```

To set up a problem with the Uniform Grid and HDF5 serial IO, the `setup` line is:

```
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/serial/UG
```

using the already defined shortcuts the `setup` line can be shortened to

```
./setup Sod -2d -auto +ug
```

To set up a problem with the Uniform Grid and HDF5 parallel IO, the complete `setup` line is:

```
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/UG
```

using the already defined shortcuts the `setup` line can be shortened to
9.2 Output Files

The IO unit can output 4 different types of files: checkpoint files, plotfiles, particle files and flash.dat, a text file holding the integrated grid quantities. FLASH also outputs a logfile, but this file is controlled by the Logfile Unit; see Chapter 23 for a description of that format.

There are a number of runtime parameters that are used to control the output and frequency of IO files. A list of all the runtime parameters and their descriptions for the IO unit can be found online all of them. Additional description is located in Table 9.2 for checkpoint parameters, Table 9.3 for plotfile parameters, Table 9.4 for particle file parameters, Table 9.5 for flash.dat parameters, and Table 9.6 for general IO parameters.

9.2.1 Checkpoint files - Restarting a Simulation

Checkpoint files are used to restart a simulation. In a typical production run, a simulation can be interrupted for a number of reasons—e.g., if the machine crashes, the present queue window closes, the machine runs out of disk space, or perhaps (gasp) there is a bug in FLASH. Once the problem is fixed, a simulation can be restarted from the last checkpoint file rather than the beginning of the run. A checkpoint file contains all the information needed to restart the simulation. The data is stored at full precision of the code (8-byte reals) and includes all of the variables, species, grid reconstruction data, scalar values, as well as meta-data about the run.

The API routine for writing a checkpoint file is IO_writeCheckpoint. Users usually will not need to call this routine directly because the FLASH IO unit calls IO_writeCheckpoint from the routine IO_output which checks the runtime parameters to see if it is appropriate to write a checkpoint file at this time. There are a number of ways to get FLASH to produce a checkpoint file for restarting. Within the flash.par, runtime parameters can be set to dump output. A checkpoint file can be dumped based on elapsed simulation time, elapsed wall clock time or the number of timesteps advanced. A checkpoint file is also produced when
the simulation ends, when the max simulation time \( t_{\text{max}} \), the minimum cosmological redshift, or the total number of steps \( n_{\text{end}} \) has been reached. A user can force a dump to a checkpoint file at another time by creating a file named \texttt{.dump.checkpoint} in the output directory of the master processor. This manual action causes FLASH to write a checkpoint in the next timestep. Checkpoint files will continue to be dumped after every timestep as long as the code finds a \texttt{.dump.checkpoint} file in the output directory, so the user must remember to remove the file once all the desired checkpoint files have been dumped. Creating a file named \texttt{.dump.restart} in the output directory will cause FLASH to output a checkpoint file and then stop the simulation. This technique is useful for producing one last checkpoint file to save time evolution since the last checkpoint, if the machine is going down or a queue window is about to end. These different methods can be combined without problems. Each counter (number of timesteps between last checkpoint, amount of simulation time since last checkpoint, the change in cosmological redshift, and the amount of wall clock time elapsed since the last checkpoint) is independent of the others, and are not influenced by the use of a \texttt{.dump.checkpoint} or \texttt{.dump.restart}.

Runtime Parameters used to control checkpoint file output include:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpointFileNumber</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of the initial checkpoint file. This number is appended to the end of the filename and incremented at each subsequent output. When restarting a simulation, this indicates which checkpoint file to use.</td>
</tr>
<tr>
<td>checkpointFileIntervalStep</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of timesteps desired between subsequent checkpoint files.</td>
</tr>
<tr>
<td>checkpointFileIntervalTime</td>
<td>REAL</td>
<td>1.</td>
<td>The amount of simulation time desired between subsequent checkpoint files.</td>
</tr>
<tr>
<td>checkpointFileIntervalZ</td>
<td>REAL</td>
<td>HUGE(1.)</td>
<td>The amount of cosmological redshift change that is desired between subsequent checkpoint files.</td>
</tr>
<tr>
<td>rolling_checkpoint</td>
<td>INTEGER</td>
<td>10000</td>
<td>The number of checkpoint files to keep available at any point in the simulation. If a checkpoint number is greater than \texttt{rolling_checkpoint}, then the checkpoint number is reset to 0. There will be at most \texttt{rolling_checkpoint} checkpoint files kept. This parameter is intended to be used when disk space is at a premium.</td>
</tr>
<tr>
<td>wall_clock_checkpoint</td>
<td>REAL</td>
<td>43200</td>
<td>The maximum amount of wall clock time (seconds) to elapse between checkpoints. When the simulation is started, the current time is stored. If \texttt{wall_clock_checkpoint} seconds elapse over the course of the simulation, a checkpoint file is stored. This is useful for ensuring that a checkpoint file is produced before a queue closes.</td>
</tr>
</tbody>
</table>
FLASH is capable of restarting from any of the checkpoint files it produces. The user should make sure that the checkpoint file is valid (e.g., the code did not stop while outputting). To tell FLASH to restart, set the restart runtime parameter to .true. in the flash.par. Also, set checkpointFileNumber to the number of the file from which you wish to restart. If plotfiles or particle files are being produced set plotfileNumber and particleFileNumber to the number of the next plotfile and particle file you want FLASH to output. In FLASH4 plotfiles and particle file outputs are forced whenever a checkpoint file is written. Sometimes several plotfiles may be produced after the last valid checkpoint file. Resetting plotfileNumber to the first plotfile produced after the checkpoint from which you are restarting will ensure that there are no gaps in the output. See Section 9.2.2 for more details on plotfiles.

9.2.2 Plotfiles

A plotfile contains all the information needed to interpret the grid data maintained by FLASH. The data in plotfiles, including the grid metadata such as coordinates and block sizes, are stored at single precision to preserve space. This can, however, be overridden by setting the runtime parameters plotfileMetadataDP and/or plotfileGridQuantityDP to true to set the grid metadata and the quantities stored on the grid (dens, pres, temp, etc.) to use double precision, respectively. Users must choose which variables to output with the runtime parameters plot_var_1, plot_var_2, etc., by setting them in the flash.par file. For example:

plot_var_1 = "dens"
plot_var_2 = "pres"

Currently, we support a number of plotvars named plot_var_n up to the number of UNKVARS in a given simulation. Similarly, scratch variables may be output to plot files Section 9.6. At this time, the plotting of face centered quantities is not supported.

FLASH Transition

In FLASH2 a few variables like density and pressure were output to the plotfiles by default. Because FLASH4 supports a wider range of simulations, it makes no assumptions that density or pressure variables are even included in the simulation. In FLASH4 a user must define plotfile variables in the flash.par file, otherwise the plotfiles will not contain any variables.

The interface for writing a plotfile is the routine IO.writePlotfile. As with checkpoint files, the user will not need to call this routine directly because it is invoked indirectly through calling IO.output when, based on runtime parameters, FLASH4 needs to write a plotfile. FLASH can produce plotfiles in much the same manner as it does with checkpoint files. They can be dumped based on elapsed simulation time, on steps since the last plotfile dump or by forcing a plotfile to be written by hand by creating a.dump.plotfile in the output directory. A plotfile will also be written at the termination of a simulation as well.

If plotfiles are being kept at particular intervals (such as time intervals) for purposes such as visualization or analysis, it is also possible to have FLASH denote a plotfile as “forced”. This designation places
the word forced between the basename and the file format type identifier (or the split number if splitting is used). These files are numbered separately from normal plotfiles. By default, plotfiles are considered forced if output for any reason other than the change in simulation time, change in cosmological redshift, change in step number, or the termination of a simulation from reaching \( n_{\text{end}} \), \( z_{\text{Final}} \), or \( t_{\text{max}} \). This option can be disabled by setting \texttt{ignoreForcedPlot} to true in a simulations \texttt{flash.par} file. The following runtime parameters pertain to controlling plotfiles:

### Table 9.3: Plotfile IO parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotFileNumber</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of the starting (or restarting) plotfile. This number is appended to the file-name.</td>
</tr>
<tr>
<td>plotFileIntervalTime</td>
<td>REAL</td>
<td>1.</td>
<td>The amount of simulation time desired between subsequent plotfiles.</td>
</tr>
<tr>
<td>plotFileIntervalStep</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of timesteps desired between subsequent plotfiles.</td>
</tr>
<tr>
<td>plotFileIntervalZ</td>
<td>INTEGER</td>
<td>( \text{HUGE}(1.) )</td>
<td>The change in cosmological redshift desired between subsequent plotfiles.</td>
</tr>
<tr>
<td>plot_var_1, ..., plot_var_n</td>
<td>STRING</td>
<td>&quot;none&quot;</td>
<td>Name of the variables to store in a plotfile. Up to 12 variables can be selected for storage, and the standard 4-character variable name can be used to select them.</td>
</tr>
<tr>
<td>ignoreForcedPlot</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>A logical variable indicating whether or not to denote certain plotfiles as forced.</td>
</tr>
<tr>
<td>forcedPlotfileNumber</td>
<td>INTEGER</td>
<td>0</td>
<td>An integer that sets the starting number for a forced plotfile.</td>
</tr>
<tr>
<td>plotfileMetadataDP</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>A logical variable indicating whether or not to output the normally single-precision grid metadata fields as double precision in plotfiles. This specifically affects coordinates, block size, and bounding box.</td>
</tr>
<tr>
<td>plotfileGridQuantityDP</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>A logical variable that sets whether or not quantities stored on the grid, such as those stored in \texttt{unk}, are output in single precision or double precision in plotfiles.</td>
</tr>
</tbody>
</table>

#### 9.2.3 Particle files

When Lagrangian particles are included in a simulation, the ParticleIO subunit controls input and output of the particle information. The particle files are stored in double precision. Particle data is written to the checkpoint file in order to restart the simulation, but is not written to plotfiles. Hence analysis and metadata
about particles is also written to the particle files. The particle files are intended for more frequent dumps. The interface for writing the particle file is `IO.writeParticles`. Again the user will not usually call this function directly because the routine `IO.output` controls particle output based on the runtime parameters controlling particle files. They are controlled in much of the same way as the plotfiles or checkpoint files and can be dumped based on elapsed simulation time, on steps since the last particle dump or by forcing a particle file to be written by hand by creating a `.dump.particle_file` in the output directory. The following runtime parameters pertain to controlling particle files:

Table 9.4: Particle File IO runtime parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>particleFileNumber</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of the starting (or restarting) particle file. This number is appended to the end of the filename.</td>
</tr>
<tr>
<td>particleFileIntervalTime</td>
<td>REAL</td>
<td>1.0</td>
<td>The amount of simulation time desired between subsequent particle file dumps.</td>
</tr>
<tr>
<td>particleFileIntervalStep</td>
<td>INTEGER</td>
<td>0</td>
<td>The number of timesteps desired between subsequent particle file dumps.</td>
</tr>
<tr>
<td>particleFileIntervalZ</td>
<td>REAL</td>
<td>HUGE(1.)</td>
<td>The change in cosmological redshift desired between subsequent particle file dumps.</td>
</tr>
</tbody>
</table>

FLASH Transition

From FLASH3 on each particle dump is written to a separate file. In FLASH2 the particles data structure was broken up into real and integer parts, where as in FLASH3 all particle properties are real values. See Section 9.9 and Chapter 19 for more information about the particles data structure in FLASH4. Additionally, filtered particles are not implemented in FLASH4.

All the code necessary to output particle data is contained in the IO subunit called `IOParticles`. Whenever the `Particles` unit is included in a simulation the correct `IOParticles` subunit will also be included. For example as setup:

```
./setup IsentropicVortex -2d -auto -unit=Particles +ug
```

will include the IO unit IO/IOMain/hdf5/serial/UG and the correct `IOParticles` subunit IO/IOParticles/hdf5/serial/UG. The shortcuts `+parallelio`, `+pnetcdf`, `+ug` will also cause the setup script to pick up the correct `IOParticles` subunit as long as a `Particles` unit is included in the simulation.

9.2.4 Integrated Grid Quantities – flash.dat

At each simulation time step, values which represent the overall state (e.g., total energy and momentum) are computed by calculating over all cells in the computations domain. These integral quantities are written to the ASCI file `flash.dat`. A default routine `IO.writeIntegralQuantities` is provided to output standard measures for hydrodynamic simulations. The user should copy and modify the routine
\texttt{IO\_writeIntegralQuantities} into a given simulation directory to store any quantities other than the default values. Two runtime parameters pertaining to the \texttt{flash.dat} file are listed in the table below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{stats_file}</td>
<td>STRING</td>
<td>&quot;flash.dat&quot;</td>
<td>Name of the file to which the integral quantities are written.</td>
</tr>
<tr>
<td>\texttt{wr_integrals_freq}</td>
<td>INTEGER</td>
<td>1</td>
<td>The number of timesteps to elapse between outputs to the scalar/integral data file (\texttt{flash.dat})</td>
</tr>
</tbody>
</table>

### 9.2.5 General Runtime Parameters

There are several runtime parameters that pertain to the general IO unit or multiple output files rather than one particular output file. They are listed in the table below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{basenm}</td>
<td>STRING</td>
<td>&quot;flash.&quot;</td>
<td>The main part of the output filenames. The full filename consists of the base name, a series of three-character abbreviations indicating whether it is a plotfile, particle file or checkpoint file, the file format, and a 4-digit file number. See Section 9.8 for a description of how FLASH output files are named.</td>
</tr>
<tr>
<td>\texttt{output_directory}</td>
<td>STRING</td>
<td>&quot;&quot;</td>
<td>Output directory for plotfiles, particle files and checkpoint files. The default is the directory in which the executable sits. \texttt{output_directory} can be an absolute or relative path.</td>
</tr>
<tr>
<td>\texttt{memory_stat_freq}</td>
<td>INTEGER</td>
<td>100000</td>
<td>The number of timesteps to elapse between memory statistic dumps to the log file (\texttt{flash.log}).</td>
</tr>
<tr>
<td>\texttt{useCollectiveHDF5}</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>When using the parallel HDF5 implementation of IO, will enable collective mode for HDF5.</td>
</tr>
<tr>
<td>\texttt{summaryOutputOnly}</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>When set to .true, write an integrated grid quantities file only. Checkpoint, plot and particle files are not written unless the user creates a .dump_plotfile, .dump_checkpoint, .dump_restart or .dump_particle file.</td>
</tr>
</tbody>
</table>
9.3 Restarts and Runtime Parameters

FLASH4 outputs the runtime parameters of a simulation to all checkpoint files. When a simulation is restarted, these values are known by the RuntimeParameters unit while the code is running. On a restart, all values from the checkpoint used in the restart are stored as previous values in the lists kept by the RuntimeParameters unit. All current values are taken from the defaults used by FLASH4 and any simulation parameter files (e.g., flash.par). If needed, the previous values from the checkpoint file can be obtained using the routines RuntimeParameters.getPrev.

9.4 Output Scalars

In FLASH4, each unit has the opportunity to request scalar data to be output to checkpoint or plotfiles. Because there is no central database, each unit “owns” different data in the simulation. For example, the Driver unit owns the timestep variable dt, the simulation variable simTime, and the simulation step number nStep. The Grid unit owns the sizes of each block, nxb, nyb, and nzb. The IO unit owns the variable checkpointFileNumber. Each of these quantities are output into checkpoint files. Instead of hard coding the values into checkpoint routines, FLASH4 offers a more flexible interface whereby each unit sends its data to the IO unit. The IO unit then stores these values in a linked list and writes them to the checkpoint file or plotfile. Each unit has a routine called “Unit_sendOutputData”, e.g., Driver_sendOutputData and Grid_sendOutputData. These routines in turn call IO.setScalar. For example, the routine Grid_sendOutputData calls

IO.setScalar("nxb", NXB)
IO.setScalar("nyb", NYB)
IO.setScalar("nzb", NZB)

To output additional simulation scalars in a checkpoint file, the user should override one of the “Unit_sendOutputData” or Simulation_sendOutputData.

After restarting a simulation from a checkpoint file, a unit might call IO.getScalar to reset a variable value. For example, the Driver unit calls IO.getScalar("dt", dr_dt) to get the value of the timestep dt reinitialized from the checkpoint file. A value from the checkpoint file can be obtained by calling IO.getPrevScalar. This call can take an optional argument to find out if an error has occurred in finding the previous value, most commonly because the value was not found in the checkpoint file. By using this argument, the user can then decide what to do if the value is not found. If the scalar value is not found and the optional argument is not used, then the subroutine will call Driver_abortFlash and terminate the run.

9.5 Output User-defined Arrays

Often in a simulation the user needs to output additional information to a checkpoint or plotfile which is not a grid scope variable. In FLASH2 any additional information had to be hard coded into the simulation. In FLASH4, we have provided a general interface IO.writeUserArray and IO.readUserArray which allows the user to write and read any generic array needed to be stored. The above two functions do not have any implementation and it is up to the user to fill them in with the needed calls to the HDF5 or pnetCDF C routines. We provide implementation for reading and writing integer and double precision arrays with the helper routines io_h5write_generic_iarr, io_h5write_generic_rarr, io_ncmpi_write_generic_iarr, and io_ncmpi_write_generic_rarr. Data is written out as a 1-dimensional array, but the user can write multidimensional arrays simply by passing a reference to the data and the total number of elements to write. See these routines and the simulation StirTurb for details on their usage.
9.6 Output Scratch Variables

In FLASH4 a user can allocate space for a scratch or temporary variable with grid scope using one of the Config keywords `SCRATCHVAR`, `SCRATCHCENTERVAR`, `SCRATCHFACEVAR`, `SCRATCHFACEYVAR` or `SCRATCHFACEZVAR` (see Section 5.5.1). To output these scratch variables, the user only needs to set the values of the runtime parameters `plot_grid_var_1`, `plot_grid_var_2`, etc., by setting them in the `flash.par` file. For example to output the magnitude of vorticity with a declaration in a Config file of `SCRATCHVAR mvrt`:

```plaintext
plot_grid_var_1 = "mvrt"
```

Note that the post-processing routines like `fidlr` do not display these variables, although they are present in the output file. Future implementations may support this visualization.

9.7 Face-Centered Data

Face-centered variables are now output to checkpoint files, when they are declared in a configuration file. Presently, up to nine face-centered variables are supported in checkpoint files. Plotfile output of face-centered data is not yet supported.

9.8 Output Filenames

FLASH constructs the output filenames based on the user-supplied basename, (runtime parameter `basemn`) and the file counter that is incremented after each output. Additionally, information about the file type and data storage is included in the filename. The general checkpoint filename is:

```
basename_s0000_{hdf5 ncmi}_chk_0000 ,
```

where `hdf5` or `ncmi` (prefix for PnetCDF) is picked depending on the particular IO implementation, the number following the “s” is the split file number, if split file IO is in use, and the number at the end of the filename is the current checkpointFileNumber. (The PnetCDF function prefix "ncmi" derived from the serial NetCDF calls beginning with "nc")

The general plotfile filename is:

```
basename_s0000_{hdf5 ncmi}_plt_{crn cnt}_0000 ,
```

where `hdf5` or `ncmi` is picked depending on the IO implementation used, `crn` and `cnt` indicate data stored at the cell corners or centers respectively, the number following “s” is the split file number, if used, and the number at the end of the filename is the current value of `plotfileNumber`. `crn` is reserved, even though corner data output is not presently supported by FLASH4’s IO.

FLASH Transition

In FLASH2 the correct format of the names of the checkpoint, plotfile and particle file were necessary in order to read the files with the FLASH fidlr visualization tool. In FLASH4 the name of the file is irrelevant to fidlr3.0 (see Chapter 29). We have kept the same naming convention for consistency but the user is free to rename files. This can be helpful during post-processing or when comparing two files.
9.9 Output Formats

HDF5 is our most widely used IO library although Parallel-NetCDF is rapidly gaining acceptance among the high performance computing community. In FLASH4 we also offer a serial direct FORTRAN IO which is currently only implemented for the uniform grid. This option is intended to provide users a way to output data if they do not have access to HDF5 or PnetCDF. Additionally, if HDF5 or PnetCDF are not performing well on a given platform the direct IO implementation can be used as a last resort. Our tools, fidlr and sfocu (Part VIII), do not currently support the direct IO implementation, and the output files from this mode are not portable across platforms.

9.9.1 HDF5

HDF5 is supported on a large variety of platforms and offers large file support and parallel IO via MPI-IO. Information about the different versions of HDF can be found at http://www.ncsa.illinois.edu/. The IO in FLASH4 implementations require HDF5 1.4.0 or later. Please note that HDF5 1.6.2 requires IDL 1.6 or higher in order to use fidlr3.0 for post processing.

Implementations of the HDF5 IO unit use the HDF application programming interface (API) for organizing data in a database fashion. In addition to the raw data, information about the data type and byte ordering (little- or big-endian), rank, and dimensions of the dataset is stored. This makes the HDF format extremely portable across platforms. Different packages can query the file for its contents without knowing the details of the routine that generated the data.

FLASH provides different HDF5 IO unit implementations – the serial and parallel versions for each supported grid, Uniform Grid and PARAMESH. It is important to remember to match the IO implementation with the correct grid, although the setup script generally takes care of this matching. PARAMESH 2, PARAMESH 4.0, and PARAMESH 4dev all work with the PARAMESH (PM) implementation of IO. Nonfixed blocksize IO has its own implementation in parallel, and is presently not supported in serial mode. Examples are given below for the five different HDF5 IO implementations.

```
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/serial/PM (included by default)
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/serial/UG
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/UG
./setup Sod -2d -auto -nofbs -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/NoFbs
```

The default IO implementation is IO/IOMain/hdf5/serial/PM. It can be included simply by adding -unit=IO to the setup line. In FLASH4, the user can set up shortcuts for various implementations. See Chapter 5 for more information about creating shortcuts.

The format of the HDF5 output files produced by these various IO implementations is identical; only the method by which they are written differs. It is possible to create a checkpoint file with the parallel routines and restart FLASH from that file using the serial routines or vice-versa. (This switch would require resetting up and compiling a code to get an executable with the serial version of IO.) When outputting with the Uniform Grid, some data is stored that isn’t explicitly necessary for data analysis or visualization, but is retained to keep the output format of PARAMESH the same as with the Uniform Grid. See Section 9.9.1.3 for more information on output data formats. For example, the refinement level in the Uniform Grid case is always equal to 1, as is the nodetype array. A tree structure for the Uniform Grid is ‘faked’ for visualization purposes. In a similar way, the non-fixedblocksize mode outputs all of the data stored by the grid as though it is one large block. This allows restarting with differing numbers of processors and decomposing the domain in an arbitrary fashion in Uniform Grid.

Parallel HDF5 mode has two runtime parameters useful for debugging: chkGuardCellsInput and chkGuardCellsOutput. When these runtime parameters are true, the FLASH4 input and output routines read and/or output the guard cells in addition to the normal interior cells. Note that the HDF5 files produced are not compatible with the visualization and analysis tools provided with FLASH4.
9.9.1.1 Collective Mode

By default, the parallel mode of HDF5 uses an independent access pattern for writing datasets and performs IO without aggregating the disk access for writing. Parallel HDF5 can also be run so that the writes to the file’s datasets are aggregated, allowing the data from multiple processors to be written to disk in fewer operations. This can greatly increase the performance of IO on filesystems that support this behavior. FLASH4 can make use of this mode by setting the runtime parameter useCollectiveHDF5 to true.

**FLASH Transition**

We recommend that HDF5 version 1.6 or later be used with the HDF5 IO implementations with FLASH4. While it is possible to use any version of HDF5 1.4.0 or later, files produced with versions predating version 1.6 will not be compatible with code using the libraries post HDF5 1.6.

**Caution**

If you are using version HDF5 > = 1.8 then you must explicitly use HDF5 1.6 API bindings. Either build HDF5 library with “–with-default-api-version=v16” configure option or compile FLASH with the C preprocessor definition H5_USE_16_API. Our preference is to set CFLAGS_HDF5 Makefile.h variable, e.g., for a GNU compilation:

```
CFLAGS_HDF5 = -I${HDF5_PATH}/include -DH5_USE_16_API
```

9.9.1.2 Machine Compatibility

The HDF5 modules have been tested successfully on the ASC platforms and on a Linux clusters. Performance varies widely across the platforms, but the parallel version is usually faster than the serial version. Experience on performing parallel IO on a Linux Cluster using PVFS is reported in Ross et al. (2001). Note that for clusters without a parallel filesystem, you should not use the parallel HDF5 IO module with an NFS mounted filesystem. In this case, all of the information will still have to pass through the node from which the disk is hanging, resulting in contention. It is recommended that a serial version of the HDF5 unit be used instead.

9.9.1.3 HDF5 Data Format

The HDF5 data format for FLASH4 is identical to FLASH2 for all grid variables and datastructures used to recreate the tree and neighbor data with the exception that bounding box, coordinates, and block size are now sized as *mdim*, or the maximum dimensions supported by FLASH’s grids, which is three, rather than *ndim*. PARAMESH 4.0 and PARAMESH 4dev, however, do require a few additional tree data structures to be output which are described below. The format of the metadata stored in the HDF5 files has changed to reduce the number of ‘writes’ required. Additionally, scalar data, like *time*, *dt*, *nstep*, etc., are now stored in a linked list and written all at one time. Any unit can add scalar data to the checkpoint file by calling the routine *IO_setScalar*. See Section 9.4 for more details. The FLASH4 HDF5 format is summarized in Table 9.7.

**Table 9.7: FLASH HDF5 file format.**

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Meta Data: included in all files</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim info</td>
<td>Stores simulation meta data in a user defined C structure. Structure datatype and attributes of the structure are described below.</td>
</tr>
</tbody>
</table>

```c
typedef struct sim_info_t {
    int file_format_version;
    char setup_call[400];
    char file_creation_time[MAX_STRING_LENGTH];
    char flash_version[MAX_STRING_LENGTH];
    char build_date[MAX_STRING_LENGTH];
    char build_dir[MAX_STRING_LENGTH];
    char build_machine[MAX_STRING_LENGTH];
    char cflags[400];
    char fflags[400];
    char setup_time_stamp[MAX_STRING_LENGTH];
    char build_time_stamp[MAX_STRING_LENGTH];
} sim_info_t;

sim_info_t sim_info;
```

- **sim_info.file_format_version**: An integer giving the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file.

- **sim_info.setup_call**: The complete syntax of the `setup` command used when creating the current FLASH executable.

- **sim_info.file_creation_time**: The time and date that the file was created.

- **sim_info.flash_version**: The version of FLASH used for the current simulation. This is returned by routine `setup_flashVersion`.

- **sim_info.build_date**: The date and time that the FLASH executable was compiled.

- **sim_info.build_dir**: The complete path to the FLASH root directory of the source tree used when compiling the FLASH executable. This is generated by the subroutine `setup_buildstats` which is created at compile time by the Makefile.

- **sim_info.build_machine**: The name of the machine (and anything else returned from `uname -a`) on which FLASH was compiled.

- **sim_info.cflags**: The c compiler flags used in the given simulation. The routine `setup_buildstats` is written by the `setup` script at compile time and also includes the `fflags` below.

- **sim_info.fflags**: The f compiler flags used in the given simulation.
Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_info.setup_time_stamp:</td>
<td>The date and time the given simulation was setup. The routine <code>setup_buildstamp</code> is created</td>
</tr>
<tr>
<td></td>
<td>by the <code>setup</code> script at compile time.</td>
</tr>
<tr>
<td>sim_info.build_time_stamp:</td>
<td>The date and time the given simulation was built. The routine <code>setup_buildstamp</code> is created</td>
</tr>
<tr>
<td></td>
<td>by the <code>setup</code> script at compile time.</td>
</tr>
</tbody>
</table>

*RuntimeParameter and Scalar data*

Data are stored in linked lists with the nodes of each entry for each type listed below.

```c
typedef struct int_list_t {
    char name[MAX_STRING_LENGTH];
    int value;
} int_list_t;

typedef struct real_list_t {
    char name[MAX_STRING_LENGTH];
    double value;
} real_list_t;

typedef struct str_list_t {
    char name[MAX_STRING_LENGTH];
    char value[MAX_STRING_LENGTH];
} str_list_t;

typedef struct log_list_t {
    char name[MAX_STRING_LENGTH];
    int value;
} log_list_t;

int_list_t *int_list;
real_list_t *real_list;
str_list_t *str_list;
log_list_t *log_list;
```

**integer runtime parameters**

A linked list holding the names and values of all the integer runtime parameters.

**real runtime parameters**

A linked list holding the names and values of all the real runtime parameters.

**string runtime parameters**

A linked list holding the names and values of all the string runtime parameters.
### 9.9. OUTPUT FORMATS

#### Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical runtime parameters</td>
<td><code>log_list_t log_list(numLogParams)</code></td>
</tr>
<tr>
<td></td>
<td>A linked list holding the names and values of all the logical runtime parameters.</td>
</tr>
<tr>
<td>integer scalars</td>
<td><code>int_list_t int_list(numIntScalars)</code></td>
</tr>
<tr>
<td></td>
<td>A linked list holding the names and values of all the integer scalars.</td>
</tr>
<tr>
<td>real scalars</td>
<td><code>real_list_t real_list(numRealScalars)</code></td>
</tr>
<tr>
<td></td>
<td>A linked list holding the names and values of all the real scalars.</td>
</tr>
<tr>
<td>string scalars</td>
<td><code>str_list_t str_list(numStrScalars)</code></td>
</tr>
<tr>
<td></td>
<td>A linked list holding the names and values of all the string scalars.</td>
</tr>
<tr>
<td>logical scalars</td>
<td><code>log_list_t log_list(numLogScalars)</code></td>
</tr>
<tr>
<td></td>
<td>A linked list holding the names and values of all the logical scalars.</td>
</tr>
</tbody>
</table>

**Grid data: included only in checkpoint files and plotfiles**

| unknown names            | `character*4 unk_names(nvar)`                                                            |
|                         | This array contains four-character names corresponding to the first index of the `unk` array. They serve to identify the variables stored in the ‘unknowns’ records. |
| refine level             | `integer lrefine(globalNumBlocks)`                                                      |
|                         | This array stores the refinement level for each block.                                  |
| node type                | `integer nodetype(globalNumBlocks)`                                                     |
|                         | This array stores the node type for a block. Blocks with node type 1 are leaf nodes, and their data will always be valid. The leaf blocks contain the data which is to be used for plotting purposes. |
| gid                      | `integer gid(nfaces+1+nchild,globalNumBlocks)`                                          |
|                         | This is the global identification array. For a given block, this array gives the block number of the blocks that neighbor it and the block numbers of its parent and children. |
| coordinates              | `real coord(mdim,globalNumBlocks)`                                                      |
|                         | This array stores the coordinates of the center of the block.                           |
|                         | `coord(1,blockID) = x-coordinate`                                                        |
|                         | `coord(2,blockID) = y-coordinate`                                                        |
|                         | `coord(3,blockID) = z-coordinate`                                                        |
Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>block size</td>
<td>real size(mdim,globalNumBlocks)</td>
</tr>
<tr>
<td></td>
<td>This array stores the dimensions of the current block.</td>
</tr>
<tr>
<td></td>
<td>size(1,blockID) = x size</td>
</tr>
<tr>
<td></td>
<td>size(2,blockID) = y size</td>
</tr>
<tr>
<td></td>
<td>size(3,blockID) = z size</td>
</tr>
<tr>
<td>bounding box</td>
<td>real bnd_box(2,mdim,globalNumBlocks)</td>
</tr>
<tr>
<td></td>
<td>This array stores the minimum (bnd_box(1,:,:)) and maximum (bnd_box(2,:,:)) coordinate of a block in each spatial direction.</td>
</tr>
<tr>
<td>which child</td>
<td>integer which_child(globalNumBlocks)</td>
</tr>
<tr>
<td></td>
<td>An integer array identifying which part of the parents’ volume this child corresponds to.</td>
</tr>
<tr>
<td>variable</td>
<td>real unk(nxb,nyb,nzb,globalNumBlocks)</td>
</tr>
<tr>
<td></td>
<td>nx = number of cells/block in x</td>
</tr>
<tr>
<td></td>
<td>ny = number of cells/block in y</td>
</tr>
<tr>
<td></td>
<td>nz = number of cells/block in z</td>
</tr>
<tr>
<td></td>
<td>This array holds the data for a single variable. The record label is identical to the four-character variable name stored in the record unknown names.</td>
</tr>
<tr>
<td></td>
<td>Note that, for a plot file with CORNERS=.true. in the parameter file, the information is interpolated to the cell corners and stored.</td>
</tr>
<tr>
<td>Particle Data: included in checkpoint files and particle files</td>
<td><strong>localnp</strong></td>
</tr>
<tr>
<td></td>
<td>integer localnp(globalNumBlocks)</td>
</tr>
<tr>
<td></td>
<td>This array holds the number of particles on each processor.</td>
</tr>
<tr>
<td>particle names</td>
<td>character*24 particle_labels(NPART_PROPS)</td>
</tr>
<tr>
<td></td>
<td>This array contains twenty four-character names corresponding to the attributes in the particles array. They serve to identify the variables stored in</td>
</tr>
<tr>
<td></td>
<td>the ‘particles’ record.</td>
</tr>
<tr>
<td>tracer particles</td>
<td>real particles(NPART_PROPS, globalNumParticles)</td>
</tr>
<tr>
<td></td>
<td>Real array holding the particles data structure. The first dimension holds the various particle properties like, velocity, tag etc. The second dimension</td>
</tr>
<tr>
<td></td>
<td>is sized as the total number of particles in the simulation. Note that all the particle properties are real values.</td>
</tr>
</tbody>
</table>
9.9. OUTPUT FORMATS

9.9.1.4 Split File IO

On machines with large numbers of processors, IO may perform better if, all processors write to a limited number of separate files rather than one single file. This technique can help mitigate IO bottlenecks and contention issues on these large machines better than even parallel-mode IO can. In addition this technique has the benefit of keeping the number of output files much lower than if every processor writes its own file. Split file IO can be enabled by setting the outputSplitNum parameter to the number of files desired (i.e. if outputSplitNum is set to 4, every checkpoint, plotfile and parfile will be broken into 4 files, by processor number). This feature is only available with the HDF5 parallel IO mode, and is still experimental. Users should use this at their own risk.

9.9.2 Parallel-NetCDF

Another implementation of the IO unit uses the Parallel-NetCDF library available at http://www.mcs.anl.gov/parallel-netcdf/. At this time, the FLASH code requires version 1.1.0 or higher. Our testing shows performance of PNetCDF library to be very similar to HDF5 library when using collective I/O optimizations in parallel I/O mode.

There are two different PnetCDF IO unit implementations. Both are parallel implementations, one for each supported grid, the Uniform Grid and PARAMESH. It is important to remember to match the IO implementation with the correct grid. To include PnetCDF IO in a simulation the user should add -unit=IO/IOMain/pnetcdf..... to the setup line. See examples below for the two different PnetCDF IO implementations.

```
./setup Sod -2d -auto -unit=IO/IOMain/pnetcdf/PM
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/pnetcdf/UG
```

The paths to these IO implementations can be long and tedious to type, users are advised to set up shortcuts for various implementations. See Chapter 5 for information about creating shortcuts.

To the end-user, the PnetCDF data format is very similar to the HDF5 format. (Under the hood the data storage is quite different.) In HDF5 there are datasets and dataspaces, in PnetCDF there are dimensions and variables. All the same data is stored in the PnetCDF checkpoint as in the HDF5 checkpoint file, although there are some differences in how the data is stored. The grid data is stored in multidimensional arrays, as it is in HDF5. These are unknown names, refine level, node type, gid, coordinates, proc number, block size and bounding box. The particles data structure is also stored in the same way. The simulation metadata, like file format version, file creation time, setup command line, etc., are stored as global attributes. The runtime parameters and the output scalars are also stored as attributes. The unk and particle labels are also stored as global attributes. In PnetCDF, all global quantities must be consistent across all processors involved in a write to a file, or else the write will fail. All IO calls are run in a collective mode in PnetCDF.

9.9.3 Direct IO

As mentioned above, the direct IO implementation has been added so users can always output data even if the HDF5 or pnetCDF libraries are unavailable. The user should examine the two helper routines io_writeData and io_readData. Copy the base implementation to a simulation directory, and modify them in order to write out specifically what is needed. To include the direct IO implementation add the following to your setup line:

```
-unit=IO/IOMain/direct/UG or -unit=IO/IOMain/direct/PM
```

9.9.4 Output Side Effects

In FLASH4 when plotfiles or checkpoint files are output by IO_output, the grid is fully restricted and user variables are computed prior to writing the file. IO_writeCheckpoint and IO_writePlotfile by default, do not do this step themselves. The restriction can be forced for all writes by setting runtime parameter alwaysRestrictCheckpoint to true and the user variables can always be computed prior to output by setting alwaysComputeUserVars to true.
9.10 Working with Output Files

The checkpoint file output formats offer great flexibility when visualizing the data. The visualization program does not have to know the details of how the file was written; rather it can query the file to find the number of dimensions, block sizes, variable data etc that it needs to visualize the data. IDL routines for reading HDF5 and PnetCDF formats are provided in tools/fidlr3/. These can be used interactively though the IDL command line (see Chapter 29). In addition, VisIt version 10.0 and higher (see Chapter 26) can natively read FLASH4 HDF5 output files by using the command line option -assume_format FLASH.

9.11 Unit Test

The IO unit test is provided to test IO performance on various platforms with the different FLASH IO implementations and parallel libraries.

FLASH Transition

The IO unit test replaces the simulation setup io_benchmark in FLASH2.

The unitTest is setup like any other FLASH4 simulation. It can be run with any IO implementation as long as the correct Grid implementation is included. This unitTest writes a checkpoint file, a plotfile, and if particles are included, a particle file. Particles IO can be tested simply by including particles in the simulation. Variables needed for particles should be uncommented in the Config file.

Example setups:

```
#setup for PARAMESH Grid and serial HDF5 io
./setup unitTest/IO -auto

#setup for PARAMESH Grid with parallel HDF5 IO (see shortcuts docs for explanation)
./setup unitTest/IO -auto +parallelIO  (same as)
./setup unitTest/IO -auto -unit=IO/IOMain/hdf5/parallel/PM

#setup for Uniform Grid with serial HDF5 IO, 3d problem, increasing default number of zones
./setup unitTest/IO -3d -auto +ug -nxb=16 -nyb=16 -nzb=16  (same as)
./setup unitTest/IO -3d -auto -unit=Grid/GridMain/UG -nxb=16 -nyb=16 -nzb=16

#setup for PM3 and parallel netCDF, with particles
./setup unitTest/IO -auto -unit=Particles +pnetcdf

#setup for UG and parallel netCDF
./setup unitTest/IO -auto +pnetcdf +ug
```

Run the test like any other FLASH simulation:

```
mpirun -np numProcs flash3
```

There are a few things to keep in mind when working with the IO unit test:

- The Config file in unitTest/IO declares some dummy grid scope variables which are stored in the unk array. If the user wants a more intensive IO test, more variables can be added. Variables are initialized to dummy values in Driver_evolveFlash.
- Variables will only be output to the plotfile if they are declared in the flash.par (see the example flash.par in the unit test).
• The only units besides the simulation unit included in this simulation are Grid, IO, Driver, Timers, Logfile, RuntimeParameters and PhysicalConstants.

• If the PARAMESH Grid implementation is being used, it is important to note that the grid will not refine on its own. The user should set lrefine_min to a value > 1 to create more blocks. The user could also set the runtime parameters nblockx, nblocky, nblockz to make a bigger problem.

• Just like any other simulation, the user can change the number of zones in a simulation using -nxb= numZones on the setup line.

9.12 Chombo

In FLASH4 we introduced a new Grid implementation that depends on Chombo library (see Section 8.7). An application built with this Grid implementation will include IO/IOMain/chombo by default. No other I/O implementation is compatible with Chombo Grid implementation. As before, I/O can be excluded from an application using the +noio setup shortcut.

Data files are written in standard Chombo file layout in HDF5 format. The visualization tool VisIt supports Chombo file layout, but other tools, such as fidlr and sfocu, are incompatible with Chombo file layout.

9.13 Derived data type I/O

In FLASH4 we introduced an alternative I/O implementation for both HDF5 and Parallel-NetCDF which is a slight spin on the standard parallel I/O implementations. In this new implementation we select the data from the mesh data structures directly using HDF5 hyperslabs (HDF5) and MPI derived datatypes (Parallel-NetCDF) and then write the selected data to datasets in the file. This eliminates the need for manually copying data into a FLASH allocated temporary buffer and then writing the data from the temporary buffer to disk.

You can include derived data type I/O in your FLASH application by adding the setup shortcuts +hdf5TypeIO for HDF5 and +pnetTypeIO for Parallel-NetCDF to your setup line. If you are using the HDF5 implementation then you need a parallel installation of HDF5. All of the runtime parameters introduced in this chapter should be compatible with derived data type I/O.

A nice property of derived data type I/O is that it eliminates a lot of the I/O code duplication which has been spreading in the FLASH I/O unit over the last decade. The same code is used for UG, NoFBS and Paramesh FLASH applications and we have also shared code between the HDF5 and Parallel-NetCDF implementations. A technical reason for using the new I/O implementation is that we provide more information to the I/O libraries about the exact data we want to read from / write to disk. This allows us to take advantage of recent enhancements to I/O libraries such as the nonblocking APIs in the Parallel-NetCDF library. We discuss experimentation with this API and other ideas in the paper “A Case Study for Scientific I/O: Improving the FLASH Astrophysics Code” www.mcs.anl.gov/uploads/cels/papers/P1819.pdf

The new I/O code has been tested in our internal FLASH regression tests from before the FLASH4 release and there are no known issues, however, it will probably be in the release following FLASH4 when we will recommend using it as the default implementation. We have made the research ideas from our case study paper usable for all FLASH applications, however, the code still needs a clean up and exhaustive testing with all the FLASH runtime parameters introduced in this chapter.
Chapter 10

Runtime Parameters Unit

Figure 10.1: The RuntimeParameters unit directory tree.

The RuntimeParameters Unit stores and maintains a global linked lists of runtime parameters that are used during program execution. Runtime parameters can be added to the lists, have their values modified, and be queried. This unit handles adding the default runtime parameters to the lists as well as reading any overwritten parameters from the flash.par file.

10.1 Defining Runtime Parameters

All parameters must be declared in a Config file with the keyword declaration PARAMETER. In the Config file, assign a data type and a default value for the parameter. If possible, assign a range of valid values for the parameter. You can also provide a short description of the parameter’s function in a comment line that begins with D.

```
#section of Config file for a Simulation
D myParameter Description of myParameter
PARAMETER myParameter REAL 22.5 [20 to 60]
```

To change the runtime parameter's value from the default, assign a new value in the flash.par for the simulation.

```
#snippet from a flash.par
myParameter = 45.0
```

See Section 5.5 for more information on declaring parameters in a Config file.

10.2 Identifying Valid Runtime Parameters

The values of runtime parameters are initialized either from default values defined in the Config files, or from values explicitly set in the file flash.par. Variables that have been changed from default are noted in the
CHAPTER 10. RUNTIME PARAMETERS UNIT

---

RuntimeParameters:
---

- `pt_numx` = 10 [CHANGED]
- `pt_numy` = 5 [CHANGED]
- `checkpointFileIntervalStep` = 0

---

Figure 10.2: Section of output log showing runtime parameters values

physics/Eos/EosMain/Multigamma
  - `gamma` [REAL] [1.6667]
    - Valid Values: Unconstrained
    - Ratio of specific heats for gas

physics/Hydro/HydroMain
  - `cfl` [REAL] [0.8]
    - Valid Values: Unconstrained
    - Courant factor

---

Figure 10.3: Portion of a `setup.params` file from an object directory.

simulation’s output log. For example, the RuntimeParameters section of the output log shown in Figure 10.2 indicates that `pt_numx` and `pt_numy` have been read in from `flash.par` and are different than the default values, whereas the runtime parameter `checkpointFileIntervalStep` has been left at the default value of 0.

After a simulation has been configured with a `setup` call, all possible valid runtime parameters are listed in the file `setup.params` located in the `object` directory (or whatever directory was chosen with `-objdir=`) with their default values. This file groups the runtime parameters according to the units with which they are associated and in alphabetical order. A short description of the runtime parameter, and the valid range or values if known, are also given. See Figure 10.3 for an example listing.

10.3 Routine Descriptions

The Runtime Parameters unit is included by default in all of the provided FLASH simulation examples, through a dependence within the Driver unit. The main FLASH initialization routine (`Driver_initFlash`) and the initialization code created by `setup` handles the creation and initialization of the runtime parameters, so users will mainly be interested in querying parameter values. Because the RuntimeParameters routines are overloaded functions which can handle character, real, integer, or logical arguments, the user must make sure to use the interface file `RuntimeParameters_Interfaces` in the calling routines.

The user will typically only have to use one routine from the Runtime Parameters API, `RuntimeParameters.get`. This routine retrieves the value of a parameter stored in the linked list in the `RuntimeParameters_data` module. In FLASH4 the value of runtime parameters for a given unit are stored in that unit’s `Unit_data` Fortran module and they are typically initialized in the unit’s `Unit_init` routine. Each unit’s ‘init’ routine is only called once at the beginning of the simulation by `Driver_initFlash`. For more documentation on the FLASH code architecture please see Chapter 4. It is important to note that even though runtime parameters are declared in a specific unit’s `Config` file, the runtime parameters linked
list is a global space and so any unit can fetch a parameter, even if that unit did not declare it. For example, the Driver unit declares the logical parameter restart, however, many units, including the IO unit get restart parameter with the RuntimeParameters_get interface. If a section of the code asks for a runtime parameter that was not declared in a Config file and thus is not in the runtime parameters linked list, the FLASH code will call Driver_abortFlash and stamp an error to the logfile. The other RuntimeParameter routines in the API are not generally called by user routines. They exist because various other units within FLASH need to access parts of the RuntimeParameters interface. For example, the input/output unit IO needs RuntimeParameters_set. There are no user-defined parameters which affect the RuntimeParameters unit.

FLASH Transition

FLASH no longer distinguishes between contexts as in FLASH2. All runtime parameters are stored in the same context, so there is no need to pass a ‘context’ argument.

10.4 Example Usage

An implementation example from the IO_init is straightforward. First, use the module containing definitions for the unit (for _init subroutines, the usual use Unit_data, ONLY: structure is waived). Next, use the module containing interface definitions of the RuntimeParameters unit, i.e., use RuntimeParameters_interface, ONLY:. Finally, read the runtime parameters and store them in unit-specific variables.

subroutine IO_init()

use IO_data
use RuntimeParameters_interface, ONLY : RuntimeParameters_get
implicit none

call RuntimeParameters_get('plotFileNumber',io_plotFileNumber)
call RuntimeParameters_get('checkpointFileNumber',io_checkpointFileNumber)

call RuntimeParameters_get('plotFileIntervalTime',io_plotFileIntervalTime)
call RuntimeParameters_get('plotFileIntervalStep',io_plotFileIntervalStep)
call RuntimeParameters_get('checkpointFileIntervalTime',io_checkpointFileIntervalTime)
call RuntimeParameters_get('checkpointFileIntervalStep',io_checkpointFileIntervalStep)

!! etc ...

Note that the parameters found in the flash.par or in the Config files, for example plotFileNumber, are generally stored in a variable of the same name with a unit prefix prepended, for example io_plotFileNumber. In this way, a program segment clearly indicates the origin of variables. Variables with a unit prefix (e.g., io_ for IO, pt_ for particles) have been initialized from the RuntimeParameters database, and other variables are locally declared. When creating new simulations, runtime parameters used as variables should be prefixed with sim_.
Chapter 11

Multispecies Unit

Figure 11.1: The Multispecies unit directory tree.

FLASH has the ability to track multiple fluids, each of which can have its own properties. The Multispecies unit handles setting and querying on the properties of fluids, as well as some common operations on properties. The advection and normalization of species is described in the context of the Hydro unit in Chapter 14.

11.1 Defining Species

The names and properties of fluids are accessed by using their constant integer values defined in the Flash.h header file. The species names are defined in a Config file. The names of the species, for example AIR, NI56, are given in the Config file with keyword SPECIES.

In the traditional method for defining species, this Config would typically be the application’s Config file in the Simulation unit. In the alternative method described below in Section 11.4, SPECIES are normally not listed explicitly in the Simulation unit Config, but instead are automatically generated by Multispecies/MultispeciesMain/Config based on the contents of the species setup variable. Either way, the setup procedure transforms those names into accessor integers with the appended description _SPEC.

These names are stored in the Flash.h file. The total number of species defined is also defined within Flash.h as NSPECIES, and the integer range of their definition is given by SPECIES_BEGIN and SPECIES_END. To access the species in your code, use the index listed in Flash.h, for example AIR_SPEC, NI56_SPEC.

Note that NSPECIES, SPECIES_BEGIN, and SPECIES_END are always defined, whether a simulation uses multiple species or not (and whether the simulation includes the Multispecies unit or not). However, if NSPECIES = 0, SPECIES_END will be less than SPECIES_BEGIN, and then neither of them should be used as an index into solution vectors.

As an illustration, Figures Figure 11.2 and Figure 11.3 are snippets from a configuration file and the corresponding section of the FLASH header file, respectively. For more information on Config files, see
# Portion of a Config file for a Simulation

SPECIES AIR
SPECIES SF6

Figure 11.2: Sample Config file showing how to define required fluid species.

```c
#define SPECIES_BEGIN (NPROP_VARS + CONSTANT_ONE)
#define AIR_SPEC 11
#define SF6_SPEC 12
#define NSPECIES 2
#define SPECIES_END (SPECIES_BEGIN + NSPECIES - CONSTANT_ONE)
```

Figure 11.3: Sample excerpt from header file Flash.h showing integer definition of fluid species.

Section 5.5; for more information on the setup procedure, see Chapter 5; for more information on the structure of the main header file Flash.h, see Chapter 6.

**FLASH Transition**

In FLASH2, you found the integer index of a species by using `find_fluid_index`. In FLASH4, the species index is always available because it is defined in Flash.h. Use the index directory, as in `xIn(NAME_SPEC - SPECIES_BEGIN + 1) = solnData(NAME_SPEC,i,j,k)`. But be careful that the species name is really defined in your simulation! You can test with

```c
if (NAME_SPEC /= NONEXISTENT) then
  okVariable = solnData(NAME_SPEC,i,j,k)
endif
```

The available properties of an individual fluid are listed in Table 11.1 and are defined in file Multispecies.h. In order to reference the properties in code, you must `#include` the file Multispecies.h. The initialization of properties is described in the following section.

## 11.2 Initializing Species Information in Simulation_initSpecies

Before you can work with the properties of a fluid, you must initialize the data in the Multispecies unit. Normally, initialization is done in the routine `Simulation_initSpecies`. An example procedure is shown below and consists of setting relevant properties for all fluids/SPECIES defined in the Config file. Fluids do not have to be isotopes; any molecular substance which can be defined by the properties shown in Figure 11.4 is a valid input to the Multispecies unit.

**FLASH Transition**

For nuclear burning networks, a `Simulation_initSpecies` routine is already predefined. It automatically initializes all isotopes found in the Config file. To use this shortcut, `REQUIRE` the module Simulation/SimulationComposition in the Config file.
Table 11.1: Properties available through the Multispecies unit.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Number of protons and neutrons in nucleus</td>
<td>real</td>
</tr>
<tr>
<td>Z</td>
<td>Atomic number</td>
<td>real</td>
</tr>
<tr>
<td>N</td>
<td>Number of neutrons</td>
<td>real</td>
</tr>
<tr>
<td>E</td>
<td>Number of electrons</td>
<td>real</td>
</tr>
<tr>
<td>BE</td>
<td>Binding Energy</td>
<td>real</td>
</tr>
<tr>
<td>GAMMA</td>
<td>Ratio of heat capacities</td>
<td>real</td>
</tr>
<tr>
<td>MS_ZMIN</td>
<td>Minimum allowed average ionization</td>
<td>real</td>
</tr>
<tr>
<td>MS_EOSTYPE</td>
<td>EOS type to use for MTMMMT EOS</td>
<td>integer</td>
</tr>
<tr>
<td>MS_EOSSUBTYPE</td>
<td>EOS subtype to use for MTMMMT EOS</td>
<td>integer</td>
</tr>
<tr>
<td>MS_EOSFREEFILE</td>
<td>Name of file with ionization data</td>
<td>string</td>
</tr>
<tr>
<td>MS_EOSENFILE</td>
<td>Name of file with internal energy data</td>
<td>string</td>
</tr>
<tr>
<td>MS_EOSPRESFILE</td>
<td>Name of file with pressure data</td>
<td>string</td>
</tr>
<tr>
<td>MS_NUMELEMS</td>
<td>Number of elements comprising this species</td>
<td>integer</td>
</tr>
<tr>
<td>MS_ZELEMS</td>
<td>Atomic number of each species element</td>
<td>array(integer)</td>
</tr>
<tr>
<td>MS_AELEMS</td>
<td>Mass number of each species element</td>
<td>array(real)</td>
</tr>
<tr>
<td>MS_FRACTIONS</td>
<td>Number fraction of each species element</td>
<td>array(real)</td>
</tr>
<tr>
<td>MS_OPLOWTEMP</td>
<td>Temperature at which cold opacities are used</td>
<td>real</td>
</tr>
</tbody>
</table>

subroutine Simulation_initSpecies()

implicit none
#include "Multispecies.h"
#include "Flash.h"

! These two variables are defined in the Config file as
! SPECIES SF6 and SPECIES AIR
    call Multispecies_setProperty(SF6_SPEC, A, 146.)
    call Multispecies_setProperty(SF6_SPEC, Z, 70.)
    call Multispecies_setProperty(SF6_SPEC, GAMMA, 1.09)
    call Multispecies_setProperty(AIR_SPEC, A, 28.66)
    call Multispecies_setProperty(AIR_SPEC, Z, 14.)
    call Multispecies_setProperty(AIR_SPEC, GAMMA, 1.4)
end subroutine Simulation_initSpecies

Figure 11.4: A Simulation_initSpecies.F90 file showing Multispecies initialization
#include "Flash.h"
#include "Multispecies.h"

! Create arrays to store constituent element data. Note that these
! arrays are always of length MS_MAXELEMS.
real :: aelems(MS_MAXELEMS)
real :: fractions(MS_MAXELEMS)
integer :: zelems(MS_MAXELEMS)

call Multispecies_setProperty(H2O_SPEC, A, 18.0/3.0) ! Set average mass number
call Multispecies_setProperty(H2O_SPEC, Z, 10.0/3.0) ! Set average atomic number
call Multispecies_setProperty(H2O_SPEC, GAMMA, 5.0/3.0)
call Multispecies_setProperty(H2O_SPEC, MS_NUMELEMS, 2)

aelems(1) = 1.0 ! Hydrogen
aelems(2) = 16.0 ! Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_AELEMS, aelems)

zelems(1) = 1 ! Hydrogen
zelems(2) = 8 ! Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_ZELEMS, zelems)

fractions(1) = 2.0/3.0 ! Two parts Hydrogen
fractions(2) = 1.0/3.0 ! One part Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_FRACTIONS, fractions)

Figure 11.5: A Simulation_initSpecies.F90 file showing Multispecies initialization

11.3 Specifying Constituent Elements of a Species

A species can represent a specific isotope or a single element or a more complex material. Some units in FLASH require information about the elements that constitute a single species. For example, water is comprised of two elements: Hydrogen and Oxygen. The Multispecies database can store a list of the atomic numbers, mass numbers, and relative number fractions of each of the elements within a given species. This information is stored in the array properties MS_ZELEMS, MS_AELEMS, and MS_FRACTIONS respectively. The property MS_NUMELEMS contains the total number of elements for a species (MS_NUMELEMS would be two for water since water is made of Hydrogen and Oxygen). There is an upper bound on the number of elements for a single species which is defined using the preprocessor symbol MS_MAXELEMS in the “Flash.h” header file and defaults to six. The value of MS_MAXELEMS can be changed using the ms_maxelems setup variable. Figure 11.5 shows an example of how the constituent elements for water can be set using the Simulation_initSpecies subroutine.

The constituent element information is optional and is only needed if a particular unit of interest requires it. At present, only the analytic cold opacities used in the Opacity unit make use of the constituent element information.

11.4 Alternative Method for Defining Species

Section 11.1 described how species can be defined by using the SPECIES keyword in the Config file. Section 11.2 then described how the properties of the species can be set using various subroutines defined in the Multispecies unit. There is an alternative to these approaches which uses setup variables to define the species, then uses runtime parameters to set the properties of each species. This allows users to change the number and names of species without modifying the Config file and also allows users to change properties without recompiling the code.
11.5 Routine Descriptions

We now briefly discuss some interfaces to the multifluid database that are likely of interest to the user. Many of these routines include optional arguments.

- **Multispecies_setProperty** This routine sets the value species property. It should be called within the subroutine `Simulation_initSpecies` for all the species of interest in the simulation problem, and for all the required properties (any of A, Z, N, E, EB, GAMMA).

### Table 11.2: Automatically Generated Multispecies Runtime Parameters

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Runtime Parameter Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ms_&lt;spec&gt;A</td>
</tr>
<tr>
<td>Z</td>
<td>ms_&lt;spec&gt;Z</td>
</tr>
<tr>
<td>N</td>
<td>ms_&lt;spec&gt;Neutral</td>
</tr>
<tr>
<td>E</td>
<td>ms_&lt;spec&gt;Negative</td>
</tr>
<tr>
<td>BE</td>
<td>ms_&lt;spec&gt;BindEnergy</td>
</tr>
<tr>
<td>GAMMA</td>
<td>ms_&lt;spec&gt;Gamma</td>
</tr>
<tr>
<td>MS_ZMIN</td>
<td>ms_&lt;spec&gt;Zmin</td>
</tr>
<tr>
<td>MS_EOSTYPE</td>
<td>eos_&lt;spec&gt;EosType</td>
</tr>
<tr>
<td>MS_EOSSUBTYPE</td>
<td>eos_&lt;spec&gt;SubType</td>
</tr>
<tr>
<td>MS_EOSZFREEFILE</td>
<td>eos_&lt;spec&gt;TableFile</td>
</tr>
<tr>
<td>MS_EOSENERFILE</td>
<td>eos_&lt;spec&gt;TableFile</td>
</tr>
<tr>
<td>MS_EOSPRESFILE</td>
<td>eos_&lt;spec&gt;TableFile</td>
</tr>
<tr>
<td>MS_NUMELEMS</td>
<td>ms_&lt;spec&gt;NumElems</td>
</tr>
<tr>
<td>MS_ZELEMS</td>
<td>ms_&lt;spec&gt;ZElems&lt;N&gt;</td>
</tr>
<tr>
<td>MS_AELEMS</td>
<td>ms_&lt;spec&gt;AElems&lt;N&gt;</td>
</tr>
<tr>
<td>MS_FRACTIONS</td>
<td>ms_&lt;spec&gt;Fractions&lt;N&gt;</td>
</tr>
<tr>
<td>MS_OPLOWTEMP</td>
<td>op_&lt;spec&gt;LowTemp</td>
</tr>
</tbody>
</table>

Species can be defined using the `species` setup variable. For example, to create two species called AIR and SF6 one would specify `species=air,sf6` in the simulation setup command. Using this setup variable and using the `SPECIES` keyword in the `Config` file are mutually exclusive. Thus, the user must choose which method they wish to use for a given simulation. Certain units, such as the Opacity unit, requires the use of the setup variable.

When species are defined using the setup variable approach, the Multispecies unit will automatically define several runtime parameters for each species. These runtime parameters can be used set the properties shown in Table 11.1. The runtime parameter names contain the species name. Table 11.2 shows an example of the mapping between runtime parameters and Multispecies properties, where `<spec>` is replaced by the species name as specified in the species setup argument list. Some of these runtime parameters are arrays, and thus the `<N>` is a number ranging from 1 to `MS_MAXELEMS`. The `Simulation_initSpecies` subroutine can be used to override the runtime parameter settings.

### FLASH Transition

In FLASH2, you could set multiple properties at once in a call to `add_fluid_to_db`. In FLASH4, individual calls are required. If you are setting up a nuclear network, there is a precoded `Simulation_initSpecies` to easily initialize all necessary species. It is located in the unit `Simulation/SimulationComposition`, which must be listed in your simulation `Config` file.
• **Multispecies getProperty** Returns the value of a requested property.

• **Multispecies getSum** Returns a weighted sum of a chosen property of species. The total number of species can be subset. The weights are optional, but are typically the mass fractions $X_i$ of each of the fluids at a point in space. In that case, if the selected property (one of $A_i$, $Z_i$, $\ldots$, $\gamma_i$) is denoted $P_i$, the sum calculated is

$$\sum_i X_i P_i .$$

• **Multispecies getAvg** Returns the weighted average of the chosen property. As in **Multispecies getSum**, weights are optional and a subset of species can be chosen. If the weights are denoted $w_i$ and the selected property (one of $A_i$, $Z_i$, $\ldots$, $\gamma_i$) is denoted $P_i$, the average calculated is

$$\frac{1}{N} \sum_i w_i P_i ,$$

where $N$ is the number of species included in the sum; it may be less than the number of all defined species if an average over a subset is requested.

• **Multispecies getSumInv** Same as **Multispecies getSum**, but compute the weighted sum of the inverse of the chosen property. If the weights are denoted $w_i$ and the selected property (one of $A_i$, $Z_i$, $\ldots$, $\gamma_i$) is denoted $P_i$, the sum calculated is

$$\sum_i w_i \frac{1}{P_i} .$$

For example, the average atomic mass of a collection of fluids is typically defined by

$$\bar{A} = \sum_i X_i \frac{A_i}{A_i} \quad (11.1)$$

where $X_i$ is the mass fraction of species $i$, and $A_i$ is the atomic mass of that species. To compute $\bar{A}$ using the multifluid database, one would use the following lines

```plaintext
call Multispecies_getSumInv(A, abarinv, xn(:))
abar = 1.e0 / abarinv
```

where `xn(:)` is an array of the mass fractions of each species in FLASH. This method allows some of the mass fractions to be zero.

• **Multispecies getSumFrac** Same as **Multispecies getSum**, but compute the weighted sum of the chosen property divided by the total number of particles ($A_i$). If the weights give the mass fractions $X_i$ of the fluids at a point in space and the selected property (one of $A_i$, $Z_i$, $\ldots$, $\gamma_i$) is denoted $P_i$, the sum calculated is

$$\sum_i X_i \frac{1}{A_i} P_i .$$

• **Multispecies getSumSqr** Same as **Multispecies getSum**, but compute the weighted sum of the squares of the chosen property values. If the weights are denoted $w_i$ and the selected property (one of $A_i$, $Z_i$, $\ldots$, $\gamma_i$) is denoted $P_i$, the sum calculated is

$$\sum_i w_i P_i^2 .$$

• **Multispecies_list** List the contents of the multifluid database in a snappy table format.
11.6 Example Usage

In general, to use Multispecies properties in a simulation, the user must only properly initialize the species as described above in the Simulation_init routine. But to program with the Multispecies properties, you must do three things:

- `#include` the Flash.h file to identify the defined species
- `#include` the Multispecies.h file to identify the desired property
- use the Fortran interface to the Multispecies unit because the majority of the routines are overloaded.

The example below shows a snippet of code to calculate the electron density.

```fortran
#include Flash.h
#include Multispecies.h

USE Multispecies_interface, ONLY: Multispecies_getSumInv, Multispecies_getSumFrac
...
do k=blkLimitsGC(LOW,KAXIS),blkLimitsGC(HIGH,KAXIS)
   do j=blkLimitsGC(LOW,JAXIS),blkLimitsGC(HIGH,JAXIS)
      do i=blkLimitsGC(LOW,IAXIS),blkLimitsGC(HIGH,IAXIS)
         call Multispecies_getSumInv(A,abar_inv)
         abar = 1.e0 / abar_inv
         call Multispecies_getSumFrac(Z,zbar)
         zbar = abar * zbar
         ye(i,j,k) = abar_inv*zbar
      enddo
   enddo
enddo
...
```

11.7 Unit Test

The unit test for Multispecies provides a complete example of how to call the various API routines in the unit with all variations of arguments. Within Multispecies_unitTest, incorrect usage is also indicated within commented-out statements.
Chapter 12

Physical Constants Unit

The Physical Constants unit provides a set of common constants, such as Pi and the gravitational constant, in various systems of measurement units. The default system of units is CGS, so named for having a length unit in centimeters, a mass unit in grams, and a time unit in seconds. In CGS, the charge unit is the esu, and the temperature unit is the Kelvin. The constants can also be obtained in the standard MKS system of units, where length is in meters, mass in kilograms, and time in seconds. For MKS units, charge is in Coloumbs, and temperature in Kelvin.

FLASH Transition

For ease of usage, the constant \( \pi = 3.14159 \ldots \) is defined in the header file \texttt{constants.h}. Including this file with \#include “constants.h” is an alternate way to access the value of \( \pi \), rather than needing to include the PhysicalConstants unit.

Any constant can optionally be converted from the standard units into any other available units. This facility makes it easy to ensure that all parts of the code are using a consistent set of physical constant values and unit conversions.

For example, a program using this unit might obtain the value of Newton’s gravitational constant \( G \) in units of Mpc\(^3\) Gyr\(^{-2}\)M\(_\odot\)^\(^{-1}\) by calling

\[
\text{call PhysicalConstants_get ("Newton", G, len_unit="Mpc", time_unit="Gyr", mass_unit="Msun")}
\]

In this example, the local variable \( G \) is set equal to the result, \( 4.4983 \times 10^{-15} \) (to five significant figures).

Physical constants are taken from K. Nahamura et al. (Particle Data Group), J. Phys. G 37, 075021 (2010).
### 12.1 Available Constants and Units

There are many constants and units available within FLASH3, see Table 12.1 and Table 12.2. Should the user wish to add additional constants or units to a particular setup, the routine \texttt{PhysicalConstants\_init} should be overridden and the new constants added within the directory of the setup.

### 12.2 Applicable Runtime Parameters

There is only one runtime parameter used by the Physical Constants unit: \texttt{pc\_unitsBase} selects the default system of units for returned constants. It is a three-character string set to "CGS" or "MKS"; the default is CGS.

### 12.3 Routine Descriptions

The following routines are supplied by this unit.

- \texttt{PhysicalConstants\_get} Request a physical constant given by a string, and returns its real value. This routine takes optional arguments for converting units from the default. If the constant name or any of the optional unit names aren’t recognized, a value of 0 is returned.

- \texttt{PhysicalConstants\_init} Initializes the Physical Constants Unit by loading all constants. This routine is called by \texttt{Driver\_initFlash} and must be called before the first invocation of \texttt{PhysicalConstants\_get}. In general, the user does not need to invoke this call.

- \texttt{PhysicalConstants\_list} Lists the available physical constants in a snappy table.

- \texttt{PhysicalConstants\_listUnits} Lists all the units available for optional conversion.

- \texttt{PhysicalConstants\_unitTest} Lists all physical constants and units, and tests the unit conversion routines.
Table 12.2: Available Units for Conversion of Physical Constants

<table>
<thead>
<tr>
<th>Base unit</th>
<th>String Constant</th>
<th>Value in CGS units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>cm</td>
<td>1.0</td>
<td>centimeter</td>
</tr>
<tr>
<td>time</td>
<td>s</td>
<td>1.0</td>
<td>second</td>
</tr>
<tr>
<td>temperature</td>
<td>K</td>
<td>1.0</td>
<td>degree Kelvin</td>
</tr>
<tr>
<td>mass</td>
<td>g</td>
<td>1.0</td>
<td>gram</td>
</tr>
<tr>
<td>charge</td>
<td>esu</td>
<td>1.0</td>
<td>ESU charge</td>
</tr>
<tr>
<td>length</td>
<td>m</td>
<td>1.0E2</td>
<td>meter</td>
</tr>
<tr>
<td>length</td>
<td>km</td>
<td>1.0E5</td>
<td>kilometer</td>
</tr>
<tr>
<td>length</td>
<td>pc</td>
<td>3.0856775807E18</td>
<td>parsec</td>
</tr>
<tr>
<td>length</td>
<td>kpc</td>
<td>3.0856775807E21</td>
<td>kiloparsec</td>
</tr>
<tr>
<td>length</td>
<td>Mpc</td>
<td>3.0856775807E24</td>
<td>megaparsec</td>
</tr>
<tr>
<td>length</td>
<td>Gpc</td>
<td>3.0856775807E27</td>
<td>gigaparsec</td>
</tr>
<tr>
<td>length</td>
<td>Rsun</td>
<td>6.96E10</td>
<td>solar radius</td>
</tr>
<tr>
<td>length</td>
<td>AU</td>
<td>1.49597870662E13</td>
<td>astronomical unit</td>
</tr>
<tr>
<td>time</td>
<td>yr</td>
<td>3.15569252E7</td>
<td>year</td>
</tr>
<tr>
<td>time</td>
<td>Myr</td>
<td>3.15569252E13</td>
<td>megayear</td>
</tr>
<tr>
<td>time</td>
<td>Gyr</td>
<td>3.15569252E16</td>
<td>gigayear</td>
</tr>
<tr>
<td>mass</td>
<td>kg</td>
<td>1.0E3</td>
<td>kilogram</td>
</tr>
<tr>
<td>mass</td>
<td>Msun</td>
<td>1.9889225E33</td>
<td>solar mass</td>
</tr>
<tr>
<td>mass</td>
<td>amu</td>
<td>1.660538782E-24</td>
<td>atomic mass unit</td>
</tr>
<tr>
<td>charge</td>
<td>C</td>
<td>2.99792458E9</td>
<td>Coulomb</td>
</tr>
</tbody>
</table>

Cosmology-friendly units using \( H_0 = 100 \text{ km/s/Mpc} \):

| length    | LFLY            | 3.0856775807E24    | 1 Mpc       |
| time      | TFLY            | 2.05759E17         | \( \frac{2}{H_0} \) |
| mass      | MFLY            | 9.8847E45          | 5.23e12 Msun |

FLASH Transition

The header file `PhysicalConstants.h` must be included in the calling routine due to the optional arguments of `PhysicalConstants_get`.

12.4 Unit Test

The `PhysicalConstants` unit test `PhysicalConstants_unitTest` is a simple exercise of the functionality in the unit. It does not require time stepping or the grid. “Correct” usage is indicated, as is erroneous usage.
Part V

Physics Units
Chapter 13

3T Capabilities for Simulation of HEDP Experiments

The FLASH code has been extended with numerous capabilities to allow it to simulate laser-driven High Energy Density Physics (HEDP) experiments. These experiments often require a multi-temperature treatment of the plasma where the ion temperature, \( T_{\text{ion}} \) and the electron temperature \( T_{\text{ele}} \) are not necessarily equal. Thermal radiation effects are also important in many High Energy Density (HED) plasmas. If the radiation field has a total energy density given by \( u_{\text{rad}}(x,t) \) then the radiation temperature is defined as \( T_{\text{rad}} = (u_{\text{rad}}/a)^{1/4} \). The radiation field is not in equilibrium with the plasma and thus \( T_{\text{rad}} \neq T_{\text{ele}} \neq T_{\text{ion}} \). We refer to this treatment, where these three temperatures are not necessarily equal, as a 3T treatment. This chapter is intended to describe the basic theory behind FLASH’s 3T implementation to direct users to other parts of the manual and simulations which provide further details on how to use these new capabilities in FLASH.

The term “3T” is not meant to imply in any way that a gray treatment of the radiation field is being assumed. The radiation temperature is only used to represent the total energy density, which is integrated over all photon frequencies. The radiation temperature never directly enters the calculation. Thus, 3T refers to the fact that FLASH is being run in a mode where 3 independent components (ions, electrons, radiation) are being modeled. The radiation field is usually treated in a frequency dependent way through multigroup radiation diffusion as described below and in Chapter 22.

The equations which FLASH solves to describe the evolution of an unmagnetized 3T plasma are:

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0, \tag{13.1a}
\]

\[
\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho vv) + \nabla P_{\text{tot}} = 0, \tag{13.1b}
\]

\[
\frac{\partial (\rho E_{\text{tot}})}{\partial t} + \nabla \cdot [(\rho E_{\text{tot}} + P_{\text{tot}}) v] = Q_{\text{las}} - \nabla \cdot q, \tag{13.1c}
\]

where:

- \( \rho \) is the total mass density
- \( v \) is the average fluid velocity
- \( P_{\text{tot}} \) is the total pressure defined as the sum over the ion, electron, and radiation pressures:
  \[
  P_{\text{tot}} = P_{\text{ion}} + P_{\text{ele}} + P_{\text{rad}} \tag{13.2}
  \]
- \( E_{\text{tot}} \) is the total specific energy which includes the specific internal energies of the electrons, ions, and radiation field along with the specific kinetic energy. Thus:
  \[
  E_{\text{tot}} = e_{\text{ion}} + e_{\text{ele}} + e_{\text{rad}} + \frac{1}{2} v \cdot v \tag{13.3}
  \]
\(q\) is the total heat flux which is assumed to have a radiation and electron conductivity component:
\[
q = q_{\text{ele}} + q_{\text{rad}} \tag{13.4}
\]

\(Q_{\text{las}}\) represents the energy source due to laser heating.

Since the plasma is not assumed to have a single temperature, additional equations must be evolved to describe the change in specific internal energies of the ions, electrons, and radiation field. For the electrons and ions these equations are:

\[
\frac{\partial}{\partial t}(\rho_e \text{ion}) + \nabla \cdot (\rho_e \text{ion} v) + P_{\text{ion}} \nabla \cdot v = \rho c_{v, \text{ele}} (T_{\text{ion}} - T_{\text{ele}}), \tag{13.5a}
\]

\[
\frac{\partial}{\partial t}(\rho_e \text{ele}) + \nabla \cdot (\rho_e \text{ele} v) + P_{\text{ele}} \nabla \cdot v = \rho c_{v, \text{ele}} (T_{\text{ion}} - T_{\text{ele}}) - \nabla \cdot q_{\text{ele}} + Q_{\text{abs}} - Q_{\text{emis}} + Q_{\text{las}}, \tag{13.5b}
\]

\[
\frac{\partial}{\partial t}(\rho_e \text{rad}) + \nabla \cdot (\rho_e \text{rad} v) + P_{\text{rad}} \nabla \cdot v = \nabla \cdot q_{\text{rad}} - Q_{\text{abs}} + Q_{\text{emis}}, \tag{13.5c}
\]

where:

- \(c_{v, \text{ele}}\) is the electron specific heat
- \(\tau_{ei}\) is the ion/electron equilibration time
- \(Q_{\text{abs}}\) represents the increase in electron internal energy due to the total absorption of radiation
- \(Q_{\text{emis}}\) represents the decrease in electron internal energy due to the total emission of radiation

The 3T equation of state in FLASH connects the internal energies, temperatures, and pressures of the components. Many different equations of state options exist in FLASH. These are described in Section 15.4.

Chapter 14 describes the hydrodynamics solvers in a general way where only equations for conservation of total mass, momentum, and energy are considered. The extension of the FLASH hydrodynamics solvers to 3T is described in Section 14.1.4. Note that these equations are not in a conservative form because of the presence of the work terms in (13.6d), (13.6e), and (13.6f). These work terms are divergent at shocks and cannot be directly evaluated. Two techniques are described in Section 14.1.4 for coping with this issue.
The electron, ion, and radiation internal energy equations in the absence of the hydrodynamic terms are shown in (13.7). The density is updated in the hydrodynamic update so for the remaining equations the density is assumed to be constant, and we remove the density from the time derivatives.

\[
\begin{align*}
\rho \frac{\partial e_{\text{ion}}}{\partial t} &= \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}), \\
\rho \frac{\partial e_{\text{ele}}}{\partial t} &= \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}) - \nabla \cdot q_{\text{ele}} + Q_{\text{abs}} - Q_{\text{emis}} + Q_{\text{las}}, \\
\rho \frac{\partial e_{\text{rad}}}{\partial t} &= \nabla \cdot q_{\text{rad}} - Q_{\text{abs}} + Q_{\text{emis}}.
\end{align*}
\]

(13.7)

The first term on the right hand side of (13.7a) and (13.7b) describes the exchange of internal energy between ions and electrons through collisions. This term will force the ion and electron temperatures to equilibrate over time. The `Heatexchange` unit, described in Section 16.5.1, solves for this part of (13.7). Specifically, it updates the ion and electron temperatures according to:

\[
\begin{align*}
\frac{\partial e_{\text{ion}}}{\partial t} &= \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}), \\
\frac{\partial e_{\text{ele}}}{\partial t} &= \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}).
\end{align*}
\]

(13.8)

The electron specific heat, \(c_{v,\text{ele}}\) is computed through calls to the equation of state. The ion/electron equilibration time is computed in the `Heatexchange` unit.

The second term on the right hand side of (13.7b) represents the transport of energy through electron thermal conduction. Thus the heat flux is defined as:

\[
q_{\text{ele}} = -K_{\text{ele}} \nabla T_{\text{ele}},
\]

(13.9)

where \(K_{\text{ele}}\) is the electron thermal conductivity and is computed in the `Conductivity` unit (see Section 21.1). The `Diffuse` unit, described in Chapter 17, is responsible for including the effect of conduction in FLASH simulations. Again, using operator splitting, the `Diffuse` unit solves the following equation over a time step:

\[
\rho \frac{\partial e_{\text{ele}}}{\partial t} = \nabla \cdot K_{\text{ele}} \nabla T_{\text{ele}}.
\]

(13.10)

This equation can be solved implicitly over the time step to avoid time-step constraints. The electron conductivity is evaluated using a flux limiter to give more a physically realistic heat flux in regions where the electron temperature gradient is very large.

The remaining terms describe radiation transport. FLASH incorporates radiation effects using multigroup diffusion (MGD) theory. The total radiation flux, emission, and absorption terms which appear in (13.7) contain contributions from each energy group. For group \(g\), where \(1 \leq g \leq N_{g}\), the total quantities can be written as summations over each group:

\[
Q_{\text{abs}} = \sum_{g=1}^{N_{g}} Q_{\text{ele},g}, \quad Q_{\text{emis}} = \sum_{g=1}^{N_{g}} Q_{\text{emis},g}, \quad q_{\text{rad}} = \sum_{g=1}^{N_{g}} q_{g}.
\]

(13.11)

The change in the radiation energy density for each group, \(u_{g}\), is described by:

\[
\frac{\partial u_{g}}{\partial t} + \nabla \cdot (u_{g} \mathbf{v}) + \left(\frac{u_{g}}{\epsilon_{\text{rad}} \rho}\right) P_{\text{rad}} \nabla \cdot \mathbf{v} = -\nabla \cdot q_{g} + Q_{\text{emis},g} - Q_{\text{abs},g}.
\]

(13.12)

The total specific radiation energy is related to \(u_{g}\) through:

\[
\rho \epsilon_{\text{rad}} = \sum_{g=1}^{N_{g}} u_{g}.
\]

(13.13)
The RadTrans unit is responsible for solving the radiation diffusion equations for each energy group. The RadTrans unit solves these diffusion equations implicitly by using the Diffuse unit. While the work term for the total radiation energy is computed in the Hydro unit, the distribution of that work amongst each energy group is performed in the RadTrans unit. Chapter 22 describes in detail how the multigroup radiation diffusion package in FLASH functions. The group radiation flux, emission, and absorption terms are all defined in that chapter. These terms are functions of the material opacity which is computed by the Opacity unit and is described in Section 21.4.

The only remaining term in (13.7) is $Q_{\text{las}}$ which represents the deposition of energy by lasers into the electrons. The Laser implementation in the EnergyDeposition unit is responsible for computing $Q_{\text{las}}$. The geometrics optics approximation to laser energy deposition is used in FLASH. Section 16.4 describes the theory and usage of the laser ray-tracing model in FLASH in detail.

As has been described above, the HEDP capabilities in FLASH are divided amongst many units including:

- **Hydro**: Responsible for the 3T hydrodynamic update
- **Eos**: Computes 3T equation of state
- **Heatexchange**: Implements ion/electron equilibration
- **Diffuse**: Responsible for implementing implicit diffusion solvers and computes effect of electron conduction
- **RadTrans**: Implements multigroup radiation diffusion
- **Opacity**: Computes opacities for radiation diffusion
- **Conductivity**: Computes electron thermal conductivities
- **EnergyDeposition**: Computes the laser energy deposition

Several simulations are included with FLASH which demonstrate the usage of the HEDP capabilities and, taken together, exercise all of the units listed above. These simulations are described briefly below. Chapter 25 describes all of the simulations in detail. Below, the relevant simulations listed with brief descriptions.

- **MGDInfinite** simulation, described in Section 25.6.1: Simple 0D test of the exchange of energy between electrons, ions, and the radiation field
- **MGDStep** simulation, described in Section 25.6.2: Simple 1D test of electron conduction, ion/electron equilibration, and MGD
- **ShafranovShock** simulation, described in Section 25.8.1: Simple 1D verification test of the structure of a shock in a radiationless plasma with $T_{\text{ele}} \neq T_{\text{ion}}$.
- **GrayDiffRadShock** simulation, described in Section 25.8.2: Simple 1D verification test of the structure of a radiating shock
- **ReinickeMeyer** simulation, described in Section 25.8.3: Verification test of a spherical blast wave with thermal conduction
- **LaserSlab** simulation, described in Section 25.7.5: Full physics 2D simulation which includes 3T hydrodynamics, tabulated EOS and opacity, MGD, electron conduction, and laser ray-tracing. This simulation is meant to demonstrate how to set up a complex simulation of an HEDP experiment
Chapter 14

Hydrodynamics Units

Figure 14.1: The Hydro unit directory tree.

The Hydro unit solves Euler’s equations for compressible gas dynamics in one, two, or three spatial dimensions. We first describe the basic functionality; see implementation sections below for various extensions.
The Euler equations can be written in conservative form as

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \quad (14.1) \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P &= \rho \mathbf{g} \quad (14.2) \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] &= \rho \mathbf{v} \cdot \mathbf{g} \quad , \quad (14.3)
\end{align*}
\]

where \( \rho \) is the fluid density, \( \mathbf{v} \) is the fluid velocity, \( P \) is the pressure, \( E \) is the sum of the internal energy \( \epsilon \) and kinetic energy per unit mass,

\[
E = \epsilon + \frac{1}{2} |\mathbf{v}|^2 ,
\]

\( \mathbf{g} \) is the acceleration due to gravity, and \( t \) is the time coordinate. The pressure is obtained from the energy and density using the equation of state. For the case of an ideal gas equation of state, the pressure is given by

\[
P = (\gamma - 1) \rho \epsilon ,
\]

where \( \gamma \) is the ratio of specific heats. More general equations of state are discussed in Section 15.2 and Section 15.3.

In regions where the kinetic energy greatly dominates the total energy, computing the internal energy using

\[
\epsilon = E - \frac{1}{2} |\mathbf{v}|^2
\]

can lead to unphysical values, primarily due to truncation error. This results in inaccurate pressures and temperatures. To avoid this problem, we can separately evolve the internal energy according to

\[
\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + P) \mathbf{v}] - \mathbf{v} \cdot \nabla P = 0 .
\]

If the internal energy is a small fraction of the kinetic energy (determined via the runtime parameter \texttt{eintSwitch}), then the total energy is recomputed using the internal energy from (14.7) and the velocities from the momentum equation. Numerical experiments using the PPM solver included with FLASH showed that using (14.7) when the internal energy falls below \( 10^{-4} \) of the kinetic energy helps avoid the truncation errors while not affecting the dynamics of the simulation.

For reactive flows, a separate advection equation must be solved for each chemical or nuclear species

\[
\frac{\partial \rho X_{\ell}}{\partial t} + \nabla \cdot (\rho X_{\ell} \mathbf{v}) = 0 ,
\]

where \( X_{\ell} \) is the mass fraction of the \( \ell \)th species, with the constraint that \( \sum_{\ell} X_{\ell} = 1 \). FLASH will enforce this constraint if you set the runtime parameter \texttt{irenorm} equal to 1. Otherwise, FLASH will only restrict the abundances to fall between \texttt{smallx} and 1. The quantity \( \rho X_{\ell} \) represents the partial density of the \( \ell \)th fluid. The code does not explicitly track interfaces between the fluids, so a small amount of numerical mixing can be expected during the course of a calculation.

The \texttt{hydro} unit has a capability to advect mass scalars. Mass scalars are field variables advected with density, similar to species mass fractions,

\[
\frac{\partial \rho \phi_{\ell}}{\partial t} + \nabla \cdot (\rho \phi_{\ell} \mathbf{v}) = 0 ,
\]

where \( \phi_{\ell} \) is the \( \ell \)th mass scalar. Note that mass scalars are optional variables; to include them specify the name of each mass scalar in a \texttt{Config} file using the \texttt{MASS_SCALAR} keyword. Mass scalars are not renormalized in order to sum to 1, except when they are declared to be part of a renormalization group. See Section 5.5.1 for more details.
14.1 Gas hydrodynamics

14.1.1 Usage

The two gas hydrodynamic solvers supplied in the release of FLASH4 are organized into two different operator splitting methods: directionally split and unsplit. The directionally split piecewise-parabolic method (PPM) makes use of second-order Strang time splitting, and the new directionally unsplit solver is based on Monotone Upstream-centered Scheme for Conservation Laws (MUSCL) Hancock type second-order scheme.

The algorithms are described in Section 14.1.2 and Section 14.1.3 and implemented in the directory tree under physics/Hydro/HydroMain/split/PPM and physics/Hydro/HydroMain/unsplit/Hydro_Unsplit. Extensions for multitemperature applications are described in Section 14.1.4.

Current and future implementations of Hydro use the runtime parameters and solution variables described in Table 14.1 and Table 14.2. Additional runtime parameters used either solely by the PPM method or the unsplit hydro solver are described in HydroMain.

14.1.2 The piecewise-parabolic method (PPM)

FLASH includes a directionally split piecewise-parabolic method (PPM) solver descended from the PROMETHEUS code (Fryxell, Müller, and Arnett 1989). The basic PPM algorithm is described in detail in Woodward and Colella (1984) and Colella and Woodward (1984). It is a higher-order version of the method developed by Godunov (1959). FLASH implements the Direct Eulerian version of PPM.

Godunov’s method uses a finite-volume spatial discretization of the Euler equations together with an explicit forward time difference. Time-advanced fluxes at cell boundaries are computed using the numerical solution to Riemann’s shock tube problem at each boundary. Initial conditions for each Riemann problem are determined by assuming the non-advanced solution to be piecewise-constant in each cell. Using the Riemann solution has the effect of introducing explicit nonlinearity into the difference equations and permits the calculation of sharp shock fronts and contact discontinuities without introducing significant nonphysical oscillations into the flow. Since the value of each variable in each cell is assumed to be constant, Godunov’s method is limited to first-order accuracy in both space and time.
PPM improves on Godunov’s method by representing the flow variables with piecewise-parabolic functions. It also uses a monotonicity constraint rather than artificial viscosity to control oscillations near discontinuities, a feature shared with the MUSCL scheme of van Leer (1979). Although these choices could lead to a method which is accurate to third order, PPM is formally accurate only to second order in both space and time, as a fully third-order scheme proved not to be cost-effective. Nevertheless, PPM is considerably more accurate and efficient than most formally second-order algorithms.

PPM is particularly well-suited to flows involving discontinuities, such as shocks and contact discontinuities. The method also performs extremely well for smooth flows, although other schemes which do not perform the extra work necessary for the treatment of discontinuities might be more efficient in these cases. The high resolution and accuracy of PPM are obtained by the explicit nonlinearity of the scheme and through the use of intelligent dissipation algorithms, such as monotonicity enforcement and interpolant flattening. These algorithms are described in detail by Colella and Woodward (1984).

A complete description of PPM is beyond the scope of this guide. However, for comparison with other codes, we note that the implementation of PPM in FLASH uses the Direct Eulerian formulation of PPM and the technique for allowing non-ideal equations of state described by Colella and Glaz (1985). For multidimensional problems, FLASH uses second-order operator splitting (Strang 1968). We note below the extensions to PPM that we have implemented.

The PPM algorithm includes a steepening mechanism to keep contact discontinuities from spreading over too many cells. Its use requires some care, since under certain circumstances, it can produce incorrect results. For example, it is possible for the code to interpret a very steep (but smooth) density gradient as a contact discontinuity. When this happens, the gradient is usually turned into a series of contact discontinuities, producing a stair step appearance in one-dimensional flows or a series of parallel contact discontinuities in multi-dimensional flows. Under-resolving the flow in the vicinity of a steep gradient is a common cause of this problem. The directional splitting used in our implementation of PPM can also aggravate the situation. The contact steepening can be disabled at runtime by setting `use_steepening = .false.`.

The version of PPM in the FLASH code has an option to more closely couple the hydrodynamic solver with a gravitational source term. This can noticeably reduce spurious velocities caused by the operator splitting of the gravitational acceleration from the hydrodynamics. In our ‘modified states’ version of PPM, when calculating the left and right states for input to the Riemann solver, we locally subtract off from the pressure field the pressure that is locally supporting the atmosphere against gravity; this pressure is unavailable for generating waves. This can be enabled by setting `ppm_modifystates = .true.`.

The interpolation/monotonization procedure used in PPM is very nonlinear and can act differently on the different mass fractions carried by the code. This can lead to updated abundances that violate the constraint that the mass fractions sum to unity. Plewa and Müller (1999) (henceforth CMA) describe extensions to PPM that help prevent overshoots in the mass fractions as a result of the PPM advection. We implement two of the modifications they describe, the renormalization of the average mass fraction state as returned from the Riemann solvers (CMA eq. 13), and the (optional) additional flattening of the mass fractions to reduce overshoots (CMA eq. 14-16). The latter procedure is off by default and can be enabled by setting `use_cma_flattening = .true.`.

Finally, there is an odd-even instability that can occur with shocks that are aligned with the grid. This was first pointed out by Quirk (1997), who tested several different Riemann solvers on a problem designed to demonstrate this instability. The solution he proposed is to use a hybrid Riemann solver, using the regular solver in most regions but switching to an HLLE solver inside shocks. In the context of PPM, such a hybrid implementation was first used for simulations of Type II supernovae. We have implemented such a procedure, which can be enabled by setting `hybrid_riemann = .true.`.

14.1.3 The unsplit hydro solver

A directionally unsplit pure hydrodynamic solver (unsplit hydro) is an alternate gas dynamics solver to the split PPM scheme. The method basically adopts a predictor-corrector type formulation (zone-edge data-extrapolated method) that provides second-order solution accuracy for smooth flows and first-order accuracy for shock flows in both space and time. Recently, the order of spatial accuracy in data reconstruction for the normal direction has been extended to implement the 3rd order PPM and 5th order Weighted ENO (WENO) methods. This unsplit hydro solver can be considered as a reduced version of the Unsplit Staggered Mesh
The unsplit hydro implementation can solve 1D, 2D and 3D problems with added capabilities of exploring various numerical implementations: different types of Riemann solvers; slope limiters; first, second, third and fifth reconstruction methods; a strong shock/rarefaction detection algorithm as well as two different entropy fix routines for Roe’s linearized Riemann solver.

One of the notable features of the unsplit hydro scheme is that it particularly improves the preservation of flow symmetries as compared to the splitting formulation. Also, the scheme used in this unsplit algorithm can take a wide range of CFL stability limits (e.g., $\text{CFL} < 1$) for all three dimensions, which is based on using upwinded transverse flux formulations developed in the multidimensional USM MHD solver (Lee, 2006; Lee and Deane, 2009; Lee, 2013).

The above set of runtime parameters provide various types of different combinations that help in obtaining numerical accuracy, efficiency and stability. However, there are some important tips users should know before using them.

- [5th order WENO]: The 5th order method WENO requires $\text{NGUARD}=6$ in FLASH’s implementation. To use WENO, users should include $\text{+supportWENO}$ in the setup for both unsplit hydro and unsplit MHD solvers. Users should expect that this larger number of guardcells will require increased number of inter-processor data exchanges, resulting in slower performance during parallel computations.

- [Extended stencil]: When $\text{NGUARD}=6$ is used, users should also use $\text{nxb}$, $\text{nyb}$, and $\text{nzb}$ larger than $2 \times \text{NGUARD}$. For example, specifying $\text{-nxb}=16$ in the setup works well for 1D cases. Once setting up $\text{NGUARD}=6$, users still can use FOG, MH, or PPM without changing $\text{NGUARD}$ back to 4.

- [transOrder]: The first order method $\text{transOrder}=1$ is a default and only supported method that is stable according to the linear Fourier stability analysis. The choices for higher-order interpolations are no longer available in this release.

- [EOSforRiemann]: $\text{EOSforRiemann} = .true.$ will call (expensive) EOS routines to compute consistent adiabatic indices (e.g., $\text{gamc}$, $\text{game}$) according to the given left and right states in Riemann solvers. For the ideal gamma law, in which those adiabatic indices are constant, it is not required to call EOS at all and users can set it $\text{.false.}$ to reduce computation time. On the other hand, for a degenerate gas, one can enable this switch to compute thermodynamically consistent $\text{gamc}$, $\text{game}$, which in turn are used to compute the sound speed and internal energy in Riemann flux calculations. When disabled, interpolations will be used instead to get approximations of $\text{gamc}$, $\text{game}$. This interpolation method has been tested and proven to gain significant computational efficiency and accuracy, giving reliable numerical solutions even for simulating a degenerate gas. It should be noted that the new optimized unsplit hydro/MHD codes (i.e., $\text{+uhd}$, $\text{+usm}$ in $\text{source/physics/Hydro/HydroMain/unsplit}$) do not support this feature any more; however it is still supported in the old unsplit codes (i.e., $\text{+uhdold}$, $\text{+usmold}$ in $\text{source/physics/Hydro/HydroMain/unsplit_old}$).

- [Gravity coupling with Unsplit Hydro Solvers]: When gravity is included in a simulation using the unsplit hydro and MHD solvers, users can choose to include gravitational source terms in the Riemann state update at $n + 1/2$ time step (i.e., $\text{use_gravHalfUpdate}=.true.$). This will provide a second-order accuracy with respect to coupling gravitational accelerations to hydrodynamics. With $\text{use_gravHalfUpdate}=.true.$, users can choose $\text{use_gravConsv}=.true.$ to adopt conservative forms (expensive); although, an efficient primitive counterpart update should be accurate enough for most cases. Otherwise, if $\text{use_gravHalfUpdate}=.false.$, the gravitational source terms will only be included at the final update step (i.e., $U^n$ to $U^{n+1}$), giving first order accuracy. We also have included a new approach to update gravity accelerations at $n + 1/2$ time step (i.e., $\text{use_gravPotUpdate}=.true.$). In this new approach, the gravity potentials are calculated by calling a Poisson solver, giving more accurate values at cell-interfaces. The option $\text{use_gravPotUpdate}=.true.$ only works with the primitive update, $\text{use_gravConsv}=.false.$. It should be noted that the new optimized unsplit hydro/MHD codes (i.e., $\text{+uhd}$, $\text{+usm}$ in $\text{source/physics/Hydro/HydroMain/unsplit}$) do not support $\text{use_gravPotUpdate}=.true.$ and $\text{use_gravConsv}=.true.$ any more; however they are still supported in the old unsplit codes (i.e., $\text{+uhdold}$, $\text{+usmold}$ in $\text{source/physics/Hydro/HydroMain/unsplit_old}$).
Table 14.3: Additional runtime parameters for \textit{Interpolation Schemes} in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro Unsplit)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>integer</td>
<td>2</td>
<td>Order of method in data reconstruction: 1st order Godunov (FOG), 2nd order MUSCL-Hancock (MH), 3rd order PPM, 5th order WENO. NOTE: WENO requires \texttt{NGUARD}=6, and should include \texttt{+supportWENO} in setup line.</td>
</tr>
<tr>
<td>transOrder</td>
<td>integer</td>
<td>1</td>
<td>Interpolation order of accuracy of taking upwind biased transverse flux derivatives in the unsplit data reconstruction: 1st, 2nd, 3rd. The choice of using \texttt{transOrder}=4 adopts a slope limiter between the 1st and 3rd order accurate methods to minimize oscillations in upwinding at discontinuities.</td>
</tr>
<tr>
<td>slopeLimiter</td>
<td>string</td>
<td>“vanLeer”</td>
<td>Slope limiter: “MINMOD”, “MC”, “VANLEER”, “HYBRID”, “LIMITED”.</td>
</tr>
<tr>
<td>LimitedSlopeBeta</td>
<td>real</td>
<td>1.0</td>
<td>Slope parameter specific for the “LIMITED” slope by Toro.</td>
</tr>
<tr>
<td>charLimiting</td>
<td>logical</td>
<td>.true.</td>
<td>Enable/disable limiting on characteristic variables (.false. will use limiting on primitive variables)</td>
</tr>
<tr>
<td>use_steepening</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable contact discontinuity steepening for PPM and WENO.</td>
</tr>
<tr>
<td>use_flattening</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable flattening (or reducing) numerical oscillations for MH, PPM, and WENO.</td>
</tr>
<tr>
<td>use_avisc</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable artificial viscosity for FOG, MH, PPM, and WENO.</td>
</tr>
<tr>
<td>cvisc</td>
<td>real</td>
<td>0.1</td>
<td>Artificial viscosity coefficient.</td>
</tr>
<tr>
<td>use_upwindTVD</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable upwinded TVD slope limiter PPM. NOTE: This requires \texttt{NGUARD}=6</td>
</tr>
<tr>
<td>use_hybridOrder</td>
<td>logical</td>
<td>.false.</td>
<td>Enable an adaptively varying reconstruction order scheme reducing its order from a high-order to first-order depending on monotonicity constraints</td>
</tr>
<tr>
<td>use_gravHalfUpdate</td>
<td>logical</td>
<td>.false.</td>
<td>On/off gravitational acceleration source terms at the half time Riemann state update</td>
</tr>
<tr>
<td>use_gravConsv</td>
<td>logical</td>
<td>.false.</td>
<td>Primitive/conservative update for including gravitational acceleration source terms at the half time Riemann state update</td>
</tr>
<tr>
<td>use_gravPotUpdate</td>
<td>logical</td>
<td>.false.</td>
<td>Use gravity source term update by calling Poisson solver. Note: this only can be used with \texttt{use_gravConsv}=.false.</td>
</tr>
<tr>
<td>use_3dFullCTU</td>
<td>logical</td>
<td>.true.</td>
<td>Enable a full CTU (e.g., similar to the standard 12-Riemann solve) algorithm that provides full CFL stability in 3D. If .false., then the theoretical CFL bound for 3D becomes less than 0.5 based on the linear Fourier analysis.</td>
</tr>
</tbody>
</table>
Table 14.4: Additional runtime parameters for Riemann Solvers in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro_Unsplit)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shockDetect</td>
<td>logical</td>
<td>.false.</td>
<td>On/off detecting strong shocks/rarefactions to gain numerical stability by reducing CFL</td>
</tr>
<tr>
<td>EOSforRiemann</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable calling EOS in computing each Godunov flux</td>
</tr>
<tr>
<td>entropy</td>
<td>logical</td>
<td>.false.</td>
<td>On/off entropy fix algorithm for Roe solver</td>
</tr>
</tbody>
</table>

- [Reduced CTU vs. Full CTU for 3D in the unsplit hydro (UHD) and staggered mesh (USM) solvers]:
  
  `use_3dFullCTU` is a new switch that enhances a numerical stability for 3D simulations in the unsplit solvers using the corner transport upwind (CTU) algorithm by Colella. The FLASH’s unsplit solvers are different from many other shock capturing codes in that both UHD and USM solvers do not need intermediate Riemann solver solutions for updating transverse fluxes in multidimensional problems. This provides a computational efficiency because there is a reduced number of calls to Riemann solvers per cell per time step. The total number of required Riemann solver solutions are two for 2D and three for 3D (except for extra Riemann calls for constraint-transport (CT) update in USM). This is smaller than the usual stability requirement in many other codes which needs four for 2D and twelve for 3D in order to provide a full CFL limit (e.g., CFL < 1).

  In general for 3D, there is another computationally efficient approach that only uses six Riemann solutions (aka, 6-CTU) instead of solving twelve Riemann problems (aka, 12-CTU). In this efficient 6-CTU, however, the numerical stability limit becomes CFL < 0.5.

  For solving 3D problems in UHD and USM, enabling the new switch `use_3dFullCTU=.true.` (i.e., full-CTU) will make the solution evolution scheme similar to 12-CTU while requiring to solve three Riemann problems only (again, except for the CT update in USM). On the other hand, `use_3dFullCTU=.false.` (i.e., reduced-CTU) will be similar to the 6-CTU integration algorithm with a reduced CFL limit (e.g, CFL < 0.5).

---

Two New Optimized Unsplit Hydro Solver and Unsplit Staggered MHD Mesh Solver

We provide two new optimized unsplit solvers in FLASH4.2 release. They serve as two default implementations of the unsplit solvers and are found in `source/physics/Hydro/HydroMain/unsplit` in the FLASH source tree. The shortcuts for the unsplit solvers `+uhd` and `+usm` now point to these two optimized solvers, whereas users still can use the previous version of unoptimized unsplit solvers with `+uhdold` and `+usmold`, which will choose the old unsplit solver implementations in `source/physics/Hydro/HydroMain/unsplit_old`.  

---
Unsplit Hydro Solver vs. Unsplit Staggered MHD Mesh Solver

One major difference between the unsplit hydro solver and the USM MHD solver is the presence of magnetic and electric fields. The associated staggered mesh configuration required for the USM MHD solver is not needed in the unsplit hydro solver, and all hydrodynamic variables are stored at cell centers.

Stability Limits for both Unsplit Hydro Solver and Unsplit Staggered Mesh Solver

As mentioned above, the two unsplit solvers can take a wide range of CFL limits in all three dimensions (e.g., CFL < 1). However, in some circumstances where there are strong shocks and rarefactions, shockDetect=.true. could be useful to gain more numerical stability by lowering the CFL accordingly (e.g., default settings provide 0.45 for 2D and 0.25 for 3D for the Donor scheme). This approach will automatically revert such reduced stability conditions to any given original condition set by users when there are no significant shocks and rarefactions detected.

Setting up a simulation with the unsplit hydro solver

One needs to specify +unsplitHydro (or +uhd for short) in the setup line in order to choose the unsplit hydro solver for a simulation. For instance, a setup script ./setup Sedov -2d -auto +unsplitHydro will run a Sedov 2D problem using the unsplit hydro solver. Without specifying +unsplitHydro, the default PPM solver will be selected.

Diffusion terms

Non-ideal terms, such as viscosity and heat conduction, can be included in the unsplit hydro solver for simulating diffusive processes. Please see related descriptions in Section 14.3.5.

Non-Cartesian Grid Support

Grid support for non-Cartesian geometries has been newly included in the unsplit hydro and MHD solvers in the current release. The supported geometries are (i) 1D spherical (ii) 2D cylindrical in r-z and r-θ. Please see related descriptions in Section 8.11.

14.1.3.1 Implementation of Stationary Rigid Body in a Simulation Domain for Unsplit Hydro Solver

An approach to include a single or multiple stationary rigid body (bodies) in a simulation domain has been newly introduced in the unsplit hydro solver. Using this new feature it is possible to add any numbers of solid bodies that are of any shapes inside a computational domain, where a reflecting boundary condition is to be applied at each solid surface. Due to the nature of regular box-like grid structure in FLASH, the surface of rigid body looks like stair steps at best rather than smooth or round shapes. High refinement levels are recommended at such stair shaped interfaces around the rigid body.
In order to add a rigid body in a simulation, users first need to add a variable called `BDRY_VAR` in a simulation Config file. The next step is to initialize `BDRY_VAR` in `Simulation_initBlock.F90` in such a way that a positive one is assigned to cells in a rigid body (i.e., `solnData(BDRY_VAR,i,j,k)=1.0`); otherwise a negative one for all other cells (i.e., `solnData(BDRY_VAR,i,j,k)=-1.0`).

Users can allow high resolutions around the rigid body by promoting `BDRY_VAR` to be one of the refinement variables (i.e., `refine_var_1='bdry'` in `flash.par`). The implementation automatically adapts order (a spatial reconstruction order; see Table 14.4) in fluid cells that are near the rigid body, reducing any order (>1) to order=1 at those fluid cells adjacent to the body. This prohibits any high order (higher than 1) interpolation algorithms from reaching the rigid body data which should not be used when reconstructing high order Riemann states in the adjacent fluid cells.

For this reason there is one stability issue during simulations when order in the fluid cells becomes 1 and hence the local reconstruction scheme becomes a first order Godunov method. For these cells, the multidimensional local data reconstruction-evolution integration scheme reduces to a donor cell method (otherwise globally the corner-transport-upwind method by Colella) which requires a reduced CFL limit (e.g., CFL < 1/2 for 2D; CFL < 1/3 for 3D). Therefore, when there is a rigid body in a simulation users need to provide such a reduced CFL number in `flash.par`.

Two example simulations can be found in Section 25.1.12.1 and Section 25.1.12.2.

### 14.1.4 Multitemperature extension for Hydro

Chapter 13 described the new capabilities in FLASH for modeling High Energy Density Physics (HEDP) experiments and describes the basic theory. These experiments often require a 3T treatment of the plasma as described in Chapter 13. Equation (13.1) shows the full set of 3T equations solved by the current version of FLASH. A series of operator splits are used to isolate the hydrodynamic terms from the various source terms. The Hydro unit is responsible for solving the system of equations which includes the hydrodynamic terms, shown in (13.6). These terms describe advection and work. Note this system contains a redundant equation since the total energy equation, (13.6c), can be written as a sum which includes (13.6b), (13.6d), (13.6e), and (13.6f).

A significant challenge exists in solving (13.6) since (13.6d), (13.6e), and (13.6f) contain source terms that involve velocity divergences; these are the work terms. The quantity $\nabla \cdot \mathbf{v}$ is not defined at a shock, and thus directly evaluating this source term is not possible. Two techniques have been implemented in FLASH for solving (13.6) without evaluating the work terms directly. These will be referred to as the entropy advection approach and the RAGE-like approach. These approaches exploit the fact that the existing hydrodynamic solvers (the split and unsplit solvers) already solve the conservation equations for total mass, momentum, and energy. These equations retain the same form in the 3T case, however they are not complete and must be augmented with other equations to close the system since the total pressure is computed using an equation of state that requires knowledge of the properties of ions and electrons independently. For example, in many equations of state, we can write that $P_{\text{ele}} = P_{\text{ele}}(\rho, e_{\text{ele}})$ and $P_{\text{ion}} = P_{\text{ion}}(\rho, e_{\text{ion}})$. Thus $P = P_{\text{ele}} + P_{\text{ion}} = P(\rho, e_{\text{ele}}, e_{\text{ion}})$. In the 1T case, the system can be closed by assuming $T_{\text{ele}} = T_{\text{ion}}$.

#### 14.1.4.1 The Entropy Advection Approach

The total entropy is not continuous at a shock. To conserve mass, momentum, and energy, shocks must irreversibly convert kinetic energy to internal energy. A good approximation that can be made is that the electron entropy is continuous across a shock. The entropy advection approach makes this assumption to solve for the state of the ions. The entropy advection approach has had limited testing. It has not been extended to incorporate radiation. Thus, there is no need to include (13.6f).

The entropy advection approach solves the first three equations of (13.6) for conservation of total mass, momentum, and energy. Now the system can be closed by solving either (13.6d) or (13.6e). However, this cannot be done, since those equations have terms that are divergent at shocks. The solution is to add an additional equation which states that electron entropy is advected with the fluid:

$$\frac{\partial}{\partial t}(\rho s_{\text{ele}}) + \nabla \cdot (\rho s_{\text{ele}} \mathbf{v}) = 0$$

(14.10)
The electron internal energy, temperature and other properties are then computed using the EOS in a mode which accepts specific electron entropy $s_{ele}$ in addition to specific combined internal energy $T e_{tot}$ and $\rho$ as inputs. Thus, the solution procedure for the entropy advection scheme is:

- Solve the system of equations defined by (13.6a), (13.6b), (13.6c), and (14.10)
- Compute the total specific internal energy, $e_{tot}$, by subtracting the kinetic energy from the total energy:
  $$e_{tot} = E_{tot} - \frac{\mathbf{v} \cdot \mathbf{v}}{2}$$
- Compute the electron specific internal energy using the 3T equation of state:
  $$e_{ele} = \text{EOS}(\rho, s_{ele}, e_{tot})$$
- Compute the ion specific internal energy from the total specific internal energy using the kinetic energy and the electron internal energy:
  $$e_{ion} = e_{tot} - e_{ele}$$

### 14.1.4.2 The RAGE-like Approach

The RAGE-like approach is so named because it is identical to the method implemented in the radiation hydrodynamics code RAGE (Gittings, 2008). Verification tests comparing the two codes have shown nearly identical behavior. The RAGE-like is physically accurate in smooth flow, but does not distribute internal energy correctly among the ions, electrons, and radiation field at shocks. This is in contrast to the entropy advection approach which does distribute energy correctly, but has its own limitations.

In general, let $\Delta e_s$ be the change in the internal energy at a particular location over some short time, $\Delta t$. The subscript $s$ refers to either ions, electrons, radiation, or the total specific internal energy. When considering only the hydrodynamic effects, $\Delta e_s = \Delta e_s^{adv} + \Delta e_s^{work} + \Delta e_s^{shock}$, where:

1. $\Delta e_s^{adv}$ refers to the change in internal energy due to the advection of internal energy with the fluid
2. $\Delta e_s^{work}$ refers to the change in internal energy due to hydrodynamic work. This term is not well defined near shocks.
3. $\Delta e_s^{shock}$ refers to changes in internal energy due to shock heating. Shock heating is the necessarily, irreversible conversion of kinetic energy to internal energy which occurs at shock as a consequence of conserving mass, momentum, and energy.

One physically accurate approximation is that $\Delta e_{ion}^{shock} = \Delta e_{tot}^{shock}$. The challenge of 3T hydrodynamics lies in dividing these components so that they can be correctly apportioned among the ions, electrons, and radiation field. The entropy advection approach avoids maintaining the physically accurate result by solving an equation for electron entropy.

The RAGE-like approach does not apportion shock heating to only the ions. Rather it apportions the quantity $\Delta e_{ion}^{shock} + \Delta e_{tot}^{shock}$ among the ions, electrons, and radiation field in proportion to the partial pressures of these components. This is consistent with (13.6) in smooth flow, but it not accurate near shocks where the shock heating should only contribute to the change in the ion internal energy. Note that it is possible to isolate internal energy changes due to advection by solving as set of advection equations for the ions, electrons, and radiation field.

Thus, the solution procedure for the RAGE-like approach is:

1. Solve the system of equations defined by (13.6a), (13.6b), and (13.6c) and simultaneously solve:
   $$\frac{\partial}{\partial t}(\rho e_{ele}) + \nabla \cdot (\rho e_{ele}\mathbf{v}) = 0,$$
   \hspace{1cm} (14.11a)
   $$\frac{\partial}{\partial t}(\rho e_{ion}) + \nabla \cdot (\rho e_{ion}\mathbf{v}) = 0,$$
   \hspace{1cm} (14.11b)
   $$\frac{\partial}{\partial t}(\rho e_{rad}) + \nabla \cdot (\rho e_{rad}\mathbf{v}) = 0,$$
   \hspace{1cm} (14.11c)
   to update the internal energies by including only advection related changes.
2. Compute the change in total specific internal energy over the time step for each computational cell:
   $$\Delta e_{tot} = \Delta E_{tot} - \mathbf{v} \cdot \mathbf{v}/2$$
Compute $\Delta e_{\text{work}} + \Delta e_{\text{shock}}$ by subtracting the advected energy changes from the total change in internal energy.

Divide $\Delta e_{\text{work}} + \Delta e_{\text{shock}}$ amongst the ions, electrons, and radiation field according to the ratio of pressures.

### 14.1.4.3 Use, Implications, and Limitations of Multitemperature Hydro Approaches

Each approach has its own strengths and weaknesses. The entropy advection approach is more accurate in the sense that it correctly includes all shock heating in the ion internal energy. The RAGE-like approach is less accurate near shocks for this reason in terms of predicting the correct downstream ion temperature and electron temperature. In general, immediately downstream of a shock, the RAGE-like approach will predict an electron temperature that is too large and an ion temperature that is too small. However, the density and velocity in the vicinity of shocks will be accurate. Furthermore, ion/electron equilibration (see Chapter 13) will quickly act to equalize the ion and electron temperatures. Therefore, the RAGE-like approach is reasonable when the ion/electron equilibration time is small as is the case in many physical scenarios. The Shafranov Shock simulation (see Section 25.8.1) compares the temperatures produced by the RAGE-like approach through these two approaches.

The entropy advection approach has some practical limitations that prevent its use in FLASH for general problems. First, many of the 3T EOS models in FLASH do not support the calculation of electron entropy. The only EOS that does is the gamma-law model which is not appropriate modeling many HEDP experiments. Furthermore, oscillations in the electron and ion temperatures have been observed when using this approach. These oscillations can lead to negative ion temperatures. For these reasons, the entropy advection approach has not yet been used for production FLASH simulations of HEDP experiments.

Users can select which 3T hydro approach to use by setting the `hy_eosModeAfter` runtime parameter. When set to “dens ie gather” a RAGE-like approach is used. When set to “dens ie sele gather”, the entropy advection approach is used. Due to the limitations described above, the RAGE-like approach is currently the default option. The `+uhd3t` setup shortcut can be used to setup a simulation using the 3T extension of the unsplit hydrodynamics solver. The split solver is included by default whenever a multitemperature EOS is included in the simulation. The Shafranov Shock simulation (Section 25.8.1) and radiative shock (Section 25.8.2) simulations demonstrate the use of the split solver. The Laser Slab simulation (Section 25.7.5) demonstrates the use of the 3T unsplit hydrodynamics solver.

The multitemperature hydro extensions are implemented in several implementation directories named `multiTemp` within the normal hydrodynamic solvers. These contain source files that replace some of the source files of the same name that reside higher in the source tree, and related source and Config files. Together, these files add code functionality and solution variables that are appropriate for multitemperature simulations. The FLASH configuration mechanism automatically takes care of building with the right versions of source files when the `multiTemp` directories are included.

The Hydro unit thus adds variables that are used to describe the state of each of the three components. In the current version, these are maintained in addition to corresponding variables for the state of the fluid as a whole. There is thus redundancy in the state description. As a benefit of this redundancy, some code units that were written without multitemperature in mind may continue to function in a multitemperature context by referring to the usual 1T set of variables (like `temp`, `pres`, `ener`, `eint`). The set of variables may be optimized in future code revisions. Table 14.5 shows some of the additional variables.

Hydro needs a multitemperature implementation of Eos in order to work for multitemperature setups. Thus when a simulation is configured to use the multiTemp Hydro code, it also needs to include one of the Eos implementations under physics/Eos/EosMain/multiTemp. The Config files under multiTemp in Hydro will take care of this, but the simulation may have to request a specific implementation under physics/Eos/EosMain/multiTemp. A simulation also needs to control which EOS modes to use during a simulation. Thus the runtime parameters `eosMode`, `eosModeInit`, and the new `hy_eosModeAfter` need to be set appropriately.
Table 14.5: Additional solution variables used with the hydrodynamics (Hydro) unit extended for multitemperature. Note that “specific” variables are understood as per mass unit of the combined fluid.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Corresponding 1T Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tion</td>
<td>temp</td>
<td>GENERIC</td>
<td>ion temperature</td>
</tr>
<tr>
<td>tele</td>
<td>temp</td>
<td>GENERIC</td>
<td>electron temperature</td>
</tr>
<tr>
<td>trad</td>
<td>temp</td>
<td>GENERIC</td>
<td>radiation temperature</td>
</tr>
<tr>
<td>pion</td>
<td>pres</td>
<td>GENERIC</td>
<td>ion pressure</td>
</tr>
<tr>
<td>pele</td>
<td>pres</td>
<td>GENERIC</td>
<td>electron pressure</td>
</tr>
<tr>
<td>prad</td>
<td>pres</td>
<td>GENERIC</td>
<td>radiation pressure</td>
</tr>
<tr>
<td>eion</td>
<td>eint/ener</td>
<td>PER, MASS</td>
<td>specific ion internal energy</td>
</tr>
<tr>
<td>eele</td>
<td>eint/ener</td>
<td>PER, MASS</td>
<td>specific electron internal energy</td>
</tr>
<tr>
<td>erad</td>
<td>eint/ener</td>
<td>PER, MASS</td>
<td>specific radiation energy</td>
</tr>
<tr>
<td>sele</td>
<td>PER, MASS (mass scalar)</td>
<td></td>
<td>specific electron entropy</td>
</tr>
</tbody>
</table>

14.1.5 Chombo compatible Hydro

This is a slightly modified version of the split Hydro solver (Section 14.1.2) which must be used when including Chombo Grid. It is available at physics/Hydro/HydroMain/split/PPM/chomboCompatible and is automatically selected by the setup script whenever an application includes Chombo Grid.

It is required because the Chombo class LevelFluxRegister which performs flux correction at fine-coarse boundaries has a very different interface to the subroutines in Paramesh. A LevelFluxRegister conserves fluxes at fine-coarse boundaries and then corrects the coarse solution data. In contrast, the Paramesh subroutine Grid_conserveFluxes only conserves fluxes and then FLASH updates the solution data in hy_ppm_updateSoln.

This implementation has not been tested extensively and should not be used in a production simulation. We have encountered worse than expected mass and total energy conservation when it has been used with Chombo Grid. It is not yet known whether this is a problem with Chombo compatible Hydro or whether our AMR refinement is too aggressive and there is fine-coarse interpolation where there are sharp shock fronts. Aggressive AMR refinement could be the problem because the availability of cell-by-cell refinement in Chombo is very different to our prior experience with Paramesh which refines entire blocks at a time.

An example of bad mass and total energy conservation happens in a Sod problem that uses the included parameter file named test_regular_fluxes_fbs_troublesome.par. The boundary conditions are setup to be periodic, so that mass and total energy should be conserved. Integrated mass and total energy at the end of the test problem change by order $1 \times 10^{-5}\%$. Increasing lrefine_max from 2 to 4 does not lead to worse conservation. We have found that we can eliminate the conservation problems in this test problem by setting BRMeshRefineBlockFactor $= 8$ and maxBlockSize $= 8$, however, such a change has not fixed every test problem configuration which has shown poor conservation. In general, we get better conservation when using mesh parameter values maxBlockSize $\geq 8$, BRMeshRefineBlockFactor $\geq 8$, refRatio $= 2$ and tagRadius $\geq 2$, and so have added warning messages in the FLASH log file when these values are outside of the given range. We expect to learn more about the conservation issue soon after the FLASH4 release.

This Hydro implementation can be manually included in an application with Paramesh mesh package as follows:

```
./setup Sod -auto chomboCompatibleHydro=True -parfile=test_regular_fluxes_fbs_troublesome.par
```

The logical parameter chomboLikeUpdateSoln controls whether or not FLASH uses the standard hy_ppm_updateSoln for the solution update. A value of .true. indicates that it will not be used and is the only acceptable value for Chombo based applications. A value of .false. means it is used and is a possibility for Paramesh based applications.
14.2 Relativistic hydrodynamics (RHD)

14.2.1 Overview

FLASH provides support for solving the equations of special relativistic hydrodynamics (RHD) in one, two and three spatial dimensions.

Relativistic fluids are characterized by at least one of the following two behaviors: (i) bulk velocities close to the speed of light (kinematically relativistic regime), (ii) internal energy greater than or comparable to the rest mass density (thermodynamically relativistic regime). As can be seen from the equations in Section 14.2.2, the two effects become coupled by the presence of the Lorentz factor; as a consequence, transverse velocities do not obey simple advection equations. Under these circumstances, Newtonian hydrodynamics is not adequate and a correct description of the flow must take relativistic effects into account.

14.2.2 Equations

The motion of an ideal fluid in special relativity is described by the system of conservation laws

\[ \frac{\partial}{\partial t} \left( \begin{array}{c} D \\ m \\ E \end{array} \right) + \nabla \cdot \left( \begin{array}{c} D v + p I \\ m \\ v \end{array} \right) = 0, \]  

(14.12)

where \( D, m, E, v \) and \( p \) define, respectively, the fluid density, momentum density, total energy density, three-velocity and pressure of the fluid. (14.12) is written in units of \( c = 1 \), where \( c \) is the speed of light. The same convention is also adopted in the FLASH code.

At present, only Cartesian (1, 2 and 3-D), 2-D cylindrical \((x = r, y = z)\) and 1-D spherical \((1-D, x = r)\) geometries are supported by FLASH. Gravity is not included, although it can be easily added with minor modifications.

An equation of state (Eos) provides an additional relation between thermodynamic quantities and closes the system of conservation laws (14.12). The current version of FLASH supports only the ideal equation of state, for which the specific enthalpy \( h \) may be expressed as

\[ h = 1 + \frac{\Gamma}{\Gamma - 1} \frac{p}{\rho}, \]  

(14.13)

where \( \Gamma \) (constant) is the specific heat ratio and \( \rho \) is the proper rest mass density. Causality \( (c_s < c) \) is preserved for \( \Gamma < 2 \). The specific heat ratio is specified as a runtime parameter \("\text{gamma}\)\).

As in classical hydrodynamics, relativistic fluids may be described in terms of a state vector of conservative, \( U = (D, m, E) \), or primitive, \( V = (\rho, v, p) \), variables. The connection between the two sets is given by

\[ D = \gamma \rho, \quad m = \rho \gamma^2 v, \quad E = \rho \gamma^2 - p, \]  

(14.14)

where \( \gamma = (1 - v^2)^{-1/2} \) is the Lorentz factor. Notice that the total energy density includes the rest mass contribution. The inverse relation, giving \( V \) in terms of \( U \), is

\[ \rho = \frac{D}{\gamma}, \quad v = \frac{m}{E + p}, \quad p = Dh\gamma - E. \]  

(14.15)

This inverse map is not trivial due to the non-linearity introduced by the Lorentz factor \( \gamma \); it can be shown, in fact, that (14.15) can be combined together to obtain the following implicit expression for \( p \):

\[ p = Dh(\rho, \tau(p))\gamma(p) - E. \]  

(14.16)
(14.16) has to be solved at least once per time step in order to recover the pressure from a set of conservative variables $U$. Notice that $\tau = \tau(p)$ depends on the pressure $p$ through $\tau = \gamma(p)/D$ and that the specific enthalpy $h$ is, in general, a function of both $p$ and $\tau$, $h = h(p, \tau(p))$. The conversion routines are implemented in the `rhd_conserveToPrimitive.F90` and `rhd_primitiveToConserve.F90` source files.

### 14.2.3 Relativistic Equation of State

A variant version of the ideal gamma law `Eos_wrapped.F90` routine is required by the RHD unit and is provided in `Eos/EosMain/Gamma/RHD`. In order to do so the unit requires a typical `Config` file which should look like this:

```plaintext
REQUIRES physics/Eos/EosMain/Gamma/RHD
```

For this specific purpose, the current RHD implementation supports `MODE_DENS_EI` (a default mode) and `MODE_DENS_PRES` only (but not `MODE_DENS_TEMP`) in making a `Eos_wrapped` call.

### 14.2.4 Additional Runtime Parameter

One additional runtime parameter used with the RHD unit is

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhd_reconType</td>
<td>integer</td>
<td>1</td>
<td>Order of reconstruction scheme: 1 for piecewise-liner; 2 for piecewise parabolic</td>
</tr>
</tbody>
</table>

### 14.3 Magnetohydrodynamics (MHD)

#### 14.3.1 Description

The FLASH4 code includes two magnetohydrodynamic (MHD) units that represent two different algorithms. The first is the eight-wave model (`8Wave`) by Powell et al. (1999) that is already present in FLASH2. The second is a newly implemented unsplit staggered mesh algorithm (`USM` or `StaggeredMesh`). It should be noted that there are several major differences between the two MHD units. The first difference is how each algorithm enforces the solenoidal constraint of magnetic fields. The eight-wave model basically uses the truncation-error method, which effectively removes the effects of unphysical magnetic monopoles if they are generated during simulations. It does not, however, completely eliminate monopoles that are spurious in a strict physical law. To improve such unphysical effects in simulations, the unsplit staggered mesh algorithm uses the constrained transport method (Evans and Hawley, 1988) to enforce divergence-free constraints of magnetic fields. This method is shown to maintain magnitudes of $\nabla \cdot B$ substantially low, e.g., to the orders of $10^{-12}$ or below, in most simulations. The second major difference is that the unsplit staggered mesh algorithm uses a directionally unsplit scheme to evolve the MHD governing equations, whereas the eight-wave method uses a directionally splitting method as in FLASH2. In general, the splitting method is shown to be robust, relatively straightforward to implement, and generally faster than the unsplit method. The splitting method, however, does generally introduce splitting errors when solving one-dimensional subproblems in each sweep direction for multidimensional MHD equations. This error gets introduced in simulations because (i) the linearized Jacobian flux matrices do not commute in most of the nonlinear multidimensional problems (LeVeque, 1992; LeVeque, 1998), and (ii) in MHD, dimensional-splitting based codes are not able to evolve the normal (to the sweep direction) magnetic field during each sweep direction (Gardiner and Stone, 2005).

Note that the eight-wave solver uses the same directionally splitting driver unit `Driver/DriverMain/split` as the PPM and RHD units do, while the unsplit staggered mesh solver (`StaggeredMesh`) has its own independent unsplit driver unit `Driver/DriverMain/unsplit`. 
Both MHD units solve the equations of compressible ideal and non-ideal magnetohydrodynamics in one, two and three dimensions on a Cartesian system. Written in non-dimensional (hence without $4\pi$ or $\mu_0$ coefficients) conservation form, these equations are

\begin{align}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v} - \mathbf{BB}) + \nabla p_* &= \rho \mathbf{g} + \nabla \cdot \mathbf{\tau} \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho \mathbf{v} E + p_*) - \mathbf{B} (\mathbf{v} \cdot \mathbf{B}) &= \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot (\mathbf{v} \cdot \mathbf{\tau} + \sigma \nabla T) + \nabla \cdot (\mathbf{B} \times (\eta \nabla \times \mathbf{B})) \\
\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v} \mathbf{B} - \mathbf{B} \mathbf{v}) &= -\nabla \times (\eta \nabla \times \mathbf{B})
\end{align}

where

\begin{align}
p_* &= p + \frac{B^2}{2}, \\
E &= \frac{1}{2} v^2 + \epsilon + \frac{1}{2} \frac{B^2}{\rho}, \\
\mathbf{\tau} &= \mu \left( (\nabla \mathbf{v}) + (\nabla \mathbf{v})^T - \frac{2}{3} (\nabla \cdot \mathbf{v}) \mathbf{I} \right)
\end{align}

are total pressure, specific total energy and viscous stress respectively. Also, $\rho$ is the density of a magnetized fluid, $\mathbf{v}$ is the fluid velocity, $p$ is the fluid thermal pressure, $T$ is the temperature, $\epsilon$ is the specific internal energy, $\mathbf{B}$ is the magnetic field, $\mathbf{g}$ is the body force per unit mass, for example, due to gravity. $\mathbf{\tau}$ is the viscosity tensor, $\mu$ is the coefficient of viscosity (dynamic viscosity), $\mathbf{I}$ is the unit (identity) tensor, $\sigma$ is the heat conductivity, and $\eta$ is the resistivity. The thermal pressure is a scalar quantity, so that the code is suitable for simulations of ideal plasmas in which magnetic fields are not so strong that they cause temperature anisotropies. As in regular hydrodynamics, the pressure is obtained from the internal energy and density using the equation of state. The two MHD units support general equations of state and multi-species fluids. Also, in order to prevent negative pressures and temperatures, a separate equation for internal energy is solved in a fashion described earlier in the hydrodynamics chapter.

The APIs of the MHD units are fairly minimal. The units honor all of hydrodynamics unit variables, interface functions and runtime parameters described in the above hydrodynamics unit chapter (see Chapter 14). In addition, both the eight-wave and the unsplit staggered mesh units declare additional plasma variables and runtime parameters, which are listed in Table 14.7 and Table 14.8.

14.3.2 Usage

In the current release, the eight-wave unit serves as a default MHD solver. In order to choose the unsplit staggered mesh unit for MHD simulations, users need to include $+$usm in a setup script. The default eight-wave unit will be automatically chosen if there is no such specification included.

A word of caution

The eight-wave solver is only compatible with native grid interpolation in AMR simulations. This is because the solver only uses two layers of guard cells in each coordinate direction. The choice $\texttt{-gridinterpolation=native}$ is automatically adopted if $+$8wave is specified in setup, otherwise, $\texttt{-gridinterpolation=native}$ should be explicitly included in order to use the eight-wave solver without specifying $+$8wave. For instance, running a script $./setup\text{magnetoHD/BrioWu -1d -auto +8wave}$ will properly setup the BrioWu problem for the 8Wave solver, and $./setup\text{magnetoHD/BrioWu -1d -auto +usm}$ for the StaggeredMesh solver.
Table 14.7: Additional solution variables used in the MHD units.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>magx</td>
<td>PER,VOLUME</td>
<td>x-component of magnetic field</td>
</tr>
<tr>
<td>magy</td>
<td>PER,VOLUME</td>
<td>y-component of magnetic field</td>
</tr>
<tr>
<td>magz</td>
<td>PER,VOLUME</td>
<td>z-component of magnetic field</td>
</tr>
<tr>
<td>magp</td>
<td>GENERIC</td>
<td>magnetic pressure</td>
</tr>
<tr>
<td>divb</td>
<td>GENERIC</td>
<td>divergence of magnetic field</td>
</tr>
</tbody>
</table>

Table 14.8: Additional runtime parameters used in the MHD units.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitSystem</td>
<td>string</td>
<td>“none”</td>
<td>System of units in which MHD calculations are to be performed. Acceptable values are “none”, “CGS” and “SI”.</td>
</tr>
<tr>
<td>killdivb</td>
<td>logical</td>
<td>.true.</td>
<td>Enable/disable divergence cleaning.</td>
</tr>
<tr>
<td>flux_correct</td>
<td>logical</td>
<td>.true.</td>
<td>Enable/disable flux correction on AMR grid.</td>
</tr>
</tbody>
</table>

Supported configurations

Both MHD units currently support the uniform grid with FIXEDBLOCKSIZE and NONFIXEDBLOCKSIZE modes, and the adaptive grid with PARAMESH on Cartesian geometries, as well as 2D cylindrical (R-Z). When using AMR grids, the eight-wave unit supports both PARAMESH 2 and PARAMESH 4, while only PARAMESH 4 is supported in the unsplit staggered mesh solver because face-centered variables are only fully supported in PARAMESH 4.

14.3.3 Algorithm: The Unsplit Staggered Mesh Solver

A directionally unsplit staggered mesh algorithm (USM), which solves ideal and non-ideal MHD governing equations (14.17) ~ (14.20) in multiple dimensions, is a new MHD solver. Since FLASH4-beta, a full 3D implementation has been included as a new official release (Lee, submitted, 2012). The unsplit staggered mesh unit is based on a finite-volume, high-order Godunov method combined with a constrained transport (CT) type of scheme which ensures the solenoidal constraint of the magnetic fields on a staggered mesh geometry. In this approach, the cell-centered variables such as the plasma mass density $\rho$, plasma momentum density $\rho v$ and total plasma energy $\rho E$ are updated via a second-order MUSCL-Hancock unsplit space-time integrator using the high-order Godunov fluxes. The rest of the cell face-centered (staggered) magnetic fields are updated using Stokes’ Theorem as applied to a set of induction equations, enforcing the divergence-free constraint of the magnetic fields. Notice that this divergence-free constraint is automatically guaranteed and satisfied in pure one-dimensional MHD simulations, but special care must be taken in multidimensional problems.

The overall procedure of the unsplit staggered mesh scheme can be broken up into the following steps (Lee, 2006; Lee and Deane, 2009; Lee, submitted, 2012):

- **Quasi-linearization**: This step replaces the nonlinear system (14.17) ~ (14.20) with an approximate, quasi-linearized system of equations.
- **Data Reconstruction-evolution**: This routine calculates and evolves cell interface values by half time step using a second-order MUSCL-Hancock TVD algorithm (Toro, 1999). The approach makes use of a new method of ‘multidimensional characteristic analysis’ that can be achieved in one single step, incorporating all flux contributions from both normal and transverse directions without requiring any
need of solving a set of Riemann problems (that is usually adopted in transverse flux updates). In this step the USM scheme includes the multidimensional MHD terms in both normal and transverse directions, satisfying a perfect balance law for the terms proportional to $\nabla \cdot \mathbf{B}$ in the induction equations.

- **An intermediate Riemann problem:** An intermediate set of high-order Godunov fluxes is calculated using the cell interface values obtained from the data reconstruction-evolution step. The resulting fluxes are then used to evolve the normal fields by a half time step in the next procedure.

- **A half time step update for the normal fields:** The normal magnetic fields are evolved by a half time step using the flux-CT method at cell interfaces, ensuring the divergence-free property on a staggered grid. This intermediate update for the normal fields and the half time step data from the data reconstruction-evolution step together provide a second-order accurate MHD states at cell interfaces.

- **Riemann problem:** Using the second-order MHD states calculated from the above procedures, the scheme proceeds to solve the Riemann problem to obtain high-order Godunov fluxes at cell interfaces.

- **Unsplit update of cell-centered variables:** The unsplit time integrations are performed using the high-order Godunov fluxes to update the cell-centered variables for the next time step.

- **Construction of electric fields:** Using the high-order Godunov fluxes, the cell-cornered (edged in 3D) electric fields are constructed. The unsplit staggered mesh scheme computes a new modified electric field construction (MEC) scheme that includes first and second multidimensional derivative terms in Taylor expansions for high-order interpolations. This modified electric field construction provides enhanced accuracy by explicitly adding proper amounts of dissipation as well as spatial gradients in its interpolation scheme.

- **Flux-CT scheme:** The electric fields from the MEC scheme are used to evolve the cell face-centered magnetic fields by solving a set of discrete induction equations. The resulting magnetic fields satisfy the divergence-free constraint up to the accuracy of machine round-off errors.

- **Reconstruct cell-centered magnetic fields:** The cell-centered magnetic fields are reconstructed from the divergence free cell face-centered magnetic fields by taking arithmetic averages of the cell face-centered fields variables.

Note that the procedure required in solving one-dimensional MHD equations is much simpler than solving the multidimensional ones and only involves the first through third and the fifth steps in the above outlined scheme. The choices of TVD slope limiters available in the unsplit staggered mesh scheme (see Table 14.9) includes the minmod limiter as well as the compressible limiters such as vanLeer or mc limiter. Another choice, called hybrid limiter, can be used to provide a mixed type of limiters as described in Balsara (2004). In this choice, one uses a compressible limiter to produce a crisp representation for linearly degenerate waves (e.g., an entropy wave and left- and right-going Alfvén waves). To this end, a compressible limiter can be applied to the density and the magnetic fields variables, where these variables contribute much of the variations in such linearly degenerate wave families. Other variables, the velocity field components and pressure, constitute four genuinely nonlinear wave families (e.g., left- and right-going fast/slow magneto-sonic waves) in MHD. These genuinely nonlinear wave families inherently behave according to their self steepening mechanism and one can simply use a diffusive but robust minmod limiter. Another limiter, called limited, is also available (see details in Toro, 1999, 2nd Ed., section 13.8.4), and users need to specify a runtime parameter $\beta$ (LimitedSlopeBeta in flash.par) if this limiter is chosen for a simulation.

The unsplit staggered mesh unit solves a set of discrete induction equations in multi-dimensional problems to proceed temporal evolutions of the staggered magnetic fields using electric fields. For instance, in a two-dimensional staggered grid, the unsplit staggered mesh unit solves a two-dimensional pair of discrete induction equations that were found originally by Yee (1966):

\[
\begin{align*}
\frac{b_{x,i+1/2,j}^{n+1}}{\Delta t} &= b_{x,i+1/2,j}^{n} - \frac{\Delta t}{\Delta y} \left\{ -E_{x,i+1/2,j+1/2}^{n+1} + E_{x,i+1/2,j-1/2}^{n+1} \right\}, \\
\frac{b_{y,i+1/2,j}^{n+1}}{\Delta t} &= b_{y,i+1/2,j}^{n} - \frac{\Delta t}{\Delta x} \left\{ -E_{y,i+1/2,j+1/2}^{n+1} + E_{y,i-1/2,j+1/2}^{n+1} \right\}.
\end{align*}
\]  

(14.24)  

(14.25)
The superindex $n+1/2$ in the above equations simply indicates an intermediate timestep right after the temporal update of the cell-centered variables.

A three-dimensional schematic figure of the staggered grid geometry with collocations of edge-based values (electric fields $E$) and face based values (magnetic fields $b$) is shown in Figure 14.2.

One of the main advantages of using the CT-type of scheme is that the cell face-centered magnetic fields $b_{x,i+1/2,j}^{n+1}$ and $b_{y,i,j+1/2}^{n+1}$, which are updated via the above induction equations, satisfy the divergence-free constraint locally. The numerical divergence of the magnetic fields is defined as

$$\nabla \cdot \mathbf{B}_{i,j}^{n+1} = \frac{b_{x,i+1/2,j}^{n+1} - b_{x,i-1/2,j}^{n+1}}{\Delta x} + \frac{b_{y,i,j+1/2}^{n+1} - b_{y,i,j-1/2}^{n+1}}{\Delta y}$$ (14.26)

and it remains zero to the accuracy of machine round-off errors, provided that $\nabla \cdot \mathbf{B}_{i,j}^n = 0$.

On an AMR grid, the unsplit staggered mesh scheme uses a direct injection method as a default to preserve divergence-free prolongation to the cell face-centered fields variables. This method is one of the simplest approaches that is offered by PARAMESH 4 to maintain the divergence-free constraint in prolongation. This simple method ensures the solenoidal constraint well enough where the fields are varying smoothly, but can introduce oscillations in regions of steep field gradient. In such cases Balsara’s prolongation algorithm can be useful. Both prolongation algorithms are supported and enabled using runtime parameters in the unsplit staggered mesh solver (see Table 14.9 below).

To solve the above induction equations (14.24) and (14.25) in a flux-CT type scheme, it is required to construct cell edge-based electric fields. The simplest choice is to use the cell face-centered high-order Godunov fluxes and take an arithmetic average to construct cell-cornered (edge-based in 3D) electric fields:

$$E_{z,i+1/2,j+1/2}^{n+1/2} = \frac{1}{4} \left\{ - F_{B_x,i+1/2,j}^{n+1/2} - F_{B_y,i+1/2,j}^{n+1/2} + G_{B_x,i,j+1/2}^{n+1/2} + G_{B_y,i,j+1/2}^{n+1/2} \right\}$$

$$= \frac{1}{4} \left\{ E_{z,i+1/2,j}^{n+1/2} + E_{z,i+1/2,j+1}^{n+1/2} + E_{z,i,j+1/2}^{n+1/2} + E_{z,i+1,j+1/2}^{n+1/2} \right\},$$ (14.27)

where $F_{B_x}$ and $G_{B_x}$ represent the $x$ and $y$ high-order Godunov flux components corresponding to the magnetic fields $B_y$ and $B_x$, respectively (see details in Balsara and Spicer, 1999).

A high-order accurate version is also available by turning on a logical switch $E_{\text{modification}}$ in the unsplit staggered mesh scheme, which takes Taylor series expansions of the cell-cornered electric field $E_{z,i+1/2,j+1/2}^{n+1/2}$ in all directions, followed by taking an arithmetic average of them (Lee, 2006; Lee and Deane, 2009).

The last step in the unsplit staggered mesh scheme is to reconstruct the cell-centered magnetic fields $B_{x,i,j}$ and $B_{y,i,j}$ from the divergence-free face-centered magnetic fields. The unsplit staggered mesh scheme takes arithmetic averages of the face-centered fields variables to obtain the cell-centered magnetic fields, which is sufficient for second order accuracy. After obtaining the new cell-centered magnetic fields, the total plasma energy may need to be corrected in order to preserve the positivity of the thermal temperature and pressure (Balsara and Spicer, 1999; Tóth, 2000). This energy correction is very useful especially in problems involving very low $\beta$ plasma flows.

There are several choices available for calculating high order Godunov fluxes in the unsplit staggered mesh scheme. The default solver is Roe’s linearized approximate solver, which takes into account all seven waves that arise in the MHD equations. The Roe solver can adopt one of the two entropy fix routines (Harten, 1983; Harten and Hyman, 1983) in order to avoid unphysical states near strong rarefaction regions. As all seven waves are considered in Roe’s solver, high numerical resolutions can be achieved in most cases. However, Roe’s solver still can fail to maintain positive states near very low densities even with the entropy fix. In this case, computationally efficient and positively conservative Riemann solvers such as HLLC (Seiferth et al., 1991), HLLC (S. Li, 2005), or HLLD (Miyoshi and Kusano, 2005) can be used to maintain positive states in simulations. A hybrid type of Riemann solver which combines using the Roe solver for high accuracy and HLLD for stability is also available.

The USM solver has been recently extended also for 2D and 2.5D cylindrical (R-Z) geometries, both for uniform grids and AMR. In the cylindrical implementation, we followed the guidelines of Mignone et al. (2007) and Skinner & Ostriker (2010). Special care was also taken to ensure a divergence free interpolation of the staggered magnetic field components, when grid movements occur in AMR. This novel prolongation scheme is based on the methods described in Balsara (2001, 2004) and Li & Li (2004). More information
Table 14.9: Runtime parameters used in the unsplit staggered mesh MHD (physics/Hydro/HydroMain/unsplit/MHD_StaggeredMesh) solver additional to those described for the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro_Unsplit).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>killdivb</td>
<td>logical</td>
<td>.true.</td>
<td>On/off ( \nabla \cdot \mathbf{B} = 0 ) handling on the staggered grid</td>
</tr>
<tr>
<td>E_modification</td>
<td>logical</td>
<td>.true.</td>
<td>Enable/disable high-order electric field construction</td>
</tr>
<tr>
<td>E_upwind</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable an upwind update for induction equations</td>
</tr>
<tr>
<td>energyFix</td>
<td>logical</td>
<td>.true.</td>
<td>Enable/disable energy correction</td>
</tr>
<tr>
<td>facevar2ndOrder</td>
<td>logical</td>
<td>.true.</td>
<td>Turn on/off a second-order facevar update</td>
</tr>
<tr>
<td>ForceHydroLimit</td>
<td>logical</td>
<td>.false.</td>
<td>On/off pure Hydro mode</td>
</tr>
<tr>
<td>prolMethod</td>
<td>string</td>
<td>&quot;INJECTION_PROL&quot;</td>
<td>Use either direct injection method (&quot;INJECTION_PROL&quot;) or Balsara’s method (&quot;BALSARA_PROL&quot;) in prolonging divergence-free magnetic fields stored in face-centered variables</td>
</tr>
<tr>
<td>RiemannSolver</td>
<td>string</td>
<td>&quot;ROE&quot;</td>
<td>&quot;HLLD&quot; is additionally available for MHD, &quot;Hybrid&quot; is also available for MHD.</td>
</tr>
</tbody>
</table>

regarding the cylindrical implementation can be found in Tzeferacos et al. (2012, in print) whereas new test problems, provided with this release, are available at Sections 25.2.3 and 25.2.4. Handling of different geometries will be available in future releases.

14.3.3.1 Slowly moving shock handling in PPM

A new dissipative mechanism is a hybridized slope limiter for PPM that combines a new upwind biased slope limiter with a conventional TVD slope limiter (Lee, D., Astronum Proc. 2010). This hybrid upwind limiter reduces spurious numerical oscillations near discontinuities, and therefore can compute sharp, monotonized profiles in compressible flows when using PPM, especially in Magnetohydrodynamics (MHD) slowly moving shock regions. (See more in Chapter 25.2.1.)

By the nature of very small numerical dissipations in PPM, unphysical oscillations in discontinuous MHD solutions can appear in a specific flow region, referred to as a slowly moving shock (SMS) relative to the grid. This new approach handles numerical non-oscillatory MHD solutions using PPM in SMS regions. The SMS should not be confused with so-called "slow MHD shock" which corresponds to two slow waves (i.e., \( u \pm c_s \)) in MHD, where \( c_s \) is the slow magneto-acoustic velocity.

The method first detects a local, slowly moving shock, and considers an upwind direction to compute a monotonicity-preserving slope limiter. This new approach, in addition to improving the numerical solutions in MHD to levels that reduce (or eliminate) such oscillatory behaviors while preserving sharp discontinuities in MHD, is also simple to implement. The method has been verified against the results from other high-resolution shock-capturing (HRSC) methods such as MUSCL and WENO schemes.

In order to enable the SMS treatment for PPM, users set a runtime parameter `upwindTVD=.true.` in `flash.par`. The SMS method does require to have an extended stencil, and users should specify `+supportWENO`, or `+supportPPMUpwind` in setup. See more in Section 14.1.3.
Figure 14.2: A 3D control volume on the staggered grid with the cell center at \((i, j, k)\). The magnetic fields are collocated at the cell face centers and the electric fields at the cell edge centers. The line integral of the electric fields \( \int \partial F \cdot n \cdot E \cdot T \, dl \) along the four edges of the face \( F_{x, i+1/2, j, k} \) gives rise to the negative of the rate of change of the magnetic field flux in \( x \)-direction through the area enclosed by the four edges (e.g., the area of \( F_{x, i+1/2, j, k} \)).
14.3. MAGNETOHYDRODYNAMICS (MHD)

**Stability limit**

As described in the unsplit hydro solver unit (physics/Hydro/HydroMain/unsplitHydro_Unsplit), the USM MHD solver can take a wide range of CFL limits in all three dimensions (e.g., $CFL < 1$). However, in some circumstances where there are strong shocks and rarefactions, `shockDetect=.true.` could be useful to gain more numerical stability by using. It is also helpful to use (1) artificial viscosity and flattening, or (2) lower order reconstruction scheme (e.g., MH), or (2) diffusive Riemann solver such as HLL-type, or LLF solvers, or (3) a reduced CFL accordingly.

**Divergence-free prolongation of magnetic fields on AMR in the unsplit staggered mesh solver**

It is of importance to preserve divergence-free evolutions of magnetic fields in MHD simulations. Moreover, some special cares are required in prolonging divergence-free magnetic fields on AMR grids. One simple straightforward way in this aspect is to prolong divergence-free fields to newly created children blocks using direct injection. This injection method therefore inherently preserves divergence-free properties on AMR block structures and works well in most cases. This method is a default in the unsplit staggered mesh solver and can also be enabled by setting a runtime parameter `prolMethod = "INJECTION_PROL"`. Another way, proposed by Balsara (2001), is also available in the unsplit staggered mesh solver and can be chosen by setting `prolMethod = "BALSARA_PROL"`. Both prolongation methods are supported in MHD's 2.5D and 3D simulations. In 2 and 2.5D cylindrical geometry however, since neither method takes into account geometrical factors, we use a modified prolongation algorithm based on Balsara (2004) and Li&Li (2004). This is the default option and is activated by choosing `prolMethod = "BALSARA_PROL"`. The need for this special refinement requires to have an MHD's own customized implementation of `Simulation_customizeProlong.F90` placed in the source/Simulation/SimulationMain/magnetoHD/.

14.3.4 Algorithm: The Eight-wave Solver

The eight-wave magnetohydrodynamic (MHD) unit is based on a finite-volume, cell-centered method that was proposed by Powell *et al.* (1999). The unit uses directional splitting to evolve the magnetohydrodynamics equations. Like the PPM and RHD units, this MHD unit makes one sweep in each spatial direction to advance physical variables from one time level to the next. In each sweep, the unit uses AMR functionality to fill in guard cells and impose boundary conditions. Then it reconstructs characteristic variables and uses these variables to compute time-averaged interface fluxes of conserved quantities. In order to enforce conservation at jumps in refinement, the unit makes flux conservation calls to AMR, which redistributes affected fluxes using the appropriate geometric area factors. Finally, the unit updates the solution and calls the EOS unit to ensure thermodynamical consistency.

After all sweeps are completed, the MHD unit enforces magnetic field divergence cleaning. In the current release only a diffusive divergence cleaning method (truncation-error method), which was the default method in FLASH2, is supported and the later release of the code will incorporate an elliptic projection cleaning method.

The ideal part of the MHD equations are solved using a high-resolution, finite-volume numerical scheme with MUSCL-type (van Leer, 1979) limited gradient reconstruction. In order to maximize the accuracy of the solver, the reconstruction procedure is applied to characteristic variables. Since this may cause certain variables, such as density and pressure, to fall outside of physically meaningful bounds, extra care is taken in the limiting step to prevent this from happening. All other variables are calculated in the unit from the interpolated characteristic variables.
In order to resolve discontinuous Riemann problems that occur at computational cell interfaces, the code by default employs a Roe-type solver derived in Powell et al. (1999). This solver provides full characteristic decomposition of the ideal MHD equations and is, therefore, particularly useful for plasma flow simulations that feature complex wave interaction patterns. An alternative Riemann solver of HLLE type, provided in FLASH2, has not been incorporated into FLASH4 yet.

Time integration in the MHD unit is done using a second-order, one-step method due to Hancock (Toro, 1999). For linear systems with unlimited gradient reconstruction, this method can be shown to coincide with the classic Lax-Wendroff scheme.

A difficulty particularly associated with solving the MHD equations numerically lies in the solenoidality of the magnetic field. The notorious $\nabla \cdot B = 0$ condition, a strict physical law, is very hard to satisfy in discrete computations. Being only an initial condition of the MHD equations, it enters the equations indirectly and is not, therefore, guaranteed to be generally satisfied unless special algorithmic provisions are made. Without discussing this issue in much detail, which goes well beyond the scope of this user’s guide (for example, see Tóth (2000) and references therein), we will remind the user that there are three commonly accepted methods to enforce the $\nabla \cdot B$ condition: the elliptic projection method (Brackbill and Barnes, 1980), the constrained transport method (Evans and Hawley, 1988), and the truncation-level error method (Powell et al., 1999). In the current release, the truncation-error cleaning methods is provided for the eight-wave MHD unit, and the constrained transport method is implemented for the unsplit staggered mesh MHD units (see Section 14.3.3 for details).

In the truncation-error method, the solenoidality of the magnetic field is enforced by including several terms proportional to $\nabla \cdot B$. This removes the effects of unphysical magnetic tension forces parallel to the field and stimulates passive advection of magnetic monopoles, if they are spuriously created. In many applications, this method has been shown to be an efficient and sufficient way to generate solutions of high physical quality. However, it has also been shown (Tóth, 2000) that this method can sometimes, for example in strongly discontinuous and stagnated flows, lead to accumulation of magnetic monopoles, whose strength is sufficient to corrupt the physical quality of computed solutions. In order to eliminate this deficiency, the eight-wave MHD code also uses a simple yet very effective method originally due to Marder (1987) to destroy the magnetic monopoles on the scale on which they are generated. In this method, a diffusive operator proportional to $\nabla \nabla \cdot B$ is added to the induction equation, so that the equation becomes

$$\frac{\partial B}{\partial t} + \nabla \cdot (vB - Bv) = -v \nabla \cdot B + \eta_a \nabla \nabla \cdot B,$$

(14.28)

with the artificial diffusion coefficient $\eta_a$ chosen to match that of grid numerical diffusion. In the FLASH code, $\eta_a = \frac{\lambda}{2} \left( \frac{1}{2\Delta x} + \frac{1}{2\Delta y} + \frac{1}{2\Delta z} \right)^{-1}$, where $\lambda$ is the largest characteristic speed in the flow. Since the grid magnetic diffusion Reynolds number is always on the order of unity, this operator locally destroys magnetic monopoles at the rate at which they are created. Recent numerical experiments (Powell et al., 2001) indicate that this approach can very effectively bring the strength of spurious magnetic monopoles to levels that are sufficiently low, so that generated solutions remain physically consistent. The entire $\nabla \cdot B$ control process is local and very inexpensive compared to other methods. Moreover, one can show that this process is asymptotically convergent (Munz et al., 2000), and each of its applications is equivalent to one Jacobi iteration in solving the Poisson equation in the elliptic projection method. The caveat is that this method only suppresses but does not completely eliminate magnetic monopoles. Whether this is acceptable depends on the particular physical problem.

As an alternative way to eliminate magnetic monopoles completely, the FLASH2 code includes an elliptic projection method, in which the unphysical divergence of the magnetic field can be removed to any desired level down to machine precision. As yet, this method has not been made available in FLASH4 and will be supported in a later release.

### 14.3.5 Non-ideal MHD

Non-ideal terms (magnetic resistive, viscous and heat conduction terms) can be enabled or disabled in FLASH at run time via the `flash.par` file. For example, a typical `flash.par` file for non-ideal runtime parameters should look more or less like this:

```plaintext
# Magnetic Resistivity
```
One way of simulating a diffusive process is to add those physics units in a simulation Config file. For example, a snippet of Config file can look like this:

```plaintext
# Magnetic Resistivity
REQUIRES physics/materialProperties/MagneticResistivity/MagneticResistivityMain

# Viscosity
REQUIRES physics/materialProperties/Viscosity/ViscosityMain

# Conductivity
REQUIRES physics/materialProperties/Conductivity/ConductivityMain/Constant-diff

# Diffusive time step calculation
REQUIRES physics/sourceTerms/Diffuse/DiffuseMain
```

New treatments has been made in including the non-ideal diffusive terms in the USM-MHD solver in the FLASH3.2 release and remain unchanged since then. In the previous releases before FLASH3.2, all the non-ideal diffusive terms had to be grouped together in the "resistive MHD" part of the unit by turning the flag variable `useMagneticResistivity` on, and the non-ideal terms were included all together. In the FLASH3.2 release, each individual term can be separately included by turning each corresponding logical variables in run time. The Diffusion time step is computed using the `Diffuse_computeDt.F90` routine in `Driver_computeDt.F90` to provide a more consistent way of computing a non-ideal time step.
15.1 Introduction

The Eos unit implements the equation of state needed by the hydrodynamics and nuclear burning solvers. The function Eos provides the interface for operating on a one-dimensional vector. The same interface can be used for a single cell by reducing the vector size to 1. Additionally, this function can be used to find the thermodynamic quantities either from the density, temperature, and composition or from the density,
internal energy, and composition. For user’s convenience, a wrapper function (Eos_wrapped) is provided, which takes a section of a block and translates it into the data format required by the Eos function, then calls the function. Upon return from the Eos function, the wrapper translates the returned data back to the same section of the block.

Four implementations of the (Eos) unit are available in the current release of FLASH4: Gamma, which implements a perfect-gas equation of state; Gamma/RHD, which implements a perfect-gas equation taking relativistic effects into account; Multigamma, which implements a perfect-gas equation of state with multiple fluids, each of which can have its own adiabatic index (γ); and Helmholtz, which uses a fast Helmholtz free-energy table interpolation to handle degenerate/relativistic electrons/positrons and includes radiation pressure and ions (via the perfect gas approximation).

As described in previous sections, FLASH evolves the Euler equations for compressible, inviscid flow. This system of equations must be closed by an additional equation that provides a relation between the thermodynamic quantities of the gas. This relationship is known as the equation of state for the material, and its structure and properties depend on the composition of the gas.

It is common to call an equation of state (henceforth EOS) routine more than 10^9 times during a two-dimensional simulation and more than 10^{11} times during the course of a three-dimensional simulation of stellar phenomena. Thus, it is very desirable to have an EOS that is as efficient as possible, yet accurately represents the relevant physics. While FLASH is capable of using any general equation of state, we discuss here the three equation of state routines that are supplied: an ideal-gas or gamma-law EOS, an EOS for a fluid composed of multiple gamma-law gases, and a tabular Helmholtz free energy EOS appropriate for stellar interiors. The two gamma-law EOSs consist of simple analytic expressions that make for a very fast EOS routine both in the case of a single gas or for a mixture of gases. The Helmholtz EOS includes much more physics and relies on a table look-up scheme for performance.

### 15.2 Gamma Law and Multigamma

FLASH uses the method of Colella & Glaz (1985) to handle general equations of state. General equations of state contain 4 adiabatic indices (Chandrasekhar 1939), but the method of Colella & Glaz parameterizes the EOS and requires only two of the adiabatic indices. The first is necessary to calculate the adiabatic sound speed and is given by

\[ \gamma_1 = \frac{\rho}{\frac{\partial P}{\partial \rho}}. \]  

(15.1)

The second relates the pressure to the energy and is given by

\[ \gamma_4 = 1 + \frac{P}{\rho \epsilon}. \]  

(15.2)

These two adiabatic indices are stored as the mesh-based variables GAMC_VAR and GAME_VAR. All EOS routines must return \( \gamma_1 \), and \( \gamma_4 \) is calculated from (15.2).

The gamma-law EOS models a simple ideal gas with a constant adiabatic index \( \gamma \). Here we have dropped the subscript on \( \gamma \), because for an ideal gas, all adiabatic indices are equal. The relationship between pressure \( P \), density \( \rho \), and specific internal energy \( \epsilon \) is

\[ P = (\gamma - 1) \rho \epsilon. \]  

(15.3)

We also have an expression relating pressure to the temperature \( T \)

\[ P = \frac{N_a k}{\bar{A}} \rho T, \]  

(15.4)

where \( N_a \) is the Avogadro number, \( k \) is the Boltzmann constant, and \( \bar{A} \) is the average atomic mass, defined as

\[ \frac{1}{\bar{A}} = \sum_i X_i \frac{A_i}{A}, \]  

(15.5)
where $X_i$ is the mass fraction of the $i$th element. Equating these expressions for pressure yields an expression for the specific internal energy as a function of temperature

$$\epsilon = \frac{1}{\gamma - 1} \frac{N_a k}{A} T .$$

(15.6)

The relativistic variant of the ideal gas equation is explained in more detail in Section 14.2.

Simulations are not restricted to a single ideal gas; the multigamma EOS provides routines for simulations with several species of ideal gases each with its own value of $\gamma$. In this case the above expressions hold, but $\gamma$ represents the weighted average adiabatic index calculated from

$$\frac{1}{(\gamma - 1)} = \bar{A} \sum_i \frac{1}{(\gamma_i - 1)} A_i .$$

(15.7)

We note that the analytic expressions apply to both the forward (internal energy as a function of density, temperature, and composition) and backward (temperature as a function of density, internal energy and composition) relations. Because the backward relation requires no iteration in order to obtain the temperature, this EOS is quite inexpensive to evaluate. Despite its fast performance, use of the gamma-law EOS is limited, due to its restricted range of applicability for astrophysical problems.

15.2.1 Ideal Gamma Law for Relativistic Hydrodynamics

The relativistic variant of the ideal gas equation is explained in more detail in Section 14.2.

15.3 Helmholtz

The Helmholtz EOS provided with the FLASH distribution contains more physics and is appropriate for addressing astrophysical phenomena in which electrons and positrons may be relativistic and/or degenerate and in which radiation may significantly contribute to the thermodynamic state. Full details of the Helmholtz equation of state are provided in Timmes & Swesty (1999). This EOS includes contributions from radiation, completely ionized nuclei, and degenerate/relativistic electrons and positrons. The pressure and internal energy are calculated as the sum over the components

$$P_{\text{tot}} = P_{\text{rad}} + P_{\text{ion}} + P_{\text{ele}} + P_{\text{pos}} + P_{\text{coul}}$$

(15.8)

$$\epsilon_{\text{tot}} = \epsilon_{\text{rad}} + \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{pos}} + \epsilon_{\text{coul}} .$$

(15.9)

Here the subscripts “rad,” “ion,” “ele,” “pos,” and “coul” represent the contributions from radiation, nuclei, electrons, positrons, and corrections for Coulomb effects, respectively. The radiation portion assumes a blackbody in local thermodynamic equilibrium, the ion portion (nuclei) is treated as an ideal gas with $\gamma = 5/3$, and the electrons and positrons are treated as a non-interacting Fermi gas.

The blackbody pressure and energy are calculated as

$$P_{\text{rad}} = \frac{aT^4}{3}$$

(15.10)

$$\epsilon_{\text{rad}} = \frac{3P_{\text{rad}}}{\rho}$$

(15.11)

where $a$ is related to the Stephan-Boltzmann constant $\sigma_B = ac/4$, and $c$ is the speed of light. The ion portion of each routine is the ideal gas of (Equations 15.3 – 15.4) with $\gamma = 5/3$. The number densities of free electrons $N_{\text{ele}}$ and positrons $N_{\text{pos}}$ in the noninteracting Fermi gas formalism are given by

$$N_{\text{ele}} = \frac{8\pi \sqrt{2}}{\hbar^3} m_e^3 c^3 \beta^{3/2} \left[ F_{1/2}(\eta, \beta) + F_{3/2}(\eta, \beta) \right]$$

(15.12)
\[ N_{\text{pos}} = \frac{8\pi \sqrt{2}}{h^3} m_e^3 c^3 \beta^{3/2} \left[ F_{1/2}(-\eta - 2/\beta, \beta) + \beta F_{3/2}(-\eta - 2/\beta, \beta) \right], \tag{15.13} \]

where \( h \) is Planck’s constant, \( m_e \) is the electron rest mass, \( \beta = kT/(m_e c^2) \) is the relativity parameter, \( \eta = \mu/kT \) is the normalized chemical potential energy \( \mu \) for electrons, and \( F_k(\eta, \beta) \) is the Fermi-Dirac integral

\[ F_k(\eta, \beta) = \int_0^\infty \frac{x^k (1 + 0.5 \beta x)^{1/2}}{\exp(x - \eta) + 1} \, dx. \tag{15.14} \]

Because the electron rest mass is not included in the chemical potential, the positron chemical potential must have the form \( \eta_{\text{pos}} = -\eta - 2/\beta \). For complete ionization, the number density of free electrons in the matter is

\[ N_{\text{ele, matter}} = \frac{Z}{A} N_a \rho = \bar{Z} N_{\text{ion}}, \tag{15.15} \]

and charge neutrality requires

\[ N_{\text{ele, matter}} = N_{\text{ele}} - N_{\text{pos}}. \tag{15.16} \]

Solving this equation with a standard one-dimensional root-finding algorithm determines \( \eta \). Once \( \eta \) is known, the Fermi-Dirac integrals can be evaluated, giving the pressure, specific thermal energy, and entropy due to the free electrons and positrons. From these, other thermodynamic quantities such as \( \gamma \) and \( S \) are found. Full details of this formalism may be found in Fryxell et al. (2000) and references therein.

The above formalism requires many complex calculations to evaluate the thermodynamic quantities, and routines for these calculations typically are designed for accuracy and thermodynamic consistency at the expense of speed. The Helmholtz EOS in FLASH provides a table of the Helmholtz free energy (hence the name) and makes use of a thermodynamically consistent interpolation scheme obviating the need to perform the complex calculations required of the above formalism during the course of a simulation. The interpolation scheme uses a bi-quintic Hermite interpolant resulting in an accurate EOS that performs reasonably well.

The Helmholtz free energy,

\[ F = \epsilon - T S \tag{15.17} \]

\[ dF = -S \, dT + \frac{P}{\rho^2} \, d\rho, \tag{15.18} \]

is the appropriate thermodynamic potential for use when the temperature and density are the natural thermodynamic variables. The free energy table distributed with FLASH was produced from the Timmes EOS (Timmes & Arnett 1999). The Timmes EOS evaluates the Fermi-Dirac integrals (15.14) and their partial derivatives with respect to \( \eta \) and \( \beta \) to machine precision with the efficient quadrature schemes of Aparicio (1998) and uses a Newton-Raphson iteration to obtain the chemical potential of (15.16). All partial derivatives of the pressure, entropy, and internal energy are formed analytically. Searches through the free energy table are avoided by computing hash indices from the values of any given \( (T, \rho \bar{Z}/A) \) pair. No computationally expensive divisions are required in interpolating from the table; all of them can be computed and stored the first time the EOS routine is called.

We note that the Helmholtz free energy table is constructed for only the electron-positron plasma, and it is a 2-dimensional function of density and temperature, i.e. \( F(\rho, T) \). It is made with \( \bar{A} = Z = 1 \) (pure hydrogen), with an electron fraction \( Y_e = 1 \). One reason for not including contributions from photons and ions in the table is that these components of the Helmholtz EOS are very simple (Equations 15.10 – 15.11), and one doesn’t need fancy table look-up schemes to evaluate simple analytical functions. A more important reason for only constructing an electron-positron EOS table with \( Y_e = 1 \) is that the 2-dimensional table is valid for any composition. Separate planes for each \( Y_e \) are not necessary (or desirable), since simple multiplication by \( Y_e \) in the appropriate places gives the desired composition scaling. If photons and ions were included in the table, then this valuable composition independence would be lost, and a 3-dimensional table would be necessary.

The Helmholtz EOS has been subjected to considerable analysis and testing (Timmes & Swesty 2000), and particular care was taken to reduce the numerical error introduced by the thermodynamical models.
15.4. MULTITEMPERATURE EXTENSION FOR EOS

Table 15.1: 1T and multitemperature Eos modes. The symbols are defined in constants.h. The column “1-T” indicates whether invoking Eos with this mode implies and/or ensures that the three components are in temperature equilibrium; a “yes” here implies that all temperatures provided as outputs are equal, and equal to T if T is listed as an input. For modes where component temperatures are allowed to stay different, the combined $T$ has no physical meaning but indicates an “average” temperature to which the components would equilibrate if cells were held fixed without exchange of heat with neighboring cells. The columns RS indicates modes where, in the current multitemperature Eos implementations, a Newton-Raphson root search may be performed internally in order to compute some combined fluid quantities, in particular $T$. In this column “(y)” means the search is only necessary to get $T$ and can otherwise be omitted. Subscripts i, e, and r stand for ions, electrons, and radiation, respectively, $\epsilon$ for specific internal energy, $s$ for specific entropy. Note that combined density $\rho$ is always assumed an input and omitted from the table. Material composition and ionization levels are currently also just inputs.

<table>
<thead>
<tr>
<th>symbol</th>
<th>inputs</th>
<th>outputs</th>
<th>1-T</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE_DENS_TEMP</td>
<td>$T$</td>
<td>$\epsilon, P$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MODE_DENS_TEMP_EQUI</td>
<td>$T$</td>
<td>$\epsilon, P$, $T_1=T_e=T_r$, $\epsilon_i, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MODE_DENS_TEMP_GATHER</td>
<td>$T_1, T_e, T_r$</td>
<td>$T, \epsilon, P$, $\epsilon_i, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MODE_DENS_EI</td>
<td>$\epsilon$</td>
<td>$T, P$, $T_1=T_e=T_r$, $\epsilon_i, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>yes</td>
<td>(y)</td>
</tr>
<tr>
<td>MODE_DENS_EI_SCATTER</td>
<td>$\epsilon$</td>
<td>$T, P$, $T_1=T_e=T_r$, $\epsilon_i, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>no</td>
<td>(y)</td>
</tr>
<tr>
<td>MODE_DENS_EI_GATHER</td>
<td>$\epsilon_i, \epsilon_e, \epsilon_r$</td>
<td>$T, P$, $T_1, T_e, T_r$, $\epsilon_i, \epsilon_e, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>no</td>
<td>(y)</td>
</tr>
<tr>
<td>MODE_DENS_EI_SELE_GATHER</td>
<td>$\epsilon_i, \epsilon_e, \epsilon_r$</td>
<td>$T, P$, $T_1, T_e, T_r$, $\epsilon_i, \epsilon_e, \epsilon_r$, $s_e$, $P_i, P_e, P_r$, ...</td>
<td>no</td>
<td>(y)</td>
</tr>
<tr>
<td>MODE_DENS_PRES</td>
<td>$P$</td>
<td>$T, \epsilon$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

below the formal accuracy of the hydrodynamics algorithm (Fryxell, et al. 2000; Timmes & Swesty 2000). The physical limits of the Helmholtz EOS are $10^{-10} < \rho < 10^{11}$ (g cm$^{-3}$) and $10^4 < T < 10^{11}$ (K). As with the gamma-law EOS, the Helmholtz EOS provides both forward and backward relations. In the case of the forward relation ($\rho, T$, given along with the composition) the table lookup scheme and analytic formulae directly provide relevant thermodynamic quantities. In the case of the backward relation ($\rho, \epsilon$, and composition given), the routine performs a Newton-Raphson iteration to determine temperature. It is possible for the input variables to be changed in the iterative modes since the solution is not exact. The returned quantities are thermodynamically consistent.

15.4 Multitemperature extension for Eos

Extended functionality is required of the Eos in order to work a multitemperature simulation. Such simulations need to use one of the implementations under the multiTemp directory. When the Hydro multiTemp code is used, as described in Section 14.1.4, one of the implementations under physics/Eos/EosMain/multiTemp must be used. Additional functionality includes

- Support for additional Hydro variables that describe the state for the fluid components (ions, electrons, radiation).
- Additional Eos modes. See Table 15.1.

15.4.1 Gamma

The multitemperature Gamma EOS implementation models the ion as well as the electron components as ideal gases, and the radiation component as black body thermal radiation (like the Helmholtz 1T implementation, see Section 15.3). It is a limitation of this implementation that the ionization state of ions is assumed to be fixed.

This Eos implementation has several runtime parameters, in part inherited from the 1T implementation, that can technically be changed but for which different values can lead to a nonsensical or inconsistent model.
In particular, the values of `gamma`, `gammaIon`, `gammaEle`, and `gammaRad` should not be changed from their default values (5/3. for the first four and 4/3. for the latter). Other values have not been tested.

This EOS can include contributions from radiation, partially or completely ionized nuclei, and electrons. The desired (constant) ionization level $\bar{Z}$ of the nuclei should be specified with the `eos_singleSpeciesZ` runtime parameter. The `eos_singleSpeciesA` runtime parameter specifies the is the average atomic mass $\bar{A}$ and should be set to the mass of one atom of the material in atomic mass units. For example, for a plasma of fully ionized Carbon-12, one would set `eos_singleSpeciesA = 12.0` and `eos_singleSpeciesA = 6.0`.

The combined pressure and internal energy are calculated as the sum over the components

$$P_{tot} = P_{ion} + P_{ele} + P_{rad}$$  \hspace{0.8cm} (15.19)

$$\epsilon_{tot} = \epsilon_{ion} + \epsilon_{ele} + \epsilon_{rad}.$$  \hspace{0.8cm} (15.20)

Here the subscripts “ion,” “ele,” and “rad” represent the contributions from nuclei, electrons, and radiation, respectively. The radiation portion assumes a black body in local thermodynamic equilibrium, the ion portion (nuclei) is treated as an ideal gas with $\gamma = 5/3$, and the electrons are treated as a classical ideal gas as well.

As for the 1T Helmholtz implementation, the black body pressure and energy relate to the radiation “temperature” by

$$P_{rad} = \frac{a T_{rad}^4}{3}$$  \hspace{0.8cm} (15.21)

$$\epsilon_{rad} = \frac{3P_{rad}}{\rho}$$  \hspace{0.8cm} (15.22)

where $a$ is related to the Stephan-Boltzmann constant $\sigma_B = ac/4$, and $c$ is the speed of light.

Like 1T implementations, All multitemperature implementations of Eos must return $\gamma = \gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial P}$, and $\gamma_4$ is calculated from (15.2). These two generalizations of adiabatic indices are usually stored as the mesh-based variables `GAMC_VAR` and `GAME_VAR` as a result of calling `Eos_wrapped`.

15.4.1.1 Gamma/Ye

The Ye variant of Gamma implementation is like the Gamma implementation, except that $\bar{A}$ and $\bar{Z}$ need not be constant throughout the domain but can vary in different parts of the fluid.

Typically, a simulation would implement a `Simulation_initBlock` that initializes mass scalar variables `sumy` and `ye` as desired. The initial values then get advected with the motion of the fluid. Eos interprets them as

$$\text{sumy} = \sum Y_i$$  \hspace{0.8cm} (15.23)

and

$$\text{ye} = Y_e$$  \hspace{0.8cm} (15.24)

and computes

$$\bar{A} = \frac{\sum Y_i Y_i}{\sum Y_i}$$  \hspace{0.8cm} (15.25)

and

$$\bar{Z} = \frac{\bar{A}}{Y_e}$$  \hspace{0.8cm} (15.26)

from them.
15.4.2 Multigamma

First a clarification, to avoid confusion: We will call ions, electrons, and radiation the three components of the fluid; and we designate different kinds of materials as different species. A fluid with several materials contributes its ions to the common ion component and its electrons to the common electron component. Currently, all adiabatic coefficients $\gamma_i$ describing the electrons and the various kinds of ions should be $5/3$ as for a monatomic ideal gas (despite the implementation’s name!). Mixing of species in a cell results in a fluid whose matter components are described by average values $\overline{A}$ and $\overline{Z}$, which depends on the local composition as well as on the $A_i$ and $Z_i$ values of the various species.

In our multitemperature approximation, the three components may have different temperatures, but there is no notion of a per-species temperature.

The multitemperature Multigamma implementation is another variation on the multitemperature Gamma theme. Except for the different way of getting $\overline{A}$ and $\overline{Z}$, all else goes as described for Gamma, above.

15.4.3 Tabulated

The Tabulated Eos implementation is currently only available for simulations configured in 3T mode. The currently supported table formats are IONMIX1 and IONMIX4. See the Opacity section Section 21.4.3 for a description of the IONMIX1 and IONMIX4 table file format.

Tables are read into memory at initialization. For IONMIX4, tables for $Z_i(T, N_{ion})$, $Z_e(T, N_{ion})$, $E_i(T, N_{ion})$, $E_e(T, N_{ion})$, $P_i(T, N_{ion})$, $P_e(T, N_{ion})$ are used.

Eos modes like MODE_DENS_TEMP, MODE_DENS_TEMP_component, MODE_DENS_TEMP_EQUI compute N_{ion} from dens and directly look up the output quantities in the appropriate tables. For other modes, root searches would be required.

However, Tabulated Eos implementation is currently not being used on its own but only in connection with the Multitype implementation.

15.4.4 Multitype

Multitype implementation does Eos computations by combining Eos calls for different materials. A material is a species in the sense of the Multispecies unit. Properties of materials relevant to the Multitype Eos are thus kept in the Multispecies database.

The Multitype implementation can combine materials of different eos types. If can be used for 3T Eos modes. When used in this generally, we speak of MTMMMT Eos - multitemperature multimaterial multitype.

Currently, the Eos types that can be combined are:

- Gamma
- Tabulated

The Multitype implementation calls eos_idealGamma and eos_tabIonmix only in MODE_DENS_TEMP_component modes.

The prescription for combining results of such calls is currently simple: Make Eos calls for different materials separately, giving passing the material’s partial density into the Eos. Ther per-material calls are thus not aware of other materials in the same cell. Results are combined as appropriate: pressures are taken as partial pressures and are added up to give the total multimaterial pressure; specific energies are multiplied with partial densities to give energy densities, which are added up.

For 3T modes, the above is applied separately for ion and electron components, resulting in separate total energies, pressures, etc. for ions and electrons. Newton-Raphson root search is done to implement Eos modes that take energies etc. as inputs MODE_DENS_EI*, MODE_DENS_PRES [not completely implemented *], separately for ions and electrons if necessary. Blackbody radiation component is added.

To use MTMMMT, use the +mtmmmt setup shortcut. See LaserSalb simulation for an example. See Multispecies unit on initializing material properties.
15.5 Usage

15.5.1 Initialization

The initialization function of the Eos unit Eos_init is fairly simple for the two ideal gas gamma law implementations included. It gathers the runtime parameters and the physical constants needed by the equation of state and stores them in the data module. The Helmholtz EOS Eos_init routine is a little more complex. The Helmholtz EOS requires an input file helm_table.dat that contains the lookup table for the electron contributions. This table is currently stored in ASCII for portability purposes. When the table is first read in, a binary version called helm_table.bdat is created. This binary format can be used for faster subsequent restarts on the same machine but may not be portable across platforms. The Eos_init routine reads in the table data on processor 0 and broadcasts it to all other processors.

15.5.2 Runtime Parameters

Runtime parameters for the Gamma unit require the user to set the thermodynamic properties for the single gas. gamma, eos.singleSpeciesA, eos.singleSpeciesZ set the ratio of specific heats and the nucleon and proton numbers for the gas. In contrast, the Multigamma implementation does not set runtime parameters to define properties of the multiple species. Instead, the simulation Config file indicates the requested species, for example helium and oxygen can be defined as

SPECIES HE4
SPECIES O16

The properties of the gases are initialized in the file Simulation_initSpecies.F90, for example

```fortran
subroutine Simulation_initSpecies()
  use Multispecies_interface, ONLY : Multispecies_setProperty
  implicit none
  #include "Flash.h"
  #include "Multispecies.h"
  call Multispecies_setProperty(HE4_SPEC, A, 4.)
  call Multispecies_setProperty(HE4_SPEC, Z, 2.)
  call Multispecies_setProperty(HE4_SPEC, GAMMA, 1.66666666667e0)
  call Multispecies_setProperty(O16_SPEC, A, 16.0)
  call Multispecies_setProperty(O16_SPEC, Z, 8.0)
  call Multispecies_setProperty(O16_SPEC, GAMMA, 1.4)
end subroutine Simulation_initSpecies
```

For the Helmholtz equation of state, the table-lookup algorithm requires a given temperature and density. When temperature or internal energy are supplied as the input parameter, an iterative solution is found. Therefore, no matter what mode is selected for Helmholtz input, the best initial value of temperature should be provided to speed convergence of the iterations. The iterative solver is controlled by two runtime parameters eos_maxNewton and eos_tolerance which define the maximum number of iterations and convergence tolerance. An additional runtime parameter for Helmholtz, eos_coulombMult, indicates whether or not to apply Coulomb corrections. In some regions of the ρ-T plane, the approximations made in the Coulomb corrections may be invalid and result in negative pressures. When the parameter eos_coulombMult is set to zero, the Coulomb corrections are not applied.

15.5.3 Direct and Wrapped Calls

The primary function in the Eos unit, Eos, operates on a vector, taking density, composition, and either temperature, internal energy, or pressure as input, and returning γ₁, and either the pressure, temperature or internal energy (whichever was not used as input). This equation of state interface is useful for initializing a problem. The user is given direct control over the input and output, since everything is passed through the argument list. Also, the vector data format is more efficient than calling the equation of state routine directly.
on a point by point basis, since it permits pipelining and provides better cache performance. Certain optional quantities such as electron pressure, degeneracy parameter, and thermodynamic derivatives can be calculated by the \texttt{Eos} function if needed. These quantities are selected for computation based upon a logical mask array provided as an input argument. A .true. value in the mask array results in the corresponding quantity being computed and reported back to the calling function. Examples of calling the basic implementation \texttt{Eos} are provided in the API description, see \texttt{Eos}.

The hydrodynamic and burning computations repeatedly call the Eos function to update pressure and temperature during the course of their calculation. Typically, values in all the cells of the block need of be updated in these calls. Since the primary Eos interface requires the data to be organized as a vector, using it directly could make the code in the calling unit very cumbersome and error prone. The wrapper interface, \texttt{Eos\_wrapped} provides a means by which the details of translating the data from block to vector and back are hidden from the calling unit. The wrapper interface permits the caller to define a section of block by giving the limiting indices along each dimension. The \texttt{Eos\_wrapped} routine translates the block section thus described into the vector format of the \texttt{Eos} interface, and upon return translates the vector format back to the block section. This wrapper routine cannot calculate the optional derivative quantities. If they are needed, call the \texttt{Eos} routine directly with the optional mask set to true and space allocated for the returned quantities.

\section{15.6 Unit Test}

The unit test of the Eos function can exercise all three implementations. Because the Gamma law allows only one species, the setup required for the three implementations is specific. To invoke any three-dimensional Eos unit test, the command is:

\texttt{./setup unitTest/Eos/implementation \ -auto \ -3d}

where \texttt{implementation} is one of \texttt{Gamma}, \texttt{Multigamma}, \texttt{Helmholtz}. The Eos unit test works on the assumption that if the four physical variables in question (density, pressure, energy and temperature) are in thermal equilibrium with one another, then applying the equation of state to any two of them should leave the other two completely unchanged. Hence, if we initialize density and temperature with some arbitrary values, and apply the equation of state to them in \texttt{MODE DENS TEMP}, then we should get pressure and energy values that are thermodynamically consistent with density and temperature. Now after saving the original temperature value, we apply the equation of state to density and newly calculated pressure. The new value of the temperature should be identical to the saved original value. This verifies that the Eos unit is computing correctly in \texttt{MODE DENS PRES} mode. By repeating this process for the remaining two modes, we can say with great confidence that the Eos unit is functioning normally.

In our implementation of the Eos unit test, the initial conditions applied to the domain create a gradient for density along the \texttt{x} axis and gradients for temperature and pressure along the \texttt{y} axis. If the test is being run for the Multigamma or Helmholtz implementations, then the species are initialized to have gradients along the \texttt{z} axis.
Chapter 16

Local Source Terms

Figure 16.1: The organizational structure of physics source terms, which include units such as Burn and Stir. Shaded units include only stub implementations. Only a subset of units is shown.
The physics/sourceTerms organizational directory contains several units that implement forcing terms. The Burn, Stir, Ionize, and Diffuse units contain implementations in FLASH4. Two other units, Cool and Heat, contain only stub level routines in their API.

16.1 Burn Unit

The nuclear burning implementation of the Burn unit uses a sparse-matrix semi-implicit ordinary differential equation (ODE) solver to calculate the nuclear burning rate and to update the fluid variables accordingly (Timmes 1999). The primary interface routines for this unit are Burn_init, which sets up the nuclear isotope tables needed by the unit, and Burn, which calls the ODE solver and updates the hydrodynamical variables in a single row of a single block. The routine Burn_computeDt may limit the computational timestep because of burning considerations. There is also a helper routine Simulation/SimulationComposition/-Simulation_initSpecies (see Simulation_initSpecies) which provides the properties of ions included in the burning network.

16.1.1 Algorithms

Modeling thermonuclear flashes typically requires the energy generation rate due to nuclear burning over a large range of temperatures, densities and compositions. The average energy generated or lost over a period of time is found by integrating a system of ordinary differential equations (the nuclear reaction network) for the abundances of important nuclei and the total energy release. In some contexts, such as supernova models, the abundances themselves are also of interest. In either case, the coefficients that appear in the equations are typically extremely sensitive to temperature. The resulting stiffness of the system of equations requires the use of an implicit time integration scheme.

A user can choose between two implicit integration methods and two linear algebra packages in FLASH. The runtime parameter odeStepper controls which integration method is used in the simulation. The choice odeStepper = 1 is the default and invokes a Bader-Deuflhard scheme. The choice odeStepper = 2 invokes a Kaps-Rentrop or Rosenbrock scheme. The runtime parameter algebra controls which linear algebra package is used in the simulation. The choice algebra = 1 is the default and invokes the sparse matrix MA28 package. The choice algebra = 2 invokes the GIFT linear algebra routines. While any combination of the integration methods and linear algebra packages will produce correct answers, some combinations may execute more efficiently than others for certain types of simulations. No general rules have been found for best
combination for a given simulation. The most efficient combination depends on the timestep being taken, the spatial resolution of the model, the values of the local thermodynamic variables, and the composition. Users are advised to experiment with the various combinations to determine the best one for their simulation. However, an extensive analysis was performed in the Timmes paper cited below.

Timmes (1999) reviewed several methods for solving stiff nuclear reaction networks, providing the basis for the reaction network solvers included with FLASH. The scaling properties and behavior of three semi-implicit time integration algorithms (a traditional first-order accurate Euler method, a fourth-order accurate Kaps-Rentrop / Rosenbrock method, and a variable order Bader-Deuflhard method) and eight linear algebra packages (LAPACK, LUDCMP, LEQS, GIFT, MA28, UMFPACK, and Y12M) were investigated by running each of these 24 combinations on seven different nuclear reaction networks (hard-wired 13- and 19-isotope networks and soft-wired networks of 47, 76, 127, 200, and 489 isotopes). Timmes’ analysis suggested that the best balance of accuracy, overall efficiency, memory footprint, and ease-of-use was provided by the two integration methods (Bader-Deuflhard and Kaps-Rentrop) and the two linear algebra packages (MA28 and GIFT) that are provided with the FLASH code.

### 16.1.2 Reaction networks

We begin by describing the equations solved by the nuclear burning unit. We consider material that may be described by a density $\rho$ and a single temperature $T$ and contains a number of isotopes $i$, each of which has $Z_i$ protons and $A_i$ nucleons (protons + neutrons). Let $n_i$ and $\rho_i$ denote the number and mass density, respectively, of the $i$th isotope, and let $X_i$ denote its mass fraction, so that

$$X_i = \frac{\rho_i}{\rho} = \frac{n_iA_i}{(\rho N_A)} \ , \quad (16.1)$$

where $N_A$ is Avogadro’s number. Let the molar abundance of the $i$th isotope be

$$Y_i = X_i/A_i = \frac{n_i}{(\rho N_A)} \ . \quad (16.2)$$

Mass conservation is then expressed by

$$\sum_{i=1}^{N} X_i = 1 \ . \quad (16.3)$$

At the end of each timestep, FLASH checks that the stored abundances satisfy (16.3) to machine precision in order to avoid the unphysical buildup (or decay) of the abundances or energy generation rate. Roundoff errors in this equation can lead to significant problems in some contexts (e.g., classical nova envelopes), where trace abundances are important.

The general continuity equation for the $i$th isotope is given in Lagrangian formulation by

$$\frac{dY_i}{dt} + \nabla \cdot (Y_iV_i) = \dot{R}_i \ . \quad (16.4)$$

In this equation $\dot{R}_i$ is the total reaction rate due to all binary reactions of the form $i(j,k)l$,

$$\dot{R}_i = \sum_{j,k} Y_jY_k\lambda_{kj(l)} - Y_iY_j\lambda_{jk(i)} \ , \quad (16.5)$$

where $\lambda_{kj}$ and $\lambda_{jk}$ are the reverse (creation) and forward (destruction) nuclear reaction rates, respectively. Contributions from three-body reactions, such as the triple-$\alpha$ reaction, are easy to append to (16.5). The mass diffusion velocities $V_i$ in (16.4) are obtained from the solution of a multicomponent diffusion equation (Chapman & Cowling 1970; Burgers 1969; Williams 1988) and reflect the fact that mass diffusion processes arise from pressure, temperature, and/or abundance gradients as well as from external gravitational or electrical forces.

The case $V_i \equiv 0$ is important for two reasons. First, mass diffusion is often unimportant when compared to other transport processes, such as thermal or viscous diffusion (i.e., large Lewis numbers and/or small Prandtl numbers). Such a situation obtains, for example, in the study of laminar flame fronts propagating through the quiescent interior of a white dwarf. Second, this case permits the decoupling of the reaction
network solver from the hydrodynamical solver through the use of operator splitting, greatly simplifying the algorithm. This is the method used by the default FLASH distribution. Setting $V_i = 0$ transforms (16.4) into
\[ \frac{dY_i}{dt} = R_i, \] (16.6)
which may be written in the more compact, standard form
\[ \dot{y} = f(y). \] (16.7)

Stated another way, in the absence of mass diffusion or advection, any changes to the fluid composition are due to local processes.

Because of the highly nonlinear temperature dependence of the nuclear reaction rates and because the abundances themselves often range over several orders of magnitude in value, the values of the coefficients which appear in (16.6) and (16.7) can vary quite significantly. As a result, the nuclear reaction network equations are “stiff.” A system of equations is stiff when the ratio of the maximum to the minimum eigenvalue of the Jacobian matrix $J \equiv \partial f / \partial y$ is large and imaginary. This means that at least one of the isotopic abundances changes on a much shorter timescale than another. Implicit or semi-implicit time integration methods are generally necessary to avoid following this short-timescale behavior, requiring the calculation of the Jacobian matrix.

It is instructive at this point to look at an example of how (16.6) and the associated Jacobian matrix are formed. Consider the $^{12C}(\alpha, \gamma)^{16O}$ reaction, which competes with the triple-$\alpha$ reaction during helium burning in stars. The rate $R$ at which this reaction proceeds is critical for evolutionary models of massive stars, since it determines how much of the core is carbon and how much of the core is oxygen after the initial helium fuel is exhausted. This reaction sequence contributes to the right-hand side of (16.7) through the terms
\[ \dot{Y}(^{4He}) = -Y(^{4He})Y(^{12C})R + \ldots \]
\[ \dot{Y}(^{12C}) = -Y(^{4He})Y(^{12C})R + \ldots \]
\[ \dot{Y}(^{16O}) = +Y(^{4He})Y(^{12C})R + \ldots, \] (16.8)
where the ellipses indicate additional terms coming from other reaction sequences. The minus signs indicate that helium and carbon are being destroyed, while the plus sign indicates that oxygen is being created. Each of these three expressions contributes two terms to the Jacobian matrix $\dot{J} = \partial f / \partial y$
\[ J(^{4He}, ^{4He}) = -Y(^{12C})R + \ldots \]
\[ J(^{12C}, ^{4He}) = -Y(^{4He})R + \ldots \]
\[ J(^{16O}, ^{4He}) = +Y(^{12C})R + \ldots \] (16.9)

Entries in the Jacobian matrix represent the flow, in number of nuclei per second, into (positive) or out of (negative) an isotope. All of the temperature and density dependence is included in the reaction rate $R$. The Jacobian matrices that arise from nuclear reaction networks are neither positive-definite nor symmetric, since the forward and reverse reaction rates are generally not equal. In addition, the magnitudes of the matrix entries change as the abundances, temperature, or density change with time.

This release of FLASH4 contains three reaction networks. A seven-isotope alpha-chain ($^{10}_{\text{iso7}}$) is useful for problems that do not have enough memory to carry a larger set of isotopes. The 13-isotope $\alpha$-chain plus heavy-ion reaction network ($^{13}_{\text{aprox13}}$) is suitable for most multi-dimensional simulations of stellar phenomena, where having a reasonably accurate energy generation rate is of primary concern. The 19-isotope reaction network ($^{19}_{\text{aprox19}}$) has the same $\alpha$-chain and heavy-ion reactions as the 13-isotope network, but it includes additional isotopes to accommodate some types of hydrogen burning (PP chains and steady-state CNO cycles), along with some aspects of photo-disintegration into $^{54}_{\text{Fe}}$. This 19 isotope reaction network is described in Weaver, Zimmerman, & Woosley (1978).

The networks supplied with FLASH are examples of a “hard-wired” reaction network, where each of the reaction sequences are carefully entered by hand. This approach is suitable for small networks, when minimizing the CPU time required to run the reaction network is a primary concern, although it suffers the disadvantage of inflexibility.
16.1.2.1 Two linear algebra packages: MA28 and GIFT

As mentioned in the previous section, the Jacobian matrices of nuclear reaction networks tend to be sparse, and they become more sparse as the number of isotopes increases. Since implicit or semi-implicit time integration schemes generally require solving systems of linear equations involving the Jacobian matrix, taking advantage of the sparsity can significantly reduce the CPU time required to solve the systems of linear equations.

The MA28 sparse matrix package used by FLASH is described by Duff, Erisman, & Reid (1986). This package, which has been described as the “Coke classic” of sparse linear algebra packages, uses a direct – as opposed to an iterative – method for solving linear systems. Direct methods typically divide the solution of $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ into a symbolic LU decomposition, a numerical LU decomposition, and a backsubstitution phase. In the symbolic LU decomposition phase, the pivot order of a matrix is determined, and a sequence of decomposition operations that minimizes the amount of fill-in is recorded. Fill-in refers to zero matrix elements which become nonzero (e.g., a sparse matrix times a sparse matrix is generally a denser matrix). The matrix is not decomposed; only the steps to do so are stored. Since the nonzero pattern of a chosen nuclear reaction network does not change, the symbolic LU decomposition is a one-time initialization cost for reaction networks. In the numerical LU decomposition phase, a matrix with the same pivot order and nonzero pattern as a previously factorized matrix is numerically decomposed into its lower-upper form. This phase must be done only once for each set of linear equations. In the backsubstitution phase, a set of linear equations is solved with the factors calculated from a previous numerical decomposition. The backsubstitution phase may be performed with as many right-hand sides as needed, and not all of the right-hand sides need to be known in advance.

MA28 uses a combination of nested dissection and frontal envelope decomposition to minimize fill-in during the factorization stage. An approximate degree update algorithm that is much faster (asymptotically and in practice) than computing the exact degrees is employed. One continuous real parameter sets the amount of searching done to locate the pivot element. When this parameter is set to zero, no searching is done and the diagonal element is the pivot, while when set to unity, partial pivoting is done. Since the matrices generated by reaction networks are usually diagonally dominant, the routine is set in FLASH to use the diagonal as the pivot element. Several test cases showed that using partial pivoting did not make a significant accuracy difference but was less efficient, since a search for an appropriate pivot element had to be performed. MA28 accepts the nonzero entries of the matrix in the $(i, j, a_{i,j})$ coordinate system and typically uses 70–90% less storage than storing the full dense matrix.

GIFT is a program which generates Fortran subroutines for solving a system of linear equations by Gaussian elimination (Gustafson, Liniger, & Willoughby 1970; Müller 1997). The full matrix $\mathbf{A}$ is reduced to upper triangular form, and backsubstitution with the right-hand side $\mathbf{b}$ yields the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. GIFT generated routines skip all calculations with matrix elements that are zero; in this restricted sense, GIFT generated routines are sparse, but the storage of a full matrix is still required. It is assumed that the pivot element is located on the diagonal and no row or column interchanges are performed, so GIFT generated routines may become unstable if the matrices are not diagonally dominant. These routines must decompose the matrix for each right-hand side in a set of linear equations. GIFT writes out (in Fortran code) the sequence of Gaussian elimination and backsubstitution steps without any do loop constructions on the matrix $A(i, j)$. As a result, the routines generated by GIFT can be quite large. For the 489 isotope network discussed by Timmes (1999), GIFT generated $\sim 5.0 \times 10^7$ lines of code! Fortunately, for small reaction networks (less than about 30 isotopes), GIFT generated routines are much smaller and generally faster than other linear algebra packages.

The FLASH runtime parameter algebra controls which linear algebra package is used in the simulation. algebra $= 1$ is the default choice and invokes the sparse matrix MA28 package. algebra $= 2$ invokes the GIFT linear algebra routines.

16.1.2.2 Two time integration methods

One of the time integration methods used by FLASH for evolving the reaction networks is a 4th-order accurate Kaps-Rentrop, or Rosenbrock method. In essence, this method is an implicit Runge-Kutta algorithm. The
reaction network is advanced over a timestep $h$ according to

$$y^{n+1} = y^n + \sum_{i=1}^{4} b_i \Delta_i , \quad (16.10)$$

where the four vectors $\Delta^i$ are found from successively solving the four matrix equations

\begin{align*}
(\mathbf{I}/\gamma h - \mathbf{J}) \cdot \Delta_1 &= f(y^n) \quad (16.11) \\
(\mathbf{I}/\gamma h - \mathbf{J}) \cdot \Delta_2 &= f(y^n + a_{21} \Delta_1) + c_{21} \Delta_1 / h \quad (16.12) \\
(\mathbf{I}/\gamma h - \mathbf{J}) \cdot \Delta_3 &= f(y^n + a_{31} \Delta_1 + a_{32} \Delta_2) + (c_{31} \Delta_1 + c_{32} \Delta_2) / h \quad (16.13) \\
(\mathbf{I}^2 h - \mathbf{J}) \cdot \Delta_4 &= f(y^n + a_{31} \Delta_1 + a_{32} \Delta_2) + (c_{41} \Delta_1 + c_{42} \Delta_2 + c_{43} \Delta_3) / h . \quad (16.14)
\end{align*}

$b_i$, $\gamma$, $a_{ij}$, and $c_{ij}$ are fixed constants of the method. An estimate of the accuracy of the integration step is made by comparing a third-order solution with a fourth-order solution, which is a significant improvement over the basic Euler method. The minimum cost of this method — which applies for a single timestep that meets or exceeds a specified integration accuracy — is one Jacobian evaluation, three evaluations of the right-hand side, one matrix decomposition, and four backsubstitutions. Note that the four matrix equations represent a staged set of linear equations ($\Delta_4$ depends on $\Delta_3 \ldots$ depends on $\Delta_1$). Not all of the right-hand sides are known in advance. This general feature of higher-order integration methods impacts the optimal choice of a linear algebra package. The fourth-order Kaps-Rentrop routine in FLASH makes use of the routine GRK4T given by Kaps & Rentrop (1979).

Another time integration method used by FLASH for evolving the reaction networks is the variable order Bader-Deuflhard method (e.g., Bader & Deuflhard 1983). The reaction network is advanced over a large timestep $H$ from $y^n$ to $y^{n+1}$ by the following sequence of matrix equations. First,

\begin{align*}
h &= H/m \\
(\mathbf{I} - \mathbf{J}) \cdot \Delta_0 &= h f(y^n) \\
y_1 &= y^n + \Delta_0 . \quad (16.15)
\end{align*}

Then from $k = 1, 2, \ldots, m - 1$

\begin{align*}
(\mathbf{I} - \mathbf{J}) \cdot x &= h f(y_k) - \Delta_{k-1} \\
\Delta_k &= \Delta_{k-1} + 2x \\
y_{k+1} &= y_k + \Delta_k , \quad (16.16)
\end{align*}

and closure is obtained by the last stage

\begin{align*}
(\mathbf{I} - \mathbf{J}) \cdot \Delta_m &= h [f(y_m) - \Delta_{m-1}] \\
y^{n+1} &= y_m + \Delta_m . \quad (16.17)
\end{align*}

This staged sequence of matrix equations is executed at least twice with $m = 2$ and $m = 6$, yielding a fifth-order method. The sequence may be executed a maximum of seven times, which yields a fifteenth-order method. The exact number of times the staged sequence is executed depends on the accuracy requirements (set to one part in $10^6$ in FLASH) and the smoothness of the solution. Estimates of the accuracy of an integration step are made by comparing the solutions derived from different orders. The minimum cost of this method — which applies for a single timestep that met or exceeded the specified integration accuracy — is one Jacobian evaluation, eight evaluations of the right-hand side, two matrix decompositions, and ten backsubstitutions. This minimum cost can be increased at a rate of one decomposition (the expensive part) and $m$ backsubstitutions (the inexpensive part) for every increase in the order $2k+1$. The cost of increasing the order is compensated for, hopefully, by being able to take correspondingly larger (but accurate) timestep. The controls for order versus step size are a built-in part of the Bader-Deuflhard method. The cost per step of this integration method is at least twice as large as the cost per step of either a traditional first-order
accurate Euler method or the fourth-order accurate Kaps-Rentrop discussed above. However, if the Bader-Deuflhard method can take accurate timesteps that are at least twice as large, then this method will be more efficient globally. Timmes (1999) shows that this is typically (but not always!) the case. Note that in Equations 16.15 – 16.17, not all of the right-hand sides are known in advance, since the sequence of linear equations is staged. This staging feature of the integration method may make some matrix packages, such as MA28, a more efficient choice.

The FLASH runtime parameter \texttt{odeStepper} controls which integration method is used in the simulation. The choice \texttt{odeStepper = 1} is the default and invokes the variable order Bader-Deuflhard scheme. The choice \texttt{odeStepper = 2} invokes the fourth order Kaps-Rentrop / Rosenbrock scheme.

16.1.3 Detecting shocks

For most astrophysical detonations, the shock structure is so thin that there is insufficient time for burning to take place within the shock. However, since numerical shock structures tend to be much wider than their physical counterparts, it is possible for a significant amount of burning to occur within the shock. Allowing this to happen can lead to unphysical results. The burner unit includes a multidimensional shock detection algorithm that can be used to prevent burning in shocks. If the \texttt{useShockBurn} parameter is set to \texttt{false}, this algorithm is used to detect shocks in the Burn unit and to switch off the burning in shocked cells.

Currently, the shock detection algorithm supports Cartesian and 2-dimensional cylindrical coordinates. The basic algorithm is to compare the jump in pressure in the direction of compression (determined by looking at the velocity field) with a shock parameter (typically 1/3). If the total velocity divergence is negative and the relative pressure jump across the compression front is larger than the shock parameter, then a cell is considered to be within a shock.

This computation is done on a block by block basis. It is important that the velocity and pressure variables have up-to-date guard cells, so a guard cell call is done for the burners only if we are detecting shocks (\textit{i.e.} \texttt{useShockBurning = false}).

16.1.4 Energy generation rates and reaction rates

The instantaneous energy generation rate is given by the sum

\[ \dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{dY_i}{dt}. \]  

(16.18)

Note that a nuclear reaction network does not need to be evolved in order to obtain the instantaneous energy generation rate, since only the right hand sides of the ordinary differential equations need to be evaluated. It is more appropriate in the FLASH program to use the average nuclear energy generated over a timestep

\[ \dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{\Delta Y_i}{\Delta t}. \]  

(16.19)

In this case, the nuclear reaction network does need to be evolved. The energy generation rate, after subtraction of any neutrino losses, is returned to the FLASH program for use with the operator splitting technique.

The tabulation of Caughlan & Fowler (1988) is used in FLASH for most of the key nuclear reaction rates. Modern values for some of the reaction rates were taken from the reaction rate library of Hoffman (2001, priv. comm.). A user can choose between two reaction rate evaluations in FLASH. The runtime parameter \texttt{useBurnTable} controls which reaction rate evaluation method is used in the simulation. The choice \texttt{useBurnTable = 0} is the default and evaluates the reaction rates from analytical expressions. The choice \texttt{useBurnTable = 1} evaluates the reactions rates from table interpolation. The reaction rate tables are formed on-the-fly from the analytical expressions. Tests on one-dimensional detonations and hydrostatic burnings suggest that there are no major differences in the abundance levels if tables are used instead of the analytic expressions; we find less than 1% differences at the end of long timescale runs. Table interpolation is about 10 times faster than evaluating the analytic expressions, but the speedup to FLASH is more modest, a few percent at best, since reaction rate evaluation never dominates in a real production run.
Finally, nuclear reaction rate screening effects as formulated by Wallace et al. (1982) and decreases in the energy generation rate $\dot{\epsilon}_{\text{nuc}}$ due to neutrino losses as given by Itoh et al. (1996) are included in FLASH.

### 16.1.5 Temperature-based timestep limiting

When using explicit hydrodynamics methods, a timestep limiter must be used to ensure the stability of the numerical solution. The standard CFL limiter is always used when an explicit hydrodynamics unit is included in FLASH. This constraint does not allow any information to travel more than one computational cell per timestep. When coupling burning with the hydrodynamics, the CFL timestep may be so large compared to the burning timescales that the nuclear energy release in a cell may exceed the existing internal energy in that cell. When this happens, the two operations (hydrodynamics and nuclear burning) become decoupled.

To limit the timestep when burning is performed, an additional constraint is imposed. The limiter tries to force the energy generation from burning to be smaller than the internal energy in a cell. The runtime parameter `enucDtFactor` controls this ratio. The timestep limiter is calculated as

$$\Delta t_{\text{burn}} = \text{enucDtFactor} \cdot \frac{E_{\text{int}}}{E_{\text{nuc}}}$$  \hspace{1cm} (16.20)

where $E_{\text{nuc}}$ is the nuclear energy, expressed as energy per volume per time, and $E_{\text{int}}$ is the internal energy per volume. For good coupling between the hydrodynamics and burning, `enucDtFactor` should be $< 1$. The default value is kept artificially high so that in most simulations the time limiting due to burning is turned off. Care must be exercised in the use of this routine.

### 16.2 Ionization Unit

The analysis of UV and X-ray observations, and in particular of spectral lines, is a powerful diagnostic tool of the physical conditions in astrophysical plasmas (e.g., the outer layers of the solar atmosphere, supernova remnants, etc.). Since deviation from equilibrium ionization may have a non-negligible effect on the UV and X-ray lines, it is crucial to take into account these effects in the modeling of plasmas and in the interpretation of the relevant observations.

In light of the above observations, FLASH contains the unit `physics/sourceTerms/Ionize/Ionize-Main/Nei`, which is capable of computing the density of each ion species of a given element taking into account non-equilibrium ionization (NEI). This is accomplished by solving a system of equations consisting of the fluid equations of the whole plasma and the continuity equations of the ionization species of the elements considered. The densities of the twelve most abundant elements in astrophysical material (He, C, N, O, Ne, Mg, Si, S, Ar, Ca, Fe, and Ni) plus fully ionized hydrogen and electrons can be computed by this unit.

The Euler equations plus the set of advection equations for all the ion species take the following form

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$$  \hspace{1cm} (16.21)

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P = \rho g$$  \hspace{1cm} (16.22)

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} \left[ + S \right]$$  \hspace{1cm} (16.23)

$$\frac{\partial n_i^Z}{\partial t} + \nabla \cdot n_i^Z \mathbf{v} = R_i^Z \quad (i = 1, \ldots, N_{\text{spec}}),$$  \hspace{1cm} (16.24)

where $\rho$ is the fluid density, $t$ is the time, $\mathbf{v}$ is the fluid velocity, $P$ is the pressure, $E$ is the sum of the internal energy and kinetic energy per unit mass, $\mathbf{g}$ is the acceleration due to gravity, $n_i^Z$ is the number density of the ion $i$ of the element $Z$, $N_{\text{spec}}$ is the total number of species, and

$$R_i^Z = N_e [n_i^Z + \alpha_i^Z S_i^Z - n_i^{Z-1} S_{i-1}^{Z-1} - n_i^{Z+1} S_i^{Z+1} - \alpha_i^{Z+1} S_i^{Z+1} - \alpha_i^{Z-1} S_i^{Z-1}].$$  \hspace{1cm} (16.25)

where $N_e$ is the electron number density, $\alpha_i^Z \equiv \alpha(N_e, T)$ are the collisional and dielectronic recombination coefficients, and $S_i^Z \equiv S(N_e, T)$ are the collisional ionization coefficients of Summers(1974).
Table 16.1: Runtime parameters used with the Ionize unit.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tneimin</td>
<td>real</td>
<td>$1.0 \times 10^4$</td>
<td>Min nei temperature</td>
</tr>
<tr>
<td>tneimax</td>
<td>real</td>
<td>$1.0 \times 10^7$</td>
<td>Max nei temperature</td>
</tr>
<tr>
<td>dneimin</td>
<td>real</td>
<td>$1.0 \times 10^4$</td>
<td>Min nei electron number density</td>
</tr>
<tr>
<td>dneimax</td>
<td>real</td>
<td>$1.0 \times 10^{12}$</td>
<td>Max nei electron number density</td>
</tr>
</tbody>
</table>

16.2.1 Algorithms

A fractional step method is required to integrate the equations and in particular to decouple the NEI solver from the hydro solver. For each timestep, the homogeneous hydrodynamic transport equations given by (16.21) are solved using the FLASH hydrodynamics solver with $R = 0$. After each transport step, the “stiff” system of ordinary differential equations for the NEI problem

$$\frac{\partial n_i^Z}{\partial t} = R_i^Z (i = 1, \ldots, N_{\text{spec}}) \quad (16.26)$$

are integrated. This step incorporates the reactive source terms. Within each grid cell, the above equations can be solved separately with a standard ODE method. Since this system is “stiff”, it is solved using the Bader-Deuflhard time integration solver with the MA28 sparse matrix package. Timmes (1999) has shown that these two algorithms together provide the best balance of accuracy and overall efficiency.

Note that in the present version, the contribution of the ionization and recombination to the energy equation (the bracketed term in (16.23)) is not accounted for. Also, it should be noted that the source term in the NEI unit implementation is adequate to solve the problem for optically thin plasma in the “coronal” approximation; just collisional ionization, auto-ionization, radiative recombination, and dielectronic recombination are considered.

16.2.2 Usage

In order to run a FLASH executable that uses the ionization unit, the ionization coefficients of Summers (1974) must be contained in a file named `summers_den_le8.rates` in the same directory as the executable when the simulation is run. This file is copied into the `object/` directory with the `Config` keyword `DATAFILES` in the `physics/sourceTerms/Ionize/IonizeMain` implementation.

The Ionize unit supplies the runtime parameters described in Table 16.1. There are two implementations of `physics/sourceTerms/Ionize/IonizeMain`: the default implementation, Nei (tested using Neitest (see Section 25.7.1)), and Eqi (untested in FLASH4). The former computes ion species for non-equilibrium ionization, and the latter computes ion species in the approximation of ionization equilibrium.

The Ionize unit requires that the subunit implementation `Simulation/SimulationComposition/-Ionize` be used to set up the ion species of the fluid. The ions are defined in a file `Simulation/SimulationComposition/Ionize/SpeciesList.txt`, however, the `Config` file in the simulation directory (e.g. `Simulation/SimulationMain/Neitest/Config`) defines which subset of these elements are to be used.

16.3 Stir Unit

The addition of driving terms in a hydrodynamical simulation can be a useful feature, for example, in generating turbulent flows or for simulating the addition of power on larger scales (e.g., supernova feedback into the interstellar medium). The Stir unit comes in two implementations: 1) the Generate implementation, in which a divergence-free, random time-correlated ‘stirring’ velocity is directly added at selected modes in the simulation and 2) theFromFile implementation, in which a stirring field is set up from data residing on a file. The FromFile implementation allows to set up identical stirring fields on different platforms, and thus comparisons can be made between different codes.
Before FLASH 4.2, the implementation now called Generate was the only one provided. It is still the default that is being used if one specifies just REQUIRES physics/sourceTerms/Stir in a Config file or -unit=physics/sourceTerms/Stir on the setup command line.

16.3.1 Stir Unit: Generate Implementation

In the generate implementation, the Stir unit directly adds a divergence-free, time-correlated ‘stirring’ velocity at selected modes in the simulation.

The time-correlation is important for modeling realistic driving forces. Most large-scale driving forces are time-correlated, rather than white-noise; for instance, turbulent stirring from larger scales will be correlated on timescales related to the lifetime of an eddy on the scale of the simulation domain. This time correlation will lead to coherent structures in the simulation that will be absent with white-noise driving.

For each mode at each timestep, six separate phases (real and imaginary in each of the three spatial dimensions) are evolved by an Ornstein-Uhlenbeck (OU) random process (Uhlenbeck 1930). The OU process is a zero-mean, constant-rms process, which at each step ‘decays’ the previous value by an exponential \( f = e^{(-\Delta t/\tau)} \) and then adds a Gaussian random variable with a given variance, weighted by a ‘driving’ factor \( \sqrt{(1 - f^2)} \). Since the OU process represents a velocity, the variance is chosen to be the square root of the specific energy input rate (set by the runtime parameter \( st\text{energy} \)) divided by the decay time \( \tau \) (\( st\text{decay} \)). In the limit that the timestep \( \Delta t \to 0 \), it is easily seen that the algorithm represents a linearly-weighted summation of the old state with the new Gaussian random variable.

By evolving the phases of the stirring modes in Fourier space, imposing a divergence-free condition is relatively straightforward. At each timestep, the solenoidal component of the velocities is projected out, leaving only the non-compressional modes to add to the velocities.

The velocities are then converted to physical space by a direct Fourier transform – i.e., adding the sin and cos terms explicitly. Since most drivings involve a fairly small number of modes, this is more efficient than an FFT, since the FFT would need large numbers of modes (equal to six times the number of cells in the domain), the vast majority of which would have zero amplitude.

16.3.2 Stir Unit: FromFile Implementation

In the from file implementation, the Stir unit sets up a stirring field from data residing on a file. Here we summarize the method for driving turbulence used in Federrath et al. (2010, A&A, 512, A81). Please refer to that paper for further details.

Turbulence decays in about a crossing time, because the kinetic energy carried by the turbulence dissipates on small scales and turns into heat. In order to study the statistics of turbulence (e.g., the PDF, power spectrum, structure functions, etc.) over a significant time period thus requires continuous stirring (also called driving or forcing) with a turbulent acceleration field, which we call \( \vec{f}(x, t) \) in the following.

The stirring field \( \vec{f} \) is often modeled with a spatially static pattern for which the amplitude is adjusted in time. This results in a roughly constant energy input on large scales, but has the disadvantage that the turbulence is not really random, because the large-scale pattern is fixed, which may introduce undesirable systematics. Other studies model \( \vec{f} \) such that it can vary in time and space to achieve a smoothly varying pattern that resembles the flow of kinetic energy from scales larger than the simulation box scale. The most widely used method to achieve this is the Ornstein-Uhlenbeck (OU) process. The OU process is a well-defined stochastic process with a finite autocorrelation timescale. It can be used to excite turbulent motions in 3D, 2D, or 1D simulations as explained in Eswaran & Pope (1988, Computers & Fluids, 16, 257).

The OU process is a stochastic differential equation describing the evolution of \( \hat{\vec{f}} \) in Fourier space (k-space):

\[
d\hat{\vec{f}}(\vec{k}, t) = f_0(\vec{k}) \mathcal{P}^{\perp}(\vec{k}) \, d\hat{\vec{V}}(t) - \hat{\vec{f}}(\vec{k}, t) \frac{dt}{T}.
\]  

The first term on the right hand side is a diffusion term. This term is modeled by a Wiener process \( \hat{\vec{W}}(t) \), which adds a Gaussian random increment to the vector field given in the previous time step \( dt \). Wiener processes are random processes, such that

\[
\hat{\vec{W}}(t) - \hat{\vec{W}}(t - dt) = \mathcal{N}(0, dt),
\]
where \( \mathcal{N}(0, dt) \) denotes the 3D, 2D, or 1D version of a Gaussian distribution with zero mean and standard deviation \( dt \). This is combined with a projection using the projection tensor \( \mathcal{P}_\zeta(\vec{k}) \) in Fourier space. In index notation, the projection operator reads

\[
\mathcal{P}^\zeta_{ij}(\vec{k}) = \zeta \mathcal{P}^\perp_{ij}(\vec{k}) + (1 - \zeta) \mathcal{P}^\parallel_{ij}(\vec{k}) = \zeta \delta_{ij} + (1 - 2\zeta) \frac{k_i k_j}{|\vec{k}|^2},
\]

(16.29)

where \( \delta_{ij} \) is the Kronecker symbol, and \( \mathcal{P}^\perp_{ij} = \delta_{ij} - k_i k_j/k^2 \) and \( \mathcal{P}^\parallel_{ij} = k_i k_j/k^2 \) are the solenoidal (divergence-free) and the compressive (curl-free) projection operators, respectively. The projection operator serves to construct a purely solenoidal stirring field by setting \( \zeta = 1 \). For \( \zeta = 0 \), a purely compressive stirring field is obtained. Any combination of solenoidal and compressive modes can be constructed by choosing \( \zeta \in [0, 1] \).

By changing the parameter \( \zeta \), we can thus set the power of compressive modes with respect to the total power in the driving field. The analytical ratio of compressive power to total power can be derived from equation (16.29) by evaluating the norm of the compressive component of the projection tensor,

\[
\left|(1 - \zeta) \mathcal{P}^\parallel_{ij}\right|^2 = (1 - \zeta)^2,
\]

(16.30)

and by evaluating the norm of the full projection tensor

\[
\left|\mathcal{P}^\zeta_{ij}\right|^2 = 1 - 2\zeta + D\zeta^2.
\]

(16.31)

The result of the last equation depends on the dimensionality \( D = 1, 2, 3 \) of the simulation, because the norm of the Kronecker symbol \( |\delta_{ij}| = 1, 2 \) or 3 in one, two or three dimensions, respectively. The ratio of equations (16.30) and (16.31) gives the relative power in compressive modes, \( F_{\text{long}}/F_{\text{tot}} \), as a function of \( \zeta \):

\[
\frac{F_{\text{long}}}{F_{\text{tot}}} = \frac{(1 - \zeta)^2}{1 - 2\zeta + D\zeta^2}.
\]

(16.32)

Figure 16.3 provides a graphical representation of this ratio for the 1D, 2D, and 3D case. For comparison, we plot numerical values of the forcing ratio obtained in eleven 3D and 2D hydrodynamical simulations by Federrath et al. (2010, A&A, 512, A81), in which we varied the forcing parameter \( \zeta \) from purely compressive stirring \( (\zeta = 0) \) to purely solenoidal stirring \( (\zeta = 1) \) in the range \( \zeta \in [0, 1] \), separated by \( \Delta \zeta = 0.1 \).

Note that a natural mixture of stirring modes is obtained for \( \zeta = 0.5 \), which leads to \( F_{\text{long}}/F_{\text{tot}} = 1/3 \) for 3D turbulence, and \( F_{\text{long}}/F_{\text{tot}} = 1/2 \) for 2D turbulence. A simple way to understand this natural ratio is to consider longitudinal and transverse waves. In 3D, the longitudinal waves occupy one of the three spatial dimensions, while the transverse waves occupy two of the three on average. Thus, the longitudinal (compressive) part has a relative power of 1/3, while the transverse (solenoidal) part has a relative power of 2/3 in 3D. In 2D, the natural ratio is 1/2, because longitudinal and transverse waves are evenly distributed in two dimensions.

The second term on the right-hand side of equation (16.27) is a drift term describing the exponential decay of the autocorrelation of \( \vec{f} \). The usual procedure is to set the autocorrelation timescale equal to the turbulent crossing time, \( T = L_{\text{peak}}/V \), on the scale of energy injection, \( L_{\text{peak}} \). This type of stirring models the kinetic energy input from large-scale turbulent fluctuations breaking up into smaller and smaller structures.

The runtime parameters associated with the StirFromFile unit are described in the 16.3.3 section.

### 16.3.3 Using the StirFromFile Unit

#### 16.3.3.1 Runtime Parameters

Table 16.2 lists the runtime parameters for the StirFromFile unit. This includes a switch for turning the stirring module on/off and a switch to restrict the timestep based on the acceleration field used for stirring (\texttt{st\_computeDt} is switched off by default, because it is normally sufficient to restrict the timestep based on the gas velocity). Finally, \texttt{st\_infilename} is the name of the input file containing the time and mode sequence used for stirring. This file must be prepared in advance with a separate Fortran program located
Figure 16.3: Ratio of compressive to total power of the turbulent stirring field, reprinted from Federrath et al. (2010, A&A, 512, A81) with permission by Astronomy & Astrophysics. The solid lines labelled with 1D, 2D, and 3D show the analytical expectation for this ratio, equation (16.32), as a function of the stirring parameter $\zeta$ for one-, two- and three-dimensional driving, respectively. The diamonds and squares show results of numerical simulations in 3D and 2D with $\zeta = [0, 1]$, separated by $\Delta \zeta = 0.1$. The two limiting cases of purely solenoidal stirring ($\zeta = 1$) and purely compressive stirring ($\zeta = 0$) are indicated as "sol" and "comp", respectively. Note that in any 1D model, all power is in the compressive component, and thus $F_{\text{long}} / F_{\text{tot}} = 1$, independent of $\zeta$.

Table 16.2: Runtime parameters for the stirring module.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useStir</td>
<td>boolean</td>
<td>.true.</td>
<td>switch stirring on/off</td>
</tr>
<tr>
<td>st_computeDt</td>
<td>boolean</td>
<td>.false.</td>
<td>restrict timestep based on stirring</td>
</tr>
<tr>
<td>st_infilename</td>
<td>string</td>
<td>&quot;forcingfile.dat&quot;</td>
<td>file containing the stirring time sequence</td>
</tr>
</tbody>
</table>

in SimulationMain/StirFromFile/forcing_generator/. The reason for this structural splitting is to predetermine what the code is going to do. For instance, by preparing the time sequence of the stirring in advance, one can always reproduce exactly the full evolution of all driving patterns applied during a simulation. It also has the advantage that exactly identical stirring patterns can be applied in completely different codes, because they read the time and mode sequence from the same stirring file (Price & Federrath, 2010, MNRAS, 406, 1659).

The stirring module is compatible with any hydro or MHD solver and any grid implementation (uniform or AMR). Upon inclusion in a FLASH setup or module, the StirFromFile module defines three additional grid scalar fields, accx, accy, and accz, holding the three vector components of the stirring field $\mathbf{f}$.

16.3.3.2 Preparing the Stirring Sequence (st_infilename)

The code requires a time sequence of stirring modes at runtime, which have to be prepared with the standalone Fortran program forcing_generator.F90 in SimulationMain/StirFromFile/forcing_generator/. A Makefile is provided in the same directory. This program prepares the time sequence of Fourier modes, which is then read by FLASH during runtime, to construct the physical acceleration fields used for stirring. It controls the spatial structure and the temporal correlation of the driving, its amplitude, the mode mixture, and the time separation between successive driving patterns. The user has to modify forcing_generator.F90 to construct a requested driving sequence and to tailor it to the desired physi-
### 16.3. STIR UNIT

Table 16.3: Parameters in `forcing_generator.F90` to prepare a stirring sequence.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndim</td>
<td>integer</td>
<td>3</td>
<td>The dimensionality of the simulation (1, 2, or 3)</td>
</tr>
<tr>
<td>xmin, xmax</td>
<td>real</td>
<td>-0.5, 0.5</td>
<td>Domain boundary coordinates in x direction</td>
</tr>
<tr>
<td>ymin, ymax</td>
<td>real</td>
<td>-0.5, 0.5</td>
<td>Domain boundary coordinates in y direction</td>
</tr>
<tr>
<td>zmin, zmax</td>
<td>real</td>
<td>-0.5, 0.5</td>
<td>Domain boundary coordinates in z direction</td>
</tr>
<tr>
<td>st_spectform</td>
<td>integer</td>
<td>1</td>
<td>Spectral shape (0: band, 1: paraboloid)</td>
</tr>
<tr>
<td>st_decay</td>
<td>real</td>
<td>0.5</td>
<td>Autocorrelation time of the OU process, $T = L_{\text{peak}}/V$</td>
</tr>
<tr>
<td>st_energy</td>
<td>real</td>
<td>2e-3</td>
<td>Determines the driving amplitude</td>
</tr>
<tr>
<td>st_stirmin</td>
<td>real</td>
<td>6.283</td>
<td>Minimum wavenumber stirred (e.g., $k_{\text{min}} \lesssim 2\pi/L_{\text{box}}$)</td>
</tr>
<tr>
<td>st_stirmax</td>
<td>real</td>
<td>18.95</td>
<td>Maximum wavenumber stirred (e.g., $k_{\text{max}} \gtrsim 6\pi/L_{\text{box}}$)</td>
</tr>
<tr>
<td>st_solweight</td>
<td>real</td>
<td>1.0</td>
<td>Mode mixture $\zeta = [0, 1]$ in Eq. (16.32). Typical values are $1.0$: solenoidal; $0.0$: compressive; $0.5$: natural mixture.</td>
</tr>
<tr>
<td>st_seed</td>
<td>integer</td>
<td>140281</td>
<td>Random seed for stirring sequence</td>
</tr>
<tr>
<td>end_time</td>
<td>real</td>
<td>5.0</td>
<td>Final time in stirring sequence</td>
</tr>
<tr>
<td>nsteps</td>
<td>integer</td>
<td>100</td>
<td>Number of realizations between $t = 0$ and end_time</td>
</tr>
<tr>
<td>outfilename</td>
<td>string</td>
<td>”forcingfile.dat”</td>
<td>Output name (input file <code>st_infilename</code> for FLASH)</td>
</tr>
</tbody>
</table>

Most of them are straightforward to set (ndim, xmin, xmax, ymin, ymax, ...), but others may require some explanation. For example, `st_spectform` determines the shape of the driving amplitude in Fourier space. Many colleagues drive a band (`st_spectform=0`), i.e., equal power injected between wavenumber modes $k_{\text{min}} = st_{\text{stirmin}}$ and $k_{\text{max}} = st_{\text{stirmax}}$. This produces a sharp transition between stirred modes and modes that are not stirred. Here we set the default to a smooth function, a paraboloid (`st_spectform=1`), such that most power is injected on wavenumber $k_{\text{peak}} = (k_{\text{min}} + k_{\text{max}})/2$ and falls off quadratically towards both wavenumber ends, normalized such that the injected power at $k_{\text{min}}$ and $k_{\text{max}}$ vanishes. This has the advantage of defining a characteristic peak injection scale $k_{\text{peak}}$ and achieves a smooth transition between stirred and non-stirred wavenumbers.

`st_decay` and `st_energy` determine the autocorrelation time of the OU process and the total injected energy, which is simply a measure for the normalization of the acceleration field. These parameters must be adjusted according to the physical setup. For instance, for a given target velocity dispersion $V$ on the injection scale $L_{\text{peak}} = 2\pi/k_{\text{peak}}$, the autocorrelation time should be set equal to the turbulent crossing time, $T = L_{\text{peak}}/V$. In contrast, setting `st_decay` to a very small or a very large number results in white noise driving or in a static driving pattern, respectively.

The parameter `st_solweight` determines whether the acceleration field will be solenoidal (divergence-free) or compressive (curl-free) or any mixture, according to Equation (16.32). Incompressible gases should naturally be driven with a purely solenoidal field ($\zeta = 1$), while compressible turbulence in the interstellar medium may be driven primarily by a mixture of solenoidal and compressive modes. A detailed study of the influence of $\zeta$ is presented in Federrath et al. (2010, A&A, 512, A81).

`st_seed` is the random seed for the OU sequence and determines the pseudo random number sequence for the integrated Box-Muller random number generator.

Finally, `end_time` and `nsteps` determine the final physical time for stirring and the number of driving patterns to be prepared within the time period from $t = 0$ to $t = \text{end_time}$. This sets the number of equally-spaced times at which FLASH is going to read a new stirring pattern from the file. This allows the user to control how frequently a new driving pattern is constructed. A useful time separation of successive driving patterns is about 10% of a crossing time (or autocorrelation time), i.e., setting `nsteps = 10 \times \text{end_time}/\text{st_decay}`. This will sample the smooth changes in the OU driving sequence sufficiently well for

---

1. Note that we typically assume a cubic box with side length $L_{\box} = \text{xmax} - \text{xmin} = \text{ymax} - \text{ymin} = \text{zmax} - \text{zmin}$
16.3.4 Stirring Unit Test

An example setup using the StirFromFile unit is located in SimulationMain/StirFromFile/. The unit test can be invoked by

```
./setup StirFromFile -auto -3d -nxb=16 -nyb=16 -nzb=16 +ug -with-unit=physics/Hydro
```

The FLASH executable must be copied into the run directory together with the standard flash.par for this setup, and together with the default forcing file (to be constructed using the standard parameters; see section 16.3.3.2). During runtime the code writes a file with the time evolution of spatially integrated quantities, amongst others, the rms Mach number and vorticity, which can be used as basic code checks.

16.4 Energy Deposition Unit

The Energy Deposition unit calculates the energy deposited by one or more laser beams incident upon the simulation domain. The function EnergyDeposition is the main control driver of the Energy Deposition unit. The current implementation treats the laser beams in the geometric optics approximation. Beams are made of a number of rays whose paths are traced through the domain based on the local refractive index of each cell. The laser power deposited in a cell is calculated based on the inverse Bremsstrahlung power in the cell and depends on the local electron number density gradient and local electron temperature gradient. Currently there are two schemes implemented into FLASH: 1) a ray tracing algorithm based on a cell-by-cell basis (Kaiser algorithm) and 2) a more refined ray tracing method based on cubic interpolation. Both schemes are discussed in the algorithmic implementation section 16.4.3 below.

16.4.1 Ray Tracing in the Geometric Optics Limit

In the geometric optics approach, the path of a laser wave can be described as the motion of a ray of unit-mass through the potential field

\[ V(r) = \frac{c^2}{2} \eta(r)^2, \]  
(16.33)

where \( c \) is the speed of light in vacuum and \( \eta \) is the index of refraction of the medium, assumed to vary on a much longer spatial scale than the wavelength of the laser wave. Also \( \eta(r) \) is considered constant during the time it takes for the ray to cross the domain (frozen medium). However \( \eta(r) \) is allowed to vary from one time step to the next. For a non-relativistic unmagnetized plasma, the refractive index is given by

\[ \eta(r)^2 = 1 - \frac{\omega_p^2(r)}{\omega^2} = 1 - \frac{n_e(r)}{n_c}, \]  
(16.34)

where \( \omega_p(r) \) is the plasma frequency, \( \omega \) the laser frequency, \( n_e(r) \) is the electron number density at location \( r \) and

\[ n_c = \frac{m_e \pi \omega^2}{e^2} = \frac{m_e \pi c^2}{\lambda^2 e^2}, \]  
(16.35)

is the critical density at which the ray frequency and the plasma frequency are equal (\( m_e \) and \( e \) are the electron mass and charge, \( \lambda \) is the laser wavelength). The ray equation of motion is then

\[ \frac{d^2 r}{dt^2} = \nabla \left( -\frac{c^2}{2} n_e(r) \right). \]  
(16.36)

The ray tracing algorithm employed by the Energy Deposition unit assumes a first order Taylor expansion of the electron number density around a specific location \( r_0 \):

\[ n_e(r) \approx n_e(r_0) + \nabla n_e(r_0) \cdot (r - r_0), \]  
(16.37)
where $\nabla n_e(r_0)$ is the electron number density gradient vector at $r_0$. Inserting equation 16.37 into the ray equation of motion 16.36 gives

$$a_0 = \frac{d^2 \mathbf{r}}{dt^2}(r_0) = -\frac{e^2}{2n_e} \nabla n_e(r_0), \quad (16.38)$$

Integration of the ray equation of motion leads to equations for the ray velocity and position as a function of time

$$v(t) = v_0 - \frac{e^2}{2n_e} \nabla n_e(r_0)t, \quad (16.39)$$

$$r(t) = r_0 + v_0 t - \frac{e^2}{4n_e} \nabla n_e(r_0)t^2, \quad (16.40)$$

where $r_0$ and $v_0$ are the initial ray position and velocity.

### 16.4.2 Laser Power Deposition

The power of an electromagnetic wave is depleted by the inverse-Bremsstrahlung (ib) process. The rate of energy loss is given by the formula

$$\nu_{ib} = \frac{n_e}{n_e} \nu_{ei}, \quad (16.41)$$

where $\nu_{ei}$ is the electron-ion collision rate

$$\nu_{ei} = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_e Z e^4 \ln \Lambda}{(kT_e)^{3/2}}. \quad (16.42)$$

Here $m_e$ is the electron mass, $Z$ is the average ionization number of the plasma, $e$ is the electron charge, $\ln \Lambda$ is the Coulomb logarithm, $k$ is the Boltzmann constant and $T_e$ is the electron temperature in Kelvin. The Coulomb logarithm is the natural logarithm of the Debye number $\Lambda$ (number of particles within the Debye sphere of radius the Debye length) and is taken here as

$$\ln \Lambda = \ln \left[ \frac{3}{2Z e^3} \left( \frac{k^3 T_e^3}{\pi n_e} \right)^{1/2} \right]. \quad (16.43)$$

The inverse-Bremsstrahlung rate depends thus on the electron number density and the electron temperature, both of which are functions of the position, and, since the position changes with time, they are ultimately functions of time

$$\nu_{ib}(t) = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{Z e^4}{n_e k^{3/2}} \frac{n_e[r(t)]^2 \ln \varLambda[r(t)]}{T_e[r(t)]^{3/2}}. \quad (16.44)$$

In order to remove the intermediate function $r(t)$ from the expression, we first assume that the Coulomb logarithm remains constant during the incremental time step

$$\ln \varLambda[r(t)] \approx \ln \varLambda[r_0] = \ln \left[ \frac{3}{2Z e^3} \left( \frac{k^3 T_e^3}{\pi n_e(r_0)} \right)^{1/2} \right]. \quad (16.45)$$

Using first order Taylor expansions on both $n_e$ and $T_e$ similar to equation 16.37

$$n_e[r(t)] = n_e(r_0) + \nabla n_e(r_0) \cdot (r(t) - r_0) \quad (16.46)$$

$$T_e[r(t)] = T_e(r_0) + \nabla T_e(r_0) \cdot (r(t) - r_0) \quad (16.47)$$

and inserting the ray position equation 16.40, we get

$$n_e[r(t)] = n_e(r_0) \left[ 1 + \frac{\nabla n_e(r_0) \cdot v_0}{n_e(r_0)} t - \frac{e^2 \nabla n_e(r_0) \cdot \nabla n_e(r_0)}{4n_e n_e(r_0)} t^2 \right], \quad (16.48)$$

$$T_e[r(t)] = T_e(r_0) \left[ 1 + \frac{\nabla T_e(r_0) \cdot v_0}{T_e(r_0)} t - \frac{e^2 \nabla T_e(r_0) \cdot \nabla n_e(r_0)}{4n_e T_e(r_0)} t^2 \right]. \quad (16.49)$$
The inverse-Bremsstrahlung rate can thus be written as a rational polynomial expression

\[ \nu_{ib}(t) = \nu_{ib}(0) \frac{(1 + Ut + Rt^2)^2}{(1 + Wt + St^2)^{3/2}}, \]  

(16.50)

where \( \nu_{ib,0} \) is the inverse-Bremsstrahlung rate at the initial point (zero time)

\[ \nu_{ib}(0) = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{Ze^4 \ln \Lambda[r_0]}{n_e k^{3/2} T_e(r_0)^{3/2}} \]  

(16.51)

and

\[ U = \frac{\nabla n_e(r_0) \cdot v_0}{n_e(r_0)} \]  

(16.52)

\[ W = \frac{\nabla T_e(r_0) \cdot v_0}{T_e(r_0)} \]  

(16.53)

\[ R = -\frac{c^2 \nabla n_e(r_0) \cdot \nabla n_e(r_0)}{4n_e n_e(r_0)} \]  

(16.54)

\[ S = -\frac{c^2 \nabla T_e(r_0) \cdot \nabla n_e(r_0)}{4n_e T_e(r_0)}. \]  

(16.55)

The ray’s power from time zero to time \( t \) is attenuated according to

\[ P_t = P_0 \exp \left\{ -\int_0^t \nu_{ib}(t') \, dt' \right\}, \]  

(16.56)

where \( P_0 \) and \( P_t \) denote the ray’s power at zero time and at time \( t \). Inserting the expression from 16.50, we solve the integral in the exponential using a second order Gaussian quadrature

\[ \int_0^t \nu_{ib}(t') \, dt' = \nu_{ib}(0) \frac{t}{2} \sum_{i=1}^{2} w_i \frac{(1 + Ut_i + Rt_i^2)^2}{(1 + Wt_i + St_i^2)^{3/2}}, \]  

(16.57)

where \( t_{1,2} = (1 \pm 1/\sqrt{3})t_0/2 \) and both weights are equal to 1. The rate of energy deposition (energy deposited per unit time) during this time is then \( P_0 - P_t \).

16.4.3 Algorithmic Implementation of the Ray Tracing

The current implementation of the laser energy deposition assumes that rays transit the entire domain in a single time step. This is equivalent of saying that the domain’s reaction time (changes of its state variables) is much longer than the time it takes for each ray to either cross the domain or get absorbed by it. For each time step it is first checked, if at least one of the laser beams is still active. If this is the case, then for each active beam a collection of rays is generated possessing the right specifications according to the beam’s direction of firing, its frequency, its pulse shape and its power. The Energy Deposition unit moves all rays across all blocks until every ray has either exited the domain or has been absorbed by the domain. The rays are moved on a block by block basis and for each block the rays are traced through the interior cells. The Energy Deposition unit utilizes the infrastructure from the Grid Particles unit 8.9 to move rays between blocks and processors. The code structure of the Energy Deposition unit during one time step is as follows:

- Loop over all blocks containing rays
  - Calculate needed block data for all cells.
  - Start threading
- Loop over all rays in block
  - Trace each ray through all cells in block
End loop

—- End threading ———

End loop

Reassign rays to blocks/processors and repeat (exit, if no more rays)

The inner loop over all rays in a block is conveniently handled in one subroutine to allow for compact (optional) threading. Currently there are two algorithmic options on how to trace the rays in FLASH: 1) Kaiser algorithm and 2) Cubic Interpolation.

16.4.3.1 Kaiser Algorithm

The Kaiser algorithm (Kaiser 2000) is based on tracing the rays on a cell-by-cell basis. Each cell has its own average center electron number density \( \langle n_e \rangle \) and electron temperature \( \langle T_e \rangle \) value. The electron number density gradient vector \( \langle \nabla n_e \rangle \) as well as the electron temperature gradient vector \( \langle \nabla T_e \rangle \) are assumed to be constant within each cell. Rays will be transported through each cell between cell faces in one step. Electron number densities \( n_{e0} \) and electron temperatures \( T_{e0} \) on the entry cell face are calculated from the cell center values in accordance with equation 16.37 using first order Taylor expansions

\[
\begin{align*}
    n_{e0} &= \langle n_e \rangle + \langle \nabla n_e \rangle \cdot (\mathbf{r}_0 - \langle \mathbf{r} \rangle) \quad \text{(16.58)} \\
    T_{e0} &= \langle T_e \rangle + \langle \nabla T_e \rangle \cdot (\mathbf{r}_0 - \langle \mathbf{r} \rangle), \quad \text{(16.59)}
\end{align*}
\]

where \( \langle \mathbf{r}_0 \rangle \) are the position coordinates of the ray at the cell’s face and \( \langle \mathbf{r} \rangle \) are the coordinates of the cell’s center. The ray’s cell crossing time \( t_{\text{cross}} \) is determined from the quadratic time equation 16.40 by inserting planar equations \( A_x x + A_y y + A_z z + D = \mathbf{A} \cdot \mathbf{r}(t) + D = 0 \) for all cell faces (in cartesian coordinates), leading to

\[
- \frac{c^2 \mathbf{A} \cdot \langle \nabla n_e \rangle}{4n_e} t^2 + \mathbf{A} \cdot \mathbf{v}_0 t + \mathbf{A} \cdot \mathbf{r}_0 + D = 0, \quad \text{(16.60)}
\]

and selecting the shortest time (excluding the zero time corresponding to the point of entry). In FLASH, the cartesian cell faces are coplanar with the xy-, xz- and the yz-plane, simplifying the quadratic time equation. A cell face located at \( x_{\text{cell}} \) and coplanar with the yz-plane has the plane equation \( x = x_{\text{cell}} \) and thus \( A_x = 1, \; A_y, A_z = 0 \) and \( D = -x_{\text{cell}} \). For this cell face, the quadratic time equation becomes

\[
- \frac{c^2 \langle \nabla n_e \rangle_x}{4n_e} t^2 + v_{0x} t + r_{0x} - x_{\text{cell}} = 0 \quad \text{(16.61)}
\]

and similar for the other cartesian components. In order to achieve a stable ray tracing algorithm and uniquely assign rays to cells and blocks while crossing through cell walls, a cell wall thickness is defined, which is shared between two adjacent cells. The code is set up such that rays are never to be found inside this wall. When crossing between two cells occurs (same block or different blocks), the rays positions are always adjusted (nudged) such that they will sit on this wall with finite thickness. The cell wall thickness is defined in the code as being a very tiny fraction (1:1,000,000) of the smallest cell dimension during a simulation.

The ray’s rate of energy deposition between the cell entry face and the cell exit face is calculated using equations 16.50 to 16.57 with \( \nabla n_e(\mathbf{r}_0) \) and \( \nabla T_e(\mathbf{r}_0) \) replaced by their cell average values \( \langle \nabla n_e \rangle \) and \( \langle \nabla T_e \rangle \) and \( n_e(\mathbf{r}_0) \) and \( T_e(\mathbf{r}_0) \) replaced by their corresponding cell entry values \( n_{e0} \) and \( T_{e0} \). The upper integration time limit is \( t_{\text{cross}} \), the ray’s cell crossing time.

The Kaiser algorithm, while conceptually simple, leads to discontinuous change in \( n_e \) at the cell interfaces. To account for this, the ray undergoes refraction according to Snell’s law. Refraction at cell interfaces causes a change in the ray’s velocity normal to the interface surface while preserving the ray velocity component transverse to the interface normal. To derive the change in the ray normal velocity component, imagine the cell interface to have a small infinitesimal thickness \( s \) and the ray moving from left to right through the interface. On the left and right the normal velocity components are \( v_{\perp}(\ell) \) and \( v_{\perp}(r) \), respectively, while the
corresponding $n_e$ are $n_e(\ell)$ and $n_e(r)$. Since we are dealing with an interface of infinitesimal thickness, we can use the first order equation 16.39 to get

$$v_\perp(r) = v_\perp(\ell) - \frac{c^2}{2n_e} \frac{\Delta n_e}{s}$$

where $\Delta n_e = n_e(r) - n_e(\ell)$, and the electron number density gradient is $\nabla n_e = \Delta n_e / s$. But $s/t$ is the average velocity of the ray passing through the interface and since we have constant acceleration we have $s/t = [v_\perp(r) + v_\perp(\ell)]/2$, which, when inserted into the last equation, gives

$$v_\perp(r) - v_\perp(\ell) = -\frac{c^2}{n_e} \frac{\Delta n_e}{[v_\perp(r) + v_\perp(\ell)]}$$

$$v_\perp^2(r) - v_\perp^2(\ell) = -\frac{c^2 \Delta n_e}{n_e}$$

$$\Delta v_\perp^2 = -\frac{c^2 \Delta n_e}{n_e}.$$ 

Clearly there is a limit as to how large $\Delta n_e$ can be for refraction to occur. If $\Delta n_e > v^2_\perp(\ell)n_e/c^2$, we have $v^2_\perp(r) < 0$, which is impossible. The only way out is then for the ray to have $\Delta n_e = 0$, i.e. the ray stays in the same cell and reflects with $v_\perp(r) = -v_\perp(\ell)$.

The basic algorithmic steps for tracing a single ray through all cells in one block (see Figure 16.4) can be summarized as follows: The initial situation has the ray positioned on one of the block’s faces with velocity components such that the ray’s direction of movement is into the block.

- Identify ray entry cell $i_0$, entry position $r_0$, entry velocity $v_0$ and entry power $P_0$.
- Calculate $n_{e0}$ and $T_{e0}$ at the entry point using equations 16.58 and 16.59.
- Solve equation 16.40 for time for each possible cell face. Identify the cell crossing time $t_{cross}$ as minimum of all solutions $> 0$.
- Using $t_{cross}$ and equation 16.40 again, find the cell exit position $r_{exit}$.
- Calculate $\ln A[r_0]$, $\nu_{eb}(0)$, $U$, $W$, $R$, $S$ and evaluate $P_{exit}$ using equations 16.56 and 16.57.
- Using $t_{cross}$ and equation 16.39, find the ray’s exit velocity $v_{exit}$.
- Based on $r_{exit}$ and $v_{exit}$ determine new entry cell $i_{exit}$.
- Calculate the electron number density jump $\Delta n_e$ between both cells $i_0$ and $i_{exit}$, using equation 16.37.
- Check for reflection or refraction using Snell’s law equation 16.63 and update $v_{\text{exit}}$ and possibly new entry cell $i_{\text{exit}}$.
- If cell $i_{\text{exit}}$ is still in block, set all exit variables equal to entry variables and repeat the steps.

### 16.4.3.2 Cubic Interpolation

The cubic interpolation algorithm is an attempt at providing continuous $n_e$ and $T_e$ representations as well as continuous first derivatives $\nabla n_e$ and $\nabla T_e$ throughout the entire domain. This allows for the calculation of a more smoother path for each ray inside each cell when compared to the Kaiser algorithm. In effect, the cell-by-cell choppiness of the Kaiser algorithm can be avoided by taking many small ray steps inside each cell. Also, the troublesome $n_e$ discontinuities at the cell boundaries and the application of Snell’s law will disappear. In order to show the essential features of the cubic interpolation method and to keep the formulas and matrices manageable, we present the method for 2D domain geometries.

Consider a particular 2D cell. We wish to calculate a domain function $F$ as a piecewise cubic polynomial in terms of $[0, 1]$ rescaled variables $x, y$ inside each cell:

$$F(x, y) = \sum_{i,j=0}^{3} a_{ij} x^i y^j.$$  \hspace{1cm} (16.64)

The $[0, 1]$ rescaled variable $x$ is obtained from the domain global variable $X$ and the cell’s lower $X_\ell$ and upper $X_u$ dimension as

$$x = \frac{X - X_\ell}{X_u - X_\ell},$$ \hspace{1cm} (16.65)

with a similar expression for $y$. From the expansion in 16.64, there are 16 expansion coefficients which need to be determined (fitted) to an appropriate set of data for specific points on the cell, which, in order to assure continuous representation of $F$ over the entire domain, have to sit on the cell’s boundaries. The most obvious choice for these points are the cell vertices, each vertex belonging to four cells. The data needed for each vertex are $F, dF/dx, dF/dy$ and $d^2F/dxdy$. This leads to a linearly independent and rotationally invariant set of 16 data values, from which the 16 expansion coefficients can be uniquely determined. It can be shown also that for other dimensions only the inclusion of all mixed simple higher order derivatives leads to a linearly independent and rotationally invariant set of data values. For example, for 3D geometries the data needed at each of the 8 vertices of a cubic cell is $F, dF/dx, dF/dy, dF/dz, d^2F/dxdy, d^2F/dxdz, d^2F/dydz$ and $d^3F/dxdydz$.

Let us now organize the 16 expansion coefficients $a_{ij}$ into a vector $a$, such that each element $a(k)$ of $a$ is associated with the following expansion coefficient:

$$a(k) = a_{ij}; \quad k = 1 + i + 4j.$$ \hspace{1cm} (16.66)

Likewise, we stack the data values of the four vertices $v = 1, 2, 3, 4$ into a vector $b$ as follows:

$$b = \left\{ \begin{array}{l} F_v \\ (dF/dx)_v \\ (dF/dy)_v \\ (d^2F/dxdy)_v \end{array} \right\}; \quad F_v = \left\{ \begin{array}{l} F_1 \\ F_2 \\ F_3 \\ F_4 \end{array} \right\}; \quad (dF/dx)_v = \left\{ \begin{array}{l} (dF/dx)_1 \\ (dF/dx)_2 \\ (dF/dx)_3 \\ (dF/dx)_4 \end{array} \right\}; \quad \text{etc...}$$ \hspace{1cm} (16.67)

Let us associate the four vertex indices with the following four $[0, 1]$ rescaled variables of the cell:

$$v = 1 \quad \rightarrow \quad (x, y) = (0, 0)$$
$$v = 2 \quad \rightarrow \quad (x, y) = (1, 0)$$
$$v = 3 \quad \rightarrow \quad (x, y) = (0, 1)$$
$$v = 4 \quad \rightarrow \quad (x, y) = (1, 1)$$ \hspace{1cm} (16.68)

Substituting these values for $x$ and $y$ into equation 16.64 and its derivatives, we can establish a connection between the expansion coefficient vector and the data vector in the form

$$b = Ba,$$ \hspace{1cm} (16.69)
where the 16 x 16 matrix \( B \) has the following structure

\[
B = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 0 & 2 & 0 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 0 & 2 & 4 & 6 & 0 & 3 & 6 & 9
\end{pmatrix}, \quad (16.70)
\]

containing only positive integers and many zeros. Since we are ultimately interested in the expansion coefficients for the cell, we invert equation

\[
a = B^{-1}b. \quad (16.71)
\]

The inverse \( B^{-1} \) is again an integer matrix (with positive and negative elements) and again contains many zero elements. The matrix \( B^{-1} \) is universal for all 2D cells. Each cell has its specific data vector \( b \) from which its expansion coefficient vector \( a \) can be established via equation (16.71). Rather than storing \( B^{-1} \) in matrix form and performing a direct matrix times vector operation for each cell, \( B^{-1} \) is used in implicit form when converting \( b \) to \( a \) for each cell. This avoids the redundant zero multiplications and is thus much more efficient.

We next state the evaluation of the \( F \) values and its derivatives in FLASH to assemble the data vectors (16.67). We assume \( F \) values given at each cell center. Let us label all cell vertices of a 2D block by index pairs \((i, j)\). A specific cell vertex \((i, j)\) is then surrounded by 4 next neighbor cell vertices \((i \pm 1, j)\) and \((i, j \pm 1)\). Evaluation of the vertex \( F \) values is done by simple linear averaging

\[
F_{ij} = \frac{1}{4} \sum_{k=1}^{4} F_k, \quad (16.72)
\]

where \( F_k \) are the 4 closest cell center \( F \) values. Evaluation of the vertex derivative values is done by symmetric differencing in all coordinate directions

\[
(dF/dx)_{ij} = (F_{i+1,j} - F_{i-1,j})/2h_i \quad (16.73)
\]
\[
(dF/dy)_{ij} = (F_{i,j+1} - F_{i,j-1})/2h_j \quad (16.74)
\]
\[
(d^2F/dxdy)_{ij} = (F_{i+1,j+1} - F_{i-1,j-1} - F_{i+1,j-1} - F_{i-1,j+1})/4h_ih_j. \quad (16.75)
\]

The increments \( h_i \) and \( h_j \) correspond to the size of the cells in \([0, 1]\) rescaled coordinates and are hence both equal to 1. The following scheme shows the individual steps during evaluation of all \( F \) (in our case \( F \) is the electron number density \( n_e \) and the electron temperature \( T_e \)) expansion coefficients for one block:

- All needed cell center \( F \) values for the block are gathered or calculated. These include also portions of the guard cells, as these will be addressed during the vertex value evaluations.
- Establish all vertex data using equations 16.72 and 16.73-16.75 for the entire block.
- Apply equation 16.71 on the entire block vertex data to obtain the entire set of cell expansion coefficients for the block.
For ray tracing applications, note that actual $F$ function evaluations inside each cell at a global domain position $(X, Y)$ require first a coordinate transformation to $[0, 1]$ rescaled coordinates via equation 16.65. Similarly, evaluation of the global coordinate differentials is done using the chain rule. For example, the derivative $dF/dX$ is evaluated as

$$
\frac{dF}{dX} = \frac{dF}{dx} \frac{dx}{dX} = \sum_{i=1}^{3} \sum_{j=0}^{3} a_{ij} x^{i-1} y^{j} / (X_u - X_\ell),
$$

where equations 16.64 and 16.65 have been used.

Having a continuous representation of $n_e$ and $T_e$ as well as their first derivatives inside a block, each ray can now be traced by a step-by-step fashion. Critical to this is the decision of a differential stepping length $\Delta r$ based on the cell’s dimension in which the ray currently is. The size of $\Delta r$ will decide how accurate the ray’s path will be traced. Suppose the ray is inside the block at a position $r_0$ with velocity $v_0$. From equation 16.40 and ignoring any acceleration contributions, the time needed to travel the stepping distance $\Delta r$ is

$$t_{\text{step}} = \Delta r / |v_0|.
$$

Inclusion of the acceleration results in having to solve a quartic time equation for $t_{\text{step}}$ and is too expensive. Even equation 16.78 is expensive, because it involves calculation of $|v_0|$ (square root). Hence we resort to the much simpler form (shown for 2D geometries)

$$t_{\text{step}} = \Delta r / \max(v_{0x}, v_{0y}),
$$

with obvious extension to 3D geometry cases. The major cost of the $t_{\text{step}}$ determination is one division. Once $t_{\text{step}}$ has been determined, we apply equations 16.39 and 16.40 with all terms included to find the new velocity and position components, and we use equations 16.50 to 16.57 with upper integration time limit $t_{\text{step}}$. If the new ray position indicates that it will be in a new cell, we cannot use $t_{\text{step}}$ but instead have to determine the time it takes for the ray to reach the nearest cell wall. This can be achieved by using equation 16.60, as was done for the Kaiser algorithm scheme. The strict path to the cell wall is necessary, because the energy deposition is collected on a cell-by-cell basis. Time steps leading to paths crossing cell boundaries are not allowed per se and must be broken down to include cell wall stops. Summarizing, the following cubic interpolation ray tracing algorithm consists of the following steps, assuming the ray is at a position $r_0$ with velocity $v_0$ inside a cell and all cubic expansion coefficients for $n_e$ and $T_e$ have been determined:

- Determine the stepping time $t_{\text{step}}$ using equation 16.79.
- Use equation 16.40 with $t = t_{\text{step}}$ to determine the new ray position vector $r(t)$.
- If $r(t)$ steps outside the current cell boundaries, discard $t_{\text{step}}$ and solve the quadratic equation 16.60 to determine the time $t_{\text{wall}}$ for the ray to reach the nearest cell wall. Update to new $r(t)$ on the cell wall using equation 16.40 with $t = t_{\text{wall}}$.
- Use either $t_{\text{step}}$ or $t_{\text{wall}}$ to determine the new velocity via equation 16.39 and the new ray power using equations 16.50-16.57.

### 16.4.4 Setting up the Laser Pulse

The laser’s pulse contains information about the laser’s energy profile and can thus be characterized by a function of laser power in terms of time $P(t)$. The curve of $P(t)$ gives the power of the laser pulse at any given simulation time. In FLASH, the form of $P(t)$ is given by a series of points, which are connected via straight lines (see Figure 16.5). Each point denotes a power/time pair $P_i(t_i)$. The average laser power $\mathcal{P}$ of the pulse during a time step from $t \to t + \Delta t$ is then

$$\mathcal{P} = \frac{\int_t^{t+\Delta t} P(t) \, dt}{\Delta t},
$$

where
where the integration is performed piecewise between all the points \( P_i(t_i) \) contained within the relevant time frame. The average laser power will be distributed among all the rays that will be created during that particular time step. Note, that \( t_1 \) is the time the laser first comes into existence. In Figure 16.5 there is therefore no line connecting the origin (zero power) with \( P_1 \). If the user wants a gradual buildup of laser power from zero, an explicit point \( P_1 = 0 \) must be given at time \( t_1 \). The same consideration applies for the other end of the timescale.

16.4.5 Setting up the Laser Beam

The laser’s beam contains all the information about orientation and shape of the laser. The orientation is characterized by specifying lens and target coordinates. The shape of the laser is given by the size and shape of the lens and target cross section areas and the cross-sectional ray power distribution law. Figure 16.6 is helpful in visualizing the position of the vectors for the formulas below. The most important vectors for setting up the rays are the two local elliptical semiaxes unit vectors \( \mathbf{u}_1 \) and \( \mathbf{u}_2 \). These two unit vectors, together with the lens and target positions, are very convenient in calculating the rays lens and target coordinates. Note, that the unit vectors are the same for both the lens and the target, if defined from their corresponding local elliptical centers. In what follows, vectors in capital letters are defined in the global coordinate system and vectors in small letters are defined in the local target coordinate system.

16.4.5.1 The Local Elliptical Semiaxis Unit Vectors

The laser beam originates at the lens, whose center is located at \( \mathbf{L} \), and hits the target, whose center is located at \( \mathbf{T} \). In 3D geometries, the elliptical target area is defined as an ellipse perpendicular to the lens-target line and whose largest semiaxis \( s_1 \) is positioned such that it makes a torsional angle \( \phi_1 \) with the local target z-axis (see Figure 16.6). Let us define the beam vector \( \mathbf{b} = \mathbf{L} - \mathbf{T} \) pointing from the target to the lens and connecting the two respective centers. We then have the defining equations for \( s_1 \)

\[
\begin{align*}
  s_{1x} b_x + s_{1y} b_y + s_{1z} b_z &= 0 \quad (16.81) \\
  s_{1x}^2 + s_{1y}^2 + s_{1z}^2 &= \ell_1 \quad (16.82) \\
  s_{1z} &= \ell_1 \cos \phi_1 \cos(\theta - \pi/2) \quad (16.83)
\end{align*}
\]

where \( \ell_1 \) is the length of \( s_1 \) and \( \theta \) is the angle that the beam vector \( \mathbf{b} \) makes with the local z-axis. The first equation 16.81 says that \( \mathbf{b} \) and \( s_1 \) are orthogonal. The second one 16.82 defines the length of \( s_1 \) and the third one 16.83 defines the local z-axis projection of \( s_1 \). The last equation can be rewritten as

\[
s_{1z} = \ell_1 \frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \cos \phi_1. \quad (16.84)
\]
Forming an expression for \( s_{1y}^2 \) from equation 16.81, an expression for \( s_{1z}^2 \) from equation 16.83 and inserting the results into equation 16.82, we obtain a quadratic equation in \( s_{1x}^2 \), from which we can obtain all three components. We get after some algebra and simplifications

\[
\begin{align*}
\mathbf{s}_{1x} &= \frac{\ell_1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_x b_z}{|\mathbf{b}|} \cos \phi_1 \pm \frac{b_y b_z}{|\mathbf{b}|} \sin \phi_1 \right] \quad (16.85) \\
\mathbf{s}_{1y} &= \frac{\ell_1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_y b_z}{|\mathbf{b}|} \cos \phi_1 \mp \frac{b_x b_z}{|\mathbf{b}|} \sin \phi_1 \right] \quad (16.86) \\
\mathbf{s}_{1z} &= \frac{\ell_1}{|\mathbf{b}|} \sqrt{b_x^2 + b_y^2} \cos \phi_1. \quad (16.87)
\end{align*}
\]

The two possible solutions for \( \mathbf{s}_{1x} \) and \( \mathbf{s}_{1y} \) correspond to the two possible definitions of the rotation angle \( \phi_1 \) in either clockwise or counterclockwise rotation from the \( z \)-axis when looking along the beam vector \( \mathbf{b} \) from the lens to the target. Let us henceforth define \( \phi_1 \) to be the clockwise rotation. Then the lower signs in \( \mathbf{s}_{1x} \) and \( \mathbf{s}_{1y} \) apply and dividing each component by \( \ell_1 \) we obtain for the unit vector components

\[
\begin{align*}
\mathbf{u}_{1x} &= \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_x b_z}{|\mathbf{b}|} \cos \phi_1 - \frac{b_y b_z}{|\mathbf{b}|} \sin \phi_1 \right] \quad (16.88) \\
\mathbf{u}_{1y} &= \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_y b_z}{|\mathbf{b}|} \cos \phi_1 + \frac{b_x b_z}{|\mathbf{b}|} \sin \phi_1 \right] \quad (16.89) \\
\mathbf{u}_{1z} &= \frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \cos \phi_1. \quad (16.90)
\end{align*}
\]

The second elliptical semiaxis \( \mathbf{s}_2 \) is perpendicular to the first and lays in the same elliptical target plane. If we define it to be at a right angle in clockwise direction from \( \mathbf{s}_1 \), the torsional angle it makes with the local \( z \)-axis is \( \phi_2 = \phi_1 + \pi/2 \). The formulas for its unit vector components are the same as for \( \mathbf{s}_1 \) but with \( \phi_1 \)
CHAPTER 16. LOCAL SOURCE TERMS

replaced by \( \phi_2 \). From trigonometric sine and cosine relations we can re-express them in terms of \( \phi_1 \)

\[
\begin{align*}
  u_{2x} &= \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ -b_y b_z \sin \phi_1 - b_y \cos \phi_1 \right] \\
  u_{2y} &= \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ b_y b_z \sin \phi_1 + b_x \cos \phi_1 \right] \\
  u_{2z} &= -\frac{1}{|b|} \sqrt{b_x^2 + b_y^2} \sin \phi_1.
\end{align*}
\]

(16.91)

(16.92)

(16.93)

Note the importance of the \( \sqrt{b_x^2 + b_y^2} \) term. If this term is equal to zero, then the unit vectors become undefined. This corresponds to the laser beam being parallel to the global z-axis (and thus coinciding with the local z-axis). Then both elliptical semiaxes are not defined uniquely through a z-axis torsional angle of zero. In this case the torsional angle must be defined through one of the other coordinate axis. The following coordinate index permutations in the above formulas apply:

\[\phi_1 \text{ defined through } x\text{-axis} = x \to y, y \to z, z \to x\]

(16.94)

\[\phi_1 \text{ defined through } y\text{-axis} = x \to z, y \to x, z \to y.\]

(16.95)

In 2D geometries, the beam’s lens and target areas shrink to a line segment. All \( z \)-components are zero and the torsional angle is equal to \( \pi/2 \). The components of the unit vector can be deduced from the equations 16.88 and 16.89 as

\[ u_x = -\frac{b_y}{|b|}\]

(16.96)

\[ u_y = \frac{b_x}{|b|}. \]

(16.97)

16.4.5.2 Extremum Values for the Elliptical Target Zone

In 3D simulations, the planar elliptical target zone can be placed in any possible way inside the domain by specifying the lens \( L \) and target \( T \) positions, both elliptical target zone semiaxes lengths \( \ell_1 \) and \( \ell_2 \), the first semiaxis torsion angle \( \phi_1 \) and the coordinate axis to which \( \phi_1 \) refers to. We wish to enforce in the code a complete containment of the entire target plane inside the domain boundaries. To check this condition we need the extremum coordinate values of the elliptical boundary curve of the target. The collection of all points \( e \), based on the local target coordinate system, can be given in the following implicit form

\[ e = \ell_1 u_1 \cos(\omega) + \ell_2 u_2 \sin(\omega), \quad 0 \leq \omega \leq 2\pi.\]

(16.98)

Differentiating with respect to \( \omega \) and equating to zero we get the minimax condition on all coordinate components as

\[ \omega_{\text{min/max}} = \arctan \left( \frac{\ell_2 u_2}{\ell_1 u_1} \right), \]

(16.99)

where \( \omega_{\text{min/max}} \) denotes the vector of the minimax angles for each cartesian component. The equation 16.99 has two possible answers for each \( \omega \) component in the range \( 0 \leq \omega \leq 2\pi \), corresponding to the minimum and the maximum value. The \( \omega_{\text{min}} \) and \( \omega_{\text{max}} \) angles differ by \( \pi \) radians for each cartesian component. The corresponding minima and maxima on the elliptical boundary curve are obtained by inserting the \( \omega_{\text{min}} \) and \( \omega_{\text{max}} \) angles into equation 16.98.

In order to simplify things, we note that what we need are the sine and cosine values of \( \omega_{\text{min}} \) and \( \omega_{\text{max}} \). From the definition of the trigonometric functions based on the length of the three sides of a right-angled triangle (\( a = \) opposite side, \( b = \) adjacent side, \( c = \) hypotenuse for an angle \( \omega \)), we have, using \( \sin \omega = a/c, \)
16.4. ENERGY DEPOSITION UNIT

Figure 16.7: Setting up the rays between the beam’s lens and target area using a square grid. Only half of the square elliptical grids are shown for clarity.

\[
\cos \omega = \frac{b}{c}, \tan \omega = \frac{a}{b} \quad \text{and} \quad c^2 = a^2 + b^2
\]

\[
\sin(\arctan(a/b)) &= \sin(\omega) = \frac{a}{\sqrt{a^2 + b^2}} \quad (16.100) \\
\cos(\arctan(a/b)) &= \cos(\omega) = \frac{b}{\sqrt{a^2 + b^2}} \quad (16.101)
\]

When applied to equation 16.98, we obtain

\[
e_{\text{min/max}} = \pm \sqrt{(\ell_1 u_1)^2 + (\ell_2 u_2)^2}. \quad (16.102)
\]

The corresponding minimax equation for the 2D geometries (ellipse → line segment) is

\[
e_{\text{min/max}} = \pm \ell u, \quad (16.103)
\]

with \(\ell\) being half the length of the target line segment and \(u\) the line segment unit vector.

16.4.6 Setting up the Rays

For each Energy Deposition unit call, the code sets up the initial collection of rays, defined to be located on the domain boundary with specific velocity components such that they will hit the target area at precise locations if they would cross an empty domain. Key concepts in setting up the rays initial position, velocity and power are the elliptical local square, radial or statistical grids and the beam cross section power function. Rays will be launched from the lens grid intersection points to the corresponding target grid intersection points. Using a square grid, a uniform beam cross-sectional ray density will be achieved, although this could be relaxed in the future to include the possibility of rectangular grids leading to different cross-sectional ray densities along the two elliptical semiaxes directions. A radial grid places the rays on concentrical ellipses.

16.4.6.1 The Elliptical Lens/Target Local Square Grid

In 3D geometries, both the lens and target areas are defined as two similar ellipses (different size, same shape). Given a specific elliptical shape by defining the lengths of both semiaxes \(\ell_1 \geq \ell_2\), we can set up a square grid inside the ellipse, such that the number of grid intersection points matches closely the number of rays \(N_{\text{rays}}\) specified by the user. The grid is defined by the separation \(\Delta\) between two grid points and the placement of the grid’s origin at the center of the ellipse. Our goal is to find this \(\Delta\) value.
Denote by \( N_{\text{rectangle}} \) the number of grid intersection points for the circumscribing rectangle with sides \( 2\ell_1 \) and \( 2\ell_2 \). We then have

\[
N_{\text{rectangle}} = \frac{\text{Number of ellipse points}}{\text{Area rectangle}} \times \frac{\text{Area rectangle}}{\text{Area ellipse}} = \frac{4N_{\text{rays}}}{\pi},
\]

(16.104)

this relation holding only approximately due to the finite resolution of the grid. Let us denote by \( r = \ell_1/\ell_2 \) the ratio between the largest and smallest semiaxis. If \( n_1 \) is the number of tics along the \( \ell_1 \)-semiaxis starting at the grid origin \((0,0)\), then the number of tics along the \( \ell_2 \)-semiaxis is approximately equal to \( n_1/r \). Looking at the circumscribed rectangle area, the total number of grid intersection points laying on the grid’s axes will be twice those on each semiaxis plus the grid’s origin: \( 2n_1 + 2n_1/r + 1 \). The total number of grid intersection points not on any of the axes is \( 4(\pi n_1) \). Both numbers together should then equal to \( N_{\text{rectangle}} \). Hence, from (16.104), we are lead to a quadratic equation in \( n_1 \)

\[
\frac{4}{r} n_1^2 + \left( 2 + \frac{2}{r} \right) n_1 - \left( \frac{4N_{\text{rays}}}{\pi} - 1 \right) = 0,
\]

(16.105)

whose solution is

\[
n_1 = \frac{1}{4} \left[ -(1 + r) + \sqrt{(1 - r)^2 + \frac{16rN_{\text{rays}}}{\pi}} \right].
\]

(16.106)

Always \( n_1 > 0 \), which can easily be seen from the lowest possible value that \( n_1 \) can attain for \( N_{\text{rays}} = 1 \) and the lowest possible ratio \( r = 1 \) (in this case \( n_1 = 0.06418... \)). Since \( n_1 \) has to be an integer we take the ceiling value of the solution (16.106). If the user specified \( N_{\text{rays}} = 1 \), the search for \( n_1 \) is bypassed and the ray is placed at the elliptical origin.

Having \( n_1 \), the next task is to find the optimum (or close to optimum) grid spacing \( \Delta \). This is done by defining a minimum and maximum grid spacing value

\[
\Delta_{\text{min}} = \frac{\ell_1}{n_1 + 1} \quad \text{and} \quad \Delta_{\text{max}} = \frac{\ell_1}{n_1 - 1}.
\]

(16.107)

(16.108)

A series of \( \Delta_{\text{min}} \leq \Delta_k \leq \Delta_{\text{max}} \) grid spacings is then tested, and for each \( \Delta_k \) the number of grid intersection points \( N_k \) inside the ellipse area is determined. Of all \( N_k \) obtained, a certain number of them will be closest to \( N_{\text{rays}} \). The average over all these \( \Delta_k \) will then be taken as the final \( \Delta \) value. For this \( \Delta \) we compute the final number of grid intersection points, which will then replace the user’s specified \( N_{\text{rays}} \).

Since the target \( \ell_1 \) and \( \ell_2 \) semiaxes are specified by the user, the \( \Delta_T \) for the target square grid is evaluated using the above algorithm. The corresponding lens \( \Delta_L \) value is set using the similarity in size between the lens and the target. Using the user’s specified \( \ell_1 \) for the lens, the preservation of length relations between similar objects leads to:

\[
\frac{\Delta_L}{\ell_{1,L}} = \frac{\Delta_T}{\ell_{1,T}}
\]

(16.109)

and hence

\[
\Delta_L = \frac{\ell_{1,L}}{\ell_{1,T}} \Delta_T.
\]

(16.110)

In 2D geometries the situation is much simpler. Due to the linear shape of the lens and target areas, it is very easy to calculate the \( \Delta_T \) value of the linear target grid such that exactly \( N_{\text{rays}} \) grid points are obtained with the outer two grid points coinciding with the target end points. The corresponding \( \Delta_L \) is evaluated using equation (16.110).
16.4.6.2 The Elliptical Lens/Target Local Radial Grid

In order to set up the radial lens and target grids, we use the implicit definition of the elliptical curve from 16.98. The grid will then be defined as the number of tics on the \( \ell_1, \ell_2 \) pair and the number of tics on the angle \( \omega \) within the range \( 0 \geq \omega \geq 2\pi \). The number of tics on both of these 'axes' will be the same and will be denoted by \( n \). The total number of radial grid points within the lens and target ellipses will be equal to \( 1 + n(n+1) \), leading to the quadratic equation

\[
n^2 + n + 1 = N_{\text{rays}},
\]

whose relevant solution is

\[
n = \frac{1}{2} \left[ -1 + \sqrt{4N_{\text{rays}} - 3} \right].
\]

In order to reduce the error to \( N_{\text{rays}} \), rounding to the nearest integer is applied for \( n \). The individual grid points are calculated from

\[
e_{ij} = \ell_1 i \frac{u_1}{n} \cos \left( \frac{2\pi j}{n} \right) + \ell_2 i \frac{u_2}{n} \sin \left( \frac{2\pi j}{n} \right),
\]

where the index ranges are

\[
j = 0, 1, 2, \ldots, n
\]

\[
i = \min(1, j), \ldots, n.
\]

16.4.6.3 Beam Cross Section Power Function

The beam cross section power function describes the power distribution of the rays inside the beam at launching time. Currently there are two types of power distribution functions implemented in FLASH: 1) uniform (flat) distribution (equal power) and 2) gaussian decay from the center of the beam. The first one is trivial: every ray gets assigned an equal amount of power. The gaussian decay function assigns to each ray a relative weight according to the position inside the elliptical cross section of the beam. Using the local \( s_1 \) and \( s_2 \) target coordinate system located at the center of the beam, the gaussian weighting function for 2D target areas (3D geometries) reads

\[
w = \exp \left[ -\left( \frac{x}{R_x} \right)^2 + \left( \frac{y}{R_y} \right)^2 \right]^{\gamma/2},
\]

where \( x \) and \( y \) denote the local coordinates inside the ellipse along the \( s_1 \) and \( s_2 \) axes, respectively, \( R_x \) and \( R_y \) are user defined decay radii values and \( \gamma \) is the user defined gaussian super exponent. For 1D target areas (2D geometries) the weighting function is

\[
w = \exp \left[ -\left( \frac{x}{R_x} \right)^2 \right]^{\gamma},
\]

In order to determine the actual power assigned to each ray, we use the average laser power from equation 16.80

\[
P_{\text{ray}} = \frac{P}{\sum w_{\text{ray}}}.
\]

The sum of the weights over all rays plays the role of a sort of partition function for the elliptical square grid and can thus be precomputed and become part of the beam properties when setting up the beams.
16.4.6.4 The Rays Initial Position and Velocity

In 3D, after the local lens and target grids have been set up, each ray has associated with it a local lens elliptical coordinate pair \((x_L, y_L)\) and a local target elliptical coordinate pair \((x_T, y_T)\). The corresponding global coordinates can be obtained using the global lens and target center coordinates and the elliptical unit vectors (see Figure 16.7)

\[
\begin{align*}
R_L & = x_L u_1 + y_L u_2 + L \\
R_T & = x_T u_1 + y_T u_2 + T.
\end{align*}
\]

(16.119)  (16.120)

Defining now the ray vector \(R\) pointing from the lens to the target

\[ R = R_T - R_L, \]

(16.121)

we can state the parametric ray line equation

\[ P = R_L + wR. \]

(16.122)

The ray line equation is used to determine where and which domain boundary surface the ray will hit. Consider a domain boundary surface contained within a plane given by the equation

\[ A_x x + A_y y + A_z z + D = 0 \]

(16.123)

The value of the real \(w\) parameter where the ray line meets this plane is obtained by inserting the ray line equation into the plane equation. It is

\[ w = \frac{A \cdot R_L + D}{A \cdot R}. \]

(16.124)

Inserting this \(w\) value into the ray line equation 16.122 we obtain the location \(P_{cross}\) of the crossing point on the plane. From Figure 16.7, for the ray vector to cross the plane, the value of \(w\) must be in the range \(0 \leq w \leq 1\). A value of \(w = 0\) or \(w = 1\) indicates that the plane is crossing the lens or target area, respectively. If \(w\) is in proper range, it must next be checked if \(P_{cross}\) is located within the domain boundary surface. If yes, that \(w\) value is accepted. Note, that several proper \(w\) values can be obtained. A ray crossing near the corner of a rectangular domain gives three proper \(w\) values corresponding to crossing the three rectangular planes defining the corner. Since the rays originate from the lens, the relevant \(w\) is the one with minimum value and the corresponding \(P_{cross}\) will be taken as the rays initial position. The initial ray velocity components are determined from the unit vector of the ray vector

\[ v = \frac{R}{|R|}, \]

(16.125)

where \(v_0\) is the initial magnitude of the velocity (in most simulations this is the speed of light). For 2D geometries all the above equations remain the same, except for the ray global coordinates, for which the terms in \(u_2\) are dropped and for the domain boundary planes, for which the terms involving the \(z\)-component in the defining equation 16.148 do not exist. The following steps summarize the determination of the initial ray positions and velocity components for each ray:

- Form the ray vector \(R\) using 16.121.
- Find the collection \(\{w\}\) of all \(0 \leq w \leq 1\) values using 16.124 for all domain surface planes \((A, D)\).
- Using the ray line equation 16.122, remove all \(w\) values from \(\{w\}\) which lead to plane crossing points \(P_{cross}\) not contained within the domain boundary surface.
- Take the minimum of the remaining \(w\)’s in \(\{w\}\) and calculate the corresponding \(P_{cross}\). This is the ray’s initial position vector.
- Using the ray vector \(R\) again calculate the velocity components using 16.125.
16.4.7 3D Laser Ray Tracing in 2D Cylindrical Symmetry

Performing a pure 2D cylindrical ray tracing has an obvious disadvantage: each ray must also be treated as a 2D cylindrical symmetrical object. Each ray can only hit the R-disk of the cylindrical domain at precisely 90 degrees and no variation in this incident angle is possible. However, if one treats the cylinder as a true 3D object, then it is possible to trace each ray through this 3D cylinder. The advantage of retaining the 2D cylindrical description of the domain is obvious: only 2D storage is needed to describe the properties of the domain. 3D in 2D ray tracing is much more complicated than either the simple pure 3D or pure 2D counterparts. The interplay between the polar and the cartesian coordinates leads to ray tracing equations which can only be solved approximately via the use of elliptical integrals. Rather than approximating the integrals involved, a second approach decomposes the 3D cylinder into several identical cylindrical wedges with each wedge having planar boundaries. The larger the number of wedges the more accurate the 3D in 2D ray tracing will be. Both approaches (approximating the integrals and wedging the 3D cylinder) are entirely equivalent.

16.4.7.1 The Exact 3D in 2D Ray Tracing Solution

We will first state the exact time-radial solution of an object of mass \( m \) moving in a central force environment with no external forces acting on it. The motion of such an object is characterized by constant energy and angular momentum and is hence confined to a 2D plane. There are two ways to describe the position \( \mathbf{r} \), velocity \( \mathbf{v} \) and acceleration \( \mathbf{a} \) vectors of a particle in a 2D plane: using cartesian \( \mathbf{i}, \mathbf{j} \) or polar \( \mathbf{\hat{R}}, \mathbf{\hat{\theta}} \) unit vectors. They are interrelated by

\[
(\mathbf{\hat{R}}, \mathbf{\hat{\theta}}) = (\mathbf{i}, \mathbf{j}) \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}
\]  

Using these unit vectors and the differentiation chain rule we get

\[
\mathbf{r} = x\mathbf{i} + y\mathbf{j} = R\mathbf{\hat{R}}
\]  

\[
\mathbf{v} = \dot{x}\mathbf{i} + \dot{y}\mathbf{j} = \dot{R}\mathbf{\hat{R}} + R\dot{\theta}\mathbf{\hat{\theta}}
\]  

\[
\mathbf{a} = \ddot{x}\mathbf{i} + \ddot{y}\mathbf{j} = (\ddot{R} - R\dot{\theta}^2)\mathbf{\hat{R}} + (R\ddot{\theta} + 2\dot{R}\dot{\theta})\mathbf{\hat{\theta}}
\]
where each dot on the dotted variables stands for $d/dt$. The angular momentum of the object in polar coordinates is

$$L = \mathbf{r} \times m\mathbf{v} = R\hat{\mathbf{R}} \times m(\dot{R}\hat{\mathbf{R}} + R\dot{\theta}\hat{\theta}) = mR^2\dot{\theta}(\hat{\mathbf{R}} \times \hat{\theta})$$

(16.130)

and its magnitude

$$L = mR^2\dot{\theta}$$

(16.131)

Since only a central force component is present, this force depends only on the radial part

$$F(R) = ma_R = m(\ddot{R} - R\ddot{\theta}^2)\hat{\mathbf{R}}.$$ 

(16.132)

We are now ready to state the equation of motion of the object in polar coordinates. Let’s assume the object has initially at time $t_0$ the coordinates $(R_0, \theta_0)$, and let $E$ be its constant energy. $E$ is composed of two parts: kinetic and potential energy. Both parts will change as time passes by. The potential energy $U$ can only change along a change in $R$, because the force has only a radial component. We have

$$E = \frac{1}{2}mv^2 + U(R) = \frac{1}{2}m(\dot{R}^2 + R^2\dot{\theta}^2) + U(R)$$

(16.133)

where in the last step we have eliminated the angular dependence through the use of Eq. (16.131). The last equation for $E$ is a first order differential equation allowing for separation of variables

$$\int_{R_0}^{R_T} \left[ \frac{2}{m}(E - U(R)) - \frac{L^2}{m^2R^2} \right] dR = \int_{t_0}^{t_T} dt$$

(16.134)

The potential energy can be obtained via integration

$$U(R) = -\int_{R_0}^{R} F(R')dR' + U(R_0) = -m\int_{R_0}^{R} a_R(R')dR' + U(R_0).$$

(16.135)

The value of the initial potential energy is arbitrary and is conveniently set to zero. Assuming further that $t_0 = 0$, we obtain

$$t_T = \int_{R_0}^{R_T} \left[ \frac{2E}{m} - \frac{L^2}{m^2R^2} + 2\int_{R_0}^{R} a_R(R')dR' \right]^{-1/2} dR.$$

(16.136)

Given an initial radial $R_0$ and a final target radial $R_T$ value, $t_T$ is the time the object will take in travelling from $R_0$ to $R_T$ under the specified initial conditions at $R_0$ (values of $E$ and $L$). If the object will not reach $R_T$ then $t_T$ is either negative or complex. The time equation (16.136) can be solved analytically only for a small number of cases (like for example if $a_R = 0$). All other cases require numerical approximation to the radial integral. For our case, when the ray moves through one radial 2D cylindrical zone using the Kaiser approximation, the central force acceleration is constant, i.e $a_R = a$. For this case we have

$$t_T = \int_{R_0}^{R_T} \left[ \frac{2E}{m} - \frac{L^2}{m^2R^2} + 2a(R - R_0) \right]^{-1/2} dR,$$

(16.137)

for which it is possible to give an analytical solution in terms of elliptical integrals. However, the analytical solution is far too complicated to be of practical use.
16.4.7.2 The Approximate 3D in 2D Ray Tracing Solution

The main reason for the complicated time equation has its roots in the $\theta$ polar variable describing the rotational symmetry. An approximate treatment can be formulated, in which each circle is treated as a linear polygon with $n$ sides ($n$ not necessarily integer), where the value of $n$ determines the quality of the approximation. The exact 3D radial acceleration cylindrical problem in $R, \theta, z$ coordinates is transformed into an approximate 3D cartesian problem in $x, y, z$ coordinates, where the $z$ coordinate remains unaffected. Each torus cell is hence approximated as a collection of truncated wedge (TW) cells. The trapezoidal sides of these wedges are parallel to the $x, y$ plane and the rectangular sides are perpendicular to this plane. Since all TW cells have exactly the same size and shape when coming from the same torus cell, it suffices to concentrate on just one of them. These representative TW cells will be placed inside the $x, y$ plane, such that the positive $x$ axis divides the trapezoidal sides into two equal areas. The collection of all representative TW cells has the shape of a wedge with opening angle $\Omega$. In order to trace the rays through the TW cells, we need the equation of all the cell boundary planes. The equations of the cell planes perpendicular to the $z$ axis (containing the trapezoidal sides) are simply $z = z_{cell}$. The equations of the cell planes perpendicular to the $x$ axis are similarly $x = x_{cell}$. The remaining cell plane equations corresponding to the non-coplanar cell faces are (see Figure 16.9)

$$y = \pm \tan\left(\frac{\Omega}{2}\right) x$$

$$= \pm mx.$$

The quadratic time equations to be solved for the $x = x_{cell}$ and $z = z_{cell}$ plane equations are of the same form as equation 16.61. For the plane equations in 16.138 we obtain, after inserting the appropriate forms into the general cartesian quadratic time equation 16.60 and considering the zero gradient approximation in $y$ direction

$$-\frac{c^2}{4n_c} [\mp m(\nabla n_c)_x] t^2 + [\mp mv_{0x} + v_{0y}] t + [\mp mr_{0x} + r_{0y}] = 0$$

We now describe the three main tasks for the 3D in 2D ray tracing: 1) proper setup of lenses and corresponding targets, 2) initial placement of the 3D rays on the 2D cylindrical domain and 3) tracing of the rays through the truncated wedges.

16.4.7.3 Extremum Global Radial 3D and 2D Distance Values for 3D Elliptical Lens and Target Zones

In curved domain setups (cylindrical and spherical), it becomes necessary to calculate extremum global radial distance values for a 3D elliptical curve from the domain origin in order to check proper placements of the
lens and target zones. We start by stating the global equation of the elliptical target curve \( E \) (the treatment of the lens ellipse is analogous) in 3D in implicit form (see Figure 16.6):

\[
E = T + s_1 \cos(\omega) + s_2 \sin(\omega), \quad 0 \geq \omega \geq 2\pi.
\] (16.140)

The 2D projections of this elliptical 3D curve in either the \((x,y)\), \((x,z)\) or \((y,z)\) plane are themselves ellipses, however the 2D projections of \( s_1 \) and \( s_2 \) do no longer correspond to the 2D elliptical major and minor semi-axes. They generally are oblique to each other and their scalar product is non-zero. We will therefore develop general formulas taking the possible reduction of 3D to 2D into account. The square of the radial distance for any point on that curve from the global origin is

\[
|E|^2 = |T|^2 + 2s_1 \cdot T \cos(\omega) + 2s_2 \cdot T \sin(\omega) + s_1 \cdot s_1 \cos^2(\omega) + s_2 \cdot s_2 \sin^2(\omega) + 2s_1 \cdot s_2 \cos(\omega) \sin(\omega).
\] (16.141)

Differentiating with respect to \( \omega \) and setting equal to zero leads to the minimax equation

\[
B \cos(\omega) - A \sin(\omega) - C \sin(\omega) \cos(\omega) + D[\cos^2(\omega) - \sin^2(\omega)] = 0,
\] (16.142)

where

\[
A = s_1 \cdot T \quad (16.143)
B = s_2 \cdot T \quad (16.144)
C = s_1 \cdot s_1 - s_2 \cdot s_2 \quad (16.145)
D = s_1 \cdot s_2. \quad (16.146)
\]

The minimax equation 16.142 contains a mixed trigonometric term and can only be solved by eliminating either the sine or the cosine function. This leads in general to a quartic equation

\[
q_4 x^4 + q_3 x^3 + q_2 x^2 + q_1 x + q_0 = 0,
\] (16.147)

whose coefficients are, depending on if \( x \) is either \( \cos(\omega) \) or \( \sin(\omega) \):

<table>
<thead>
<tr>
<th>( x = \cos(\omega) )</th>
<th>( x = \sin(\omega) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_4 )</td>
<td>( C^2 + 4D^2 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( 2AC + 4BD )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( A^2 + B^2 - C^2 - 4D^2 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( -2AC - 2BD )</td>
</tr>
<tr>
<td>( q_0 )</td>
<td>( D^2 - A^2 )</td>
</tr>
</tbody>
</table>

All four \( A,B,C,D \) can have a value of zero. The coefficient of \( x^4 \) plays a special role. Note, that for this coefficient to be zero, both \( C \) and \( D \) must be zero, which corresponds to a circular curve in both 3D and 2D. For a circular target the quartic reduces to a quadratic. For the 3D cases we always have \( D = 0 \). The following table 16.4 shows the different cases that can arise for 3D, together with a description of the geometrical shapes of the target figures involved and the corresponding maximum and minimum radial distances: In this table we have used our convention \( |s_1| \geq |s_2| \) and placement of the elliptical semi-axes in such a way that the angle they make with \( T \) is \( \geq 90^\circ \). The cases that arise for the 2D are summarized in the table 16.5: When solving the quartic or quadratic product equations, we must remember to eliminate the two non-valid extra solutions introduced due to squaring. All real solutions between \(-1\) and \(+1\) are inserted into 16.141 and the maximum and minimum of all four possible values are selected.

### 16.4.7.4 Initial Placement of the 3D Rays on the 2D Cylindrical Domain

There are two kinds of domain surfaces for a cylindrical domain. The first kind corresponds to the flat circular surface at both ends of the cylinder. These surfaces are planar and the determination of the ray
Table 16.4: Different cases arising in 3D, depending on values of \(A, B, C, D\)

| \(D\) | \(C\) | \(A\) | \(B\) | Shape | \(E|_{\text{min}}^2\) | \(E|_{\text{max}}^2\) |
|-----|-----|-----|-----|-----|----------|----------|
| 0   | 0   | 0   | 0   | Circle (radius \(|s_1|\)) | \(T^2 + |s_1|^2\) | \(T^2 + |s_1|^2\) |
| 0   | 0   | 0   | \(\neq\) 0 | Tilted circle | \(|T - s_2|^2\) | \(|T + s_2|^2\) |
| 0   | 0   | \(\neq\) 0 | 0   | Tilted circle | \(|T - s_1|^2\) | \(|T + s_1|^2\) |
| 0   | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Double-tilted circle | Solve quadratic |
| \(\neq\) 0 | 0   | \(\neq\) 0 | 0   | Ellipse | \(|T + s_2|^2\) | \(|T - s_2|^2\) |
| \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Tilted ellipse | Solve quadratic product |
| \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Tilted ellipse | Solve quadratic |

Table 16.5: Different cases arising in 2D, depending on values of \(A, B, C, D, |s_1|, |s_2|\)

| \(D\) | \(C\) | \(A\) | \(B\) | \(|s_1|\) | \(|s_2|\) | Shape | \(E|_{\text{min}}^2\) | \(E|_{\text{max}}^2\) |
|-----|-----|-----|-----|-----|-----|-----|----------|----------|
| 0   | 0   | 0   | 0   | 0   | 0   | Point | \(T^2\) | \(T^2\) |
| \(\neq\) 0 | 0   | \(\neq\) 0 | 0   | \(\neq\) 0 | \(\neq\) 0 | Line (length \(|s_2|\)) | \(|T - s_2|^2\) | \(|T + s_2|^2\) |
| \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Line (length \(|s_1|\)) | \(|T - s_1|^2\) | \(|T + s_1|^2\) |
| 0   | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Circle (radius \(|s_1|\)) | \(|T - |s_1||^2\) | \(|T + |s_1||^2\) |
| \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Ellipse \((s_2||T)\) | Solve quadratic product |
| \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Ellipse \((s_1||T)\) | \(|T - |s_1||^2\) | \(|T + |s_1||^2\) |
| 0   | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | \(\neq\) 0 | Tilted ellipse | Solve quartic |

Intersection position on these surfaces proceeds along the same lines as shown in section 16.4.6.4. The second kind of surface is the cylinder mantle, whose geometrical equation is:

\[
x^2 + y^2 - D = 0
\]

\[
(x \cdot y) - D = 0.
\]  
(16.148)

Inserting the parametric ray line equation 16.122, we obtain a quadratic equation in \(w\):

\[
(R \cdot R)w^2 + (2R \cdot R_L)w + (R_L \cdot R_L - D) = 0,
\]  
(16.149)

where only the \(x\) and \(y\) components of the vectors are taken to form the scalar products. Having found the appropriate \(w\) for each ray and the corresponding crossing point \(P_{\text{cross}}\) on the domain cylindrical surface, we need to translate the ray 3D domain crossing coordinates \(P_x, P_y, P_z\) into the initial ray 2D cylindrical wedge coordinates \(W_R, W_y, W_z\), where \(W_R\) and \(W_z\) correspond to the 2D cylindrical domain and \(W_y\) is the linearized angular coordinate from 16.138. Since the starting angular point is irrelevant (there is no preference of any point on the circle), we set this point equal to zero. Thus:

\[
W_R = \sqrt{P_x^2 + P_y^2}
\]  
(16.150)

\[
W_y = 0
\]  
(16.151)

\[
W_z = P_z
\]  
(16.152)

To determine the initial ray wedge velocities, we define two unit vectors of the 3D ray vector: 1) the unit vector in the \((x,y)\)-plane and 2) the unit vector along the \(z\)-axis:

\[
u_{xy} = \frac{R_{xy}}{|R|}
\]  
(16.153)

\[
u_z = \frac{R_z}{|R|},
\]  
(16.154)
where $\mathbf{R}_{xy}$ is the $(x,y)$-plane projection of the 3D ray vector. Likewise, the crossing point vector is also split into two components:

$$\mathbf{P}_{\text{cross}} = \mathbf{R}_{xy} + \mathbf{P}_z.$$  \hfill (16.155) 

Since the origin of the cylindrical domain is located at $(x,y) = (0,0)$, the vector $\mathbf{P}_{xy}$ is a radial vector in the $(x,y)$-plane. The ray vector component $\mathbf{R}_{xy}$ on the other hand is usually not radially oriented, because the ray’s origin (on the lens) does not necessarily lay on the $(x,y) = (0,0)$ line. Both vectors $\mathbf{P}_{xy}$ and $\mathbf{R}_{xy}$ are useful in order to determine the ray’s initial wedge velocities (see Figure 16.10). Denote the angle between $\mathbf{R}_{xy}$ and $\mathbf{P}_{xy}$ by $\alpha$. Then

$$\cos(\alpha) = \frac{\mathbf{R}_{xy} \cdot \mathbf{P}_{xy}}{|\mathbf{R}_{xy}| |\mathbf{P}_{xy}|}$$  \hfill (16.156) 

and the initial ray wedge velocities are

\begin{align*}
v_R &= v_0 \cdot |\mathbf{u}_{xy}| \cdot \cos(\alpha) \\
v_y &= v_0 \cdot \text{sign}(\mathbf{R}_{xy} \times \mathbf{P}_{xy}) \cdot |\mathbf{u}_{xy}| \cdot \sin(\alpha) \\
v_z &= v_0 \cdot |\mathbf{u}_z|,
\end{align*}  \hfill (16.157 - 16.159) 

where the sine and the cross product vector present in the expression for $v_y$ are calculated as

$$\sin(\alpha) = +\sqrt{1 - \cos^2(\alpha)}.$$  \hfill (16.160) 

and

$$\mathbf{R}_{xy} \times \mathbf{P}_{xy} = \begin{vmatrix} i & j & k \\ R_x & R_y & 0 \\ P_x & P_y & 0 \end{vmatrix} = (R_z P_y - R_y P_z) \mathbf{k}.$$  \hfill (16.161) 

The sign of the cross product refers to the sign of the $\mathbf{k}$ component. Note, that the wedge’s $y$-direction refers to the cylindrical $\theta$-coordinate and the convention is that counterclockwise rotation along $\theta$ is considered positive.

### 16.4.7.5 Tracing the Rays through the Truncated Wedges

When a ray hits the $R$- or $z$-planes of a truncated wedge, the $R$- or $z$-components of the ray’s position and velocity change the same way as for a pure 2D cylindrical simulation. The $y$-components stay the same.
16.4. ENERGY DEPOSITION UNIT

When the ray hits the \( y \)-planes of the truncated wedge, it will cross into a new rotated wedge and the \( y \)-position and the \( R \)- and \( y \)-components of the velocity must change to reflect that rotation. The change in \( y \)-position is simple: just invert its sign. For the velocity component changes we need the cosine and sine of the wedge’s opening angle \( \Omega \). The new components are thus

\[
W_y(\text{new}) = -W_y
\]  
\[
v_R(\text{new}) = v_R \cos(\Omega) \pm v_y \sin(\Omega)
\]  
\[
v_y(\text{new}) = v_y \cos(\Omega) \mp v_R \sin(\Omega),
\]

where the sign options in \( v_R \) and \( v_y \) correspond to the two possible \( y \)-planes defined in 16.138 (see Figure 16.11). The change in direction of both velocity vector components is not a rotational effect on the ray due to the medium through which it travels but rather a geometrical effect due to changing wedges. When crossing a common \( y \)-plane between two wedges, the ray is still on the same position in the \((R,z)\) 2D cylindrical plane. Note, that \( \Omega \) must not necessarily be a divisor of 360°. Since \( \Omega \) is fixed through the entire simulation, \( \cos(\Omega) \) and \( \sin(\Omega) \) are conveniently computed only once during initialization.

16.4.8 Synchronous and Asynchronous Ray Tracing

The default laser implementation uses the Lagrangian particle framework to move rays (modeled as particles) across block and process boundaries. We generally find that particle movement consumes a relatively small portion of total runtime even in large-scale simulations with millions of particles (Dubey et al., 2012). This is not always case in laser-driven FLASH simulations, where realistic simulations can spend the majority of runtime moving rays between blocks. The reason for the discrepancy in performance is because a ray generally travels a much larger distance than a tracer particle in a single time-step. In the tracer particle case we only need one application of the communication subroutines in a single time-step because a particle never moves further than a nearest-neighbor block (see CFL limit). In contrast a ray can move to the other side of the domain in a single time-step which may involve visiting hundreds of intermediate blocks.

There are various implementations of particle exchange in the GridParticles sub-unit. In order to understand the performance bottlenecks, we describe the default implementation used in FLASH applications configured with Paramesh. The particle movement communication consists of an \texttt{MPI	extunderscore Allreduce} to determine the number of particles and messages that each MPI rank will receive, followed by point to point
messages which exchange the actual particles. In addition to the communication in the GridParticles unit, there is also an `MPI_Allreduce` in the laser unit which determines if all the rays have either left the domain or have been absorbed. This is the termination criteria and controls whether another round of particle exchange is needed.

Performance profiles of simulations using the laser indicate that most time is spent in `MPI_Allreduce` calls in a particle movement subroutine. The trace in Figure 16.12 shows that this happens because of load imbalance and not slowness of the underlying collective communication call. The two main colors in this trace are red and purple; red is time spent in a computational kernel named `ed_traceBlockRays2Dcy13D` and purple is time spent in `MPI_Allreduce`. We see that the color map is dominated by purple which indicates most MPI ranks are waiting for the `MPI_Allreduce` to complete. The cascading red pattern shows the ray paths as rays move between blocks assigned to different MPI ranks. It is clear that there are many round of bulk synchronous particle exchange and that some MPI ranks must wait a long time before receiving any rays. The biggest impediment to good performance is the dependency between MPI ranks which happens because ray paths are not known ahead of time. Performance quickly degrades at higher processor counts because increased domain decomposition means rays must cross more process boundaries. One approach to improve performance would be to use the mesh replication feature of FLASH to trivially parallelize over the rays, but this is not ideal because it introduces redundant computations of e.g. hydrodynamics in each mesh. The approach we take is to introduce pipelined parallelism to hide the dependency between MPI ranks; rays are sent to nearest-neighbor blocks on different MPI ranks as early as possible to minimize the time MPI ranks must wait for work. Computation and communication are overlapped, which is different to the current bulk synchronous approach where there are distinct computation and communication phases. The “asynchronous ray trace” is included in a FLASH application by setting up with the shortcut `+asyncLaser`.

We describe the ray exchange and termination communication kernels in the asynchronous scheme below.

The ray exchange communication kernel requires all MPI ranks set up designated send and receive communication channels with only those MPI ranks which own nearest-neighbor blocks. This is a relatively small subset because of the block locality in Paramesh. We post one speculative non-blocking receive for each neighbor and then test all communication channels (at a rate determined by runtime parameter `ed_commRaysBetweenMsgTest`) for new rays. When an MPI rank receives a message we copy the rays and then post a new speculative non-blocking receive. This ensures that a message can be received from any nearest neighbor at any time. We send rays using non-blocking sends from source MPI ranks when send
buffers (of size determined by runtime parameter \texttt{ed\_commChannelSize}) are full. At idle phases, we also send rays before the send buffer is full to prevent deadlock and ensure good global progress. When exchange is complete all MPI ranks send a zero byte message to each neighboring MPI rank to match the remaining speculative receives.

The termination communication kernel must also be non-blocking to allow progress in the ray exchange communication kernel. This rules out using a blocking \texttt{MPI\_Allreduce} as is used in the default synchronous scheme. We have multiple non-blocking implementations of the termination kernel which all have different library and MPI dependencies. The default implementation, which has no special dependencies, uses a master-slave approach to determine completion. It works by sending a count of local inactive rays from slaves to a master which maintains a count of global inactive rays. We send a notification message from the master to the slaves when the global number of inactive rays is equal to the initial count of active rays. Even though we have demonstrated that this implementation \textit{can} work when using 8192 MPI ranks on Intrepid BG/P, there is a danger of exceeding internal MPI limits on the number of messages the master rank can receive. For this reason we have created another implementation which uses a non-blocking \texttt{MPI\_Iallreduce} which is part of MPI-3. We recommend using this implementation because there is less danger of exceeding internal MPI limits. It can be used by setting up a FLASH application with the setup shortcut \texttt{+mpi3}. Given that the MPI-3 standard is still quite new, it is likely that some MPI implementations on today’s production machines do not support non-blocking collectives. Therefore, we also support the non-blocking reduction in the Non Blocking Collectives (NBC) library \url{http://htor.inf.ethz.ch/research/nbcoll/libnbc/}. The necessary reduction name substitutions happen when a FLASH application is setup with \texttt{+libnbc}.

We repeat the experiment in Figure 16.12 with the asynchronous communication scheme and show results in Figure 16.13. The number of ranks on the y-axis and time scale on the x-axis are the same as before. The color map is much more chaotic than before because all MPI ranks are either computing or spinning in various subroutines looking for more work. As before, the red color indicates time spent in a computational kernel named \texttt{ed\_traceBlockRays2Dcyl3D}. The red is now much more clustered on the left side of the figure indicating MPI ranks wait less time for rays than before. We see large segments of green for MPI rank 0 which indicates time spent testing for new messages. This is happening because we made use of the master-slave termination kernel in this test. A performance comparison of the synchronous and asynchronous communication schemes is shown in Figure 16.14. The parameters \texttt{ed\_commChannelSize} and
Figure 16.14: Strong scaling of a laser simulation run on Intrepid BG/P with the synchronous and asynchronous communication scheme.

The `ed_commChannelSize` parameter should be kept relatively small because it determines the amount of space available for rays in send and receive buffers. Large values lead to a very high memory footprint because there is a designated portion of space in the buffers for all communication channels (i.e. all possible pairs of communicating processors) and there can be 10-100s of communication channels. The parameter also determines the number of rays which are buffered before being sent from source processors to destination processors, and so smaller values will improve overall load balance. All messages are non-blocking and so the source processor returns immediately to computation and is free to send rays to different destination processors. However, the source processor will block (actually spin on various MPI/Test statements with no danger of deadlock) if a ray should be sent to the original processor and the original message is not yet delivered. This means it is important for the destination processor to frequently check for new messages not only to start work as soon as possible but also to enable the source processor to return to useful work. The parameter `ed_commRaysBetweenMsgTest` should therefore be kept relatively small so that the destination processor performs less computational work between message checks.

We set the default values of `ed_commRaysBetweenMsgTest=50` and `ed_commChannelSize=100`. Very small values for both parameters improve load balance but introduce new overheads such as the cost of continually testing for new messages and the cost of preparing the laser computation routines for the newly received rays. The preparation work includes sorting the rays in block order and then scanning the ray array to find the start and end index for each block. Sorting improves the memory-access pattern and is a sensible optimization when computational time is large relative to the sort time (as always happens in the original synchronous communication scheme), however, it hurts performance when computational work is very fine-grained. Hence, we do not advise using parameter values much smaller than the default values. In future it would be interesting to see if removing the serial optimization improves the global time to solution. This could happen because more processors are doing useful work despite there being worse memory-access on each processor.

The asynchronous communication scheme conflicts with various capabilities in FLASH. It only works with Paramesh at the current time, although support for UG could probably be added without a huge amount of effort. We initially did not support UG because we did not know the MPI rank IDs of the corner processors.
neighboring blocks, although this is no longer the case. It also conflicts with the Laser I/O.

The asynchronous communication scheme is able to run without the GridParticles dependency by setting up with `useGridParticles=False`. This is generally a good idea because the GridParticles sub-unit has initialization code which allocates large send and receive buffers for bulk synchronous communication. These buffers are not needed in the asynchronous scheme and so removing the dependency significantly reduces the memory footprint. The only down-side is that a slower sorting subroutine will be used in place of GridSortParticles, which is able to achieve a fast sort by using the space in the bulk synchronous communication buffers.

The new asynchronous ray tracing implementation works and has been tested with several laser problems using multiple compilers and MPI implementations. However, since this is a rather new implementation, the asynchronous ray tracing mode should still be considered as being in its experimental stage. Both synchronous and asynchronous ray tracing modes are currently made available, with synchronous mode being the default. The asynchronous mode can be activated by using the appropriate setup parameter and only works when using the Kaiser ray tracing algorithm 16.4.3.1.

### 16.4.9 Usage

To include the use of the Energy Deposition unit, the following should be included into the setup line command:

```
+laser ed_maxPulses=<number> ed_maxPulseSections=<number> ed_maxBeams=<number>
```

The `+laser` is a shortcut that handles all the logistics for properly including the `EnergyDeposition` unit. By default, the ray tracing algorithm used is the Kaiser algorithm 16.4.3.1 in its synchronous communication mode 16.4.8. To activate the cubic interpolation ray tracing scheme 16.4.3.2, replace the `+laser` shortcut by the shortcut `+laserCubicInterpolation`. This automatically includes the relevant routines for cubic interpolation and bypasses the ones responsible for the Kaiser algorithm. To activate the asynchronous ray tracing communication mode, replace the `+laser` shortcut by the shortcut `+asynchLaser`. This enables the asynchronous communication routines and suppresses the synchronous ones. Only the Kaiser ray tracing algorithm is currently implemented in asynchronous communication mode. The other three setup variables fix the dimensions needed for the pulses and beams:

- **ed_maxPulses**: The maximum number of different laser pulses for the simulation.
- **ed_maxPulseSections**: The maximum number of power/time pairs per pulse.
- **ed_maxBeams**: The maximum number laser beams for the simulation.

The Energy Deposition unit reads all the information it needs to construct the laser beams and pulses from runtime parameters specified in the flash.par file. Below is the list of runtime parameters that is needed to properly build the laser. For clarification and figuring out the input to a particular laser simulation, the reader is encouraged to have Figures 16.5, 16.6 and 16.7 at hand.

### 16.4.9.1 Laser Pulses Runtime Parameters

- **ed_numberOfPulses**: Controls the number of different laser pulses that are going to be used.
- **ed_numberOfSections_n**: Indicates the number of power/time pairs that are going to be used to set up the shape of the n-th laser pulse. There must be at least as many of these runtime parameters as there are number of laser pulses defined, i.e. \( n = 1, \ldots, \text{ed_numberOfPulses} \).
- **ed_power_n_i**: Sets the i-th power of the i-th power/time pair of the n-th laser pulse. The ranges of the indices must be at least: \( i = 1, \ldots, \text{ed_numberOfSections_n} \) and \( n = 1, \ldots, \text{ed_numberOfPulses} \).
- **ed_time_n_i**: Sets the i-th time of the i-th power/time pair of the n-th laser pulse. The ranges of the indices must be at least: \( i = 1, \ldots, \text{ed_numberOfSections_n} \) and \( n = 1, \ldots, \text{ed_numberOfPulses} \).
16.4.9.2 Laser Beams Runtime Parameters

- **ed_numberOfBeams**: The number of laser beams that are going to be used.
- **ed_lensX_n**: The x-component of the global lens center position vector \( \mathbf{L} \) (n-th beam).
- **ed_lensY_n**: The y-component of the global lens center position vector \( \mathbf{L} \) (n-th beam).
- **ed_lensZ_n**: The z-component of the global lens center position vector \( \mathbf{L} \) (n-th beam).
- **ed_targetX_n**: The x-component of the global target center position vector \( \mathbf{T} \) (n-th beam).
- **ed_targetY_n**: The y-component of the global target center position vector \( \mathbf{T} \) (n-th beam).
- **ed_targetZ_n**: The z-component of the global target center position vector \( \mathbf{T} \) (n-th beam).
- **ed_targetSemiAxisMajor_n**: The major (largest) semiaxis length \( \ell_1 \) for the elliptical target area (n-th beam).
- **ed_targetSemiAxisMinor_n**: The minor (smallest) semiaxis length \( \ell_2 \) for the elliptical target area (n-th beam).
- **ed_lensSemiAxisMajor_n**: The major (largest) semiaxis length \( \ell_1 \) for the elliptical lens area (n-th beam).
- **ed_semiAxisMajorTorsionAngle_n**: The major elliptical semiaxis torsion angle \( \phi_1 \) along the beam’s lens target center line (n-th beam).
- **ed_semiAxisMajorTorsionAxis_n**: The major elliptical semiaxis torsion axis (’x’, ’y’ or ’z’) from which the torsion angle \( \phi_1 \) is defined (n-th beam).
- **ed_pulseNumber_n**: The pulse shape identification number (n-th beam).
- **ed_wavelength_n**: The wavelength of the laser (n-th beam).
- **ed_initialRaySpeed_n**: The initial speed of the rays when hitting the domain boundary, in units of the speed of light (n-th beam).
- **ed_ignoreBoundaryCondition_n**: Ignore the domain boundary conditions (reflective) when rays enter the domain (n-th beam)?
- **ed_crossSectionFunctionType_n**: Beam cross section power function (flat or gaussian decay) type. For a flat profile use ’uniform’, for gaussian decay use ’gaussian1D’ (two-dimensional beams) or ’gaussian2D’ (three-dimensional beams) (n-th beam).
- **ed_gaussianExponent_n**: The Gaussian super exponent \( \gamma \) for the beam cross section power function in equation 16.116 (n-th beam).
- **ed_gaussianRadiusMajor_n**: The Gaussian radius (e-folding length) \( R_x \) along the major elliptical semiaxis in equation 16.116 (n-th beam).
- **ed_gaussianRadiusMinor_n**: The Gaussian radius (e-folding length) \( R_y \) along the minor elliptical semiaxis in equation 16.116 (n-th beam).
- **ed_gaussianCenterMajor_n**: The Gaussian center location along the major elliptical semiaxis (n-th beam).
- **ed_gaussianCenterMinor_n**: The Gaussian center location along the minor elliptical semiaxis (n-th beam).
- **ed_numberOfRays_n**: Number of rays to be created for the beam. Might be overwritten in 3D geometrical cases (n-th beam).
16.4. ENERGY DEPOSITION UNIT

- **ed_gridType.n**: Specifies the grid type to be used for placing the rays inside the beam. For two-dimensional beams the options are ‘regular1D’ or ‘statistical1D’. For three-dimensional beams the options are ‘square2D’, ‘radial2D’ or ‘statistical2D’. (n-th beam).

- **ed_gridnRadialTics**: For radial grid types, the number of wanted grid positions along each radial spike can be specified (n-th beam).

- **ed_gridnAngularTics**: For radial grid types, the number of wanted angular slices can be specified (n-th beam).

16.4.9.3 Laser General Runtime Parameters

- **ed_maxRayCount**: The maximum number of rays that can be created on one processor.

- **ed_gradOrder**: (Kaiser algorithm only) The order of approximation used for the electron number density \( n_e \) and the electron temperature \( T_e \) in a cell (equation 16.37). A value of 1 leads to linear and a value of 2 to parabolic (quadratic) ray trajectories inside the cell. The first case includes no gradients and only takes the cell’s average value.

- **ed_computeGradNeleX**: (Kaiser algorithm only) If false, the x-components of the gradients \( \langle \nabla n_e \rangle \) are not computed, i.e. set to zero.

- **ed_computeGradNeleY**: (Kaiser algorithm only) If false, the y-components of the gradients \( \langle \nabla n_e \rangle \) are not computed, i.e. set to zero.

- **ed_computeGradNeleZ**: (Kaiser algorithm only) If false, the z-components of the gradients \( \langle \nabla n_e \rangle \) are not computed, i.e. set to zero.

- **ed_enforcePositiveNele**: (Kaiser algorithm only) If true, the x-, y- and z-components of the gradients \( \langle \nabla n_e \rangle \) will be rescaled such that they always deliver a positive (greater or equal zero) value for the number of electrons in a cell.

- **ed_enforcePositiveTele**: (Kaiser algorithm only) If true, the x-, y- and z-components of the gradients \( \langle \nabla T_e \rangle \) will be rescaled such that they always deliver a positive (greater or equal zero) value for the electron temperature in a cell.

- **ed_printMain**: If true, it prints general information regarding the laser setup to a file with name \(<\text{basename}>\text{LaserMainDataPrint.txt}\), where \(<\text{basename}>\) is the base name of the simulation.

- **ed_printPulses**: If true, it prints detailed information about the laser pulses to a file with name \(<\text{basename}>\text{LaserPulsesPrint.txt}\), where \(<\text{basename}>\) is the base name of the simulation.

- **ed_printBeams**: If true, it prints detailed information about the laser beams to a file with name \(<\text{basename}>\text{LaserBeamsPrint.txt}\), where \(<\text{basename}>\) is the base name of the simulation.

- **ed_printRays**: If true, it prints detailed information about the all rays initially generated on each processor to a file(s) with name(s) \(<\text{basename}>\text{LaserRaysPrint}\langle\text{PID}\rangle\text{.txt}\), where \(<\text{basename}>\) is the base name of the simulation and PID is the processor rank number.

- **ed_laser3Din2D**: If true, 3D rays will be traced on a 2D cylindrical grid.

- **ed_laser3Din2DwedgeAngle**: The wedge opening angle \( \Omega \) (in degrees) for a 3D in 2D laser ray trace simulation.

- **ed_rayZeroPower**: Below this value (erg/s), the ray is considered to have zero power.

- **ed_rayDeterminism**: If true, the Grid Unit will be forced to use the Sieve Algorithm to move the ray particle data. Forcing this algorithm will result in a slower movement of data, but will fix the order the processors pass data and eliminate round off differences in consecutive runs.
CHAPTER 16. LOCAL SOURCE TERMS

- **ed\_cellRayDifferentialFactor**: (Cubic interpolation only) This factor times the smallest dimension of each cell is taken as the differential ray stepping length $\Delta r$, to be used in calculating the stepping time in equation 16.79.

- **ed\_cellWallThicknessFactor**: Controls the (imaginary) thickness of the cell walls to ensure computational stability of the laser code. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to $10^{-6}$ and should only very rarely be changed.

- **ed\_saveOutOfDomainRays**: If true, the rays info will be stored into a separate saved ray array for those rays that exit the domain.

- **useEnergyDeposition**: If false, the energy deposition is not activated, even if the code was compiled to do so. Bypasses the need to rebuild the code.

- **threadRayTrace**: If true, the innermost ray loop, tracing all rays through a block, is threaded. This runtime parameter can only be set during setup of the code.

16.4.9.4 LaserIO Runtime Parameters and Usage

Often times it is useful to be able to visualize the paths of the individual rays in a simulation. This can be very helpful in ensuring that the laser beams are set up in the intended manner. The LaserIO directory in the source tree located at:

```
source/physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser/LaserIO
```

By default, this unit is requested by the Laser package (for example, when the +laser setup shortcut is specified).

The LaserIO package gives users the ability to write the trajectory of a certain number of rays to plot files when normal FLASH plot file writes occur. This feature is only compatible with parallel HDF5 IO implementations (for example, with +parallelIO or +hdf5typeio). Only three runtime parameter options need to be set to use LaserIO. These are:

- **ed\_useLaserIO**: Set to .true. to activate LaserIO

- **ed\_laserIOMaxNumberOfRays**: Sets the approximate maximum number of rays to write to the plot file. This number can be less than the total number of rays in the simulation. When greater than the maximum number of rays, every ray trajectory is written.

- **ed\_laserIOMaxNumberOfPositions**: Sets the size of the LaserIO buffer. A good estimate of this value is $5 \times \text{NXB} \times \text{NYB} \times \text{NZB} \times \text{ed\_laserIOMaxNumberOfRays}$.

The ed\_laserIOMaxNumberOfRays parameter sets the approximate maximum number of rays to write out. The exact number of rays that will be written is equal to:

$$\max\left[\text{int}\left(\frac{N_{\text{rays}}}{\text{ed\_laserIOMaxNumberOfRays}}\right), 1\right],$$

where $N_{\text{rays}}$ is the sum of the number of rays to launch for each beam. The LaserIO ray information is written to the normal HDF5 plot files in the RayData dataset. The extract\_rays.py script that is distributed with FLASH in the tools/scripts directory can be used to extract this data. This is a python script which requires the NumPy and PyTables python packages to operate. The script takes as arguments a list of plot files. For each plot file, the script will generate a corresponding VTK file which can be loaded into VisIt. The RayPower.Watts Pseudocolor in VisIt can be plotted to show the ray trajectories. The rays are colored based on their power (in Watts). See the LaserSlab simulation Section 25.7.5 for an example which uses LaserIO.
16.4. ENERGY DEPOSITION UNIT

Figure 16.15: One ray moving through an ellipsoidal tube with a quadratic electron number density profile and reaching a focal point $F$.

16.4.10 Unit Tests

The unit tests set up a series of beams that launch rays onto a domain with a particular electron number density and electron temperature distribution in such a way that an analytical solution is possible for the ray paths and their power deposited inside the domain. The tests are 3D cartesian extensions to the 2D analytical test solution for the quadratic trough as presented in the paper by Kaiser (2000). Figure 16.15 shows the basic structure. 2D cartesian versions of the tests is also available.

16.4.10.1 Analytic Path Solution for the Ellipsoidal Quadratic Potential Tube

Let us define a quadratic ellipsoidal electron number density profile in a 3D space along the xz-plane and having no variation along the y-axis:

$$ n_e(r) = n_w + A(x - x_w)^2 + B(z - z_w)^2, $$

Here $x_w$ and $z_w$ are the center coordinates of the ellipsoidal tube cross section, $n_w$ is the value of the electron number density at the center of the ellipse and $A, B$ are the scaling factors for the individual components. The values of $A$ and $B$ are determined by the boundary conditions of $n_e(r)$. For example, if one wants $n_e(r)$ to be equal to a critical value $n_c$ at specific coordinates $x_c$ and (separately) $z_c$, then we have:

$$ A = \frac{(n_c - n_w)/(x_c - x_w)^2}{(n_c - n_w)/z_c - z_w)^2}, $$

$$ B = \frac{(n_c - n_w)/(z_c - z_w)^2}{(n_c - n_w)/x_c - x_w)^2}. $$

Using the ray equation of motion

$$ \frac{d^2 \mathbf{r}}{dt^2} = \nabla \left( -\frac{c^2 n_e(r)}{2n_e} \right), $$

we can solve for the position vector $\mathbf{r}$ under several circumstances (boundary conditions). The first thing to note is that $n_e(r)$ does not contain mixed variable terms, hence the differential ray equation is separable into the individual components. Let us take the x-component. We have

$$ \frac{d^2 x}{dt^2} = -\frac{c^2}{2n_e \frac{dx}{dx}} A(x - x_w)^2 $$

$$ = -k(x - x_w), $$
where \( k = c^2 A/n_c \). This is the differential equation for a simple harmonic oscillator around the point \( x_w \). The boundary conditions are such that at time \( t = 0 \) the ray will be at an initial position \( x_0 \) and will posses a certain velocity component \( v_x \)

\[
\begin{align*}
x &= x_0 \text{ (at } t = 0) \\
\frac{dx}{dt} &= v_x \text{ (at } t = 0).
\end{align*}
\]

(16.170) (16.171)

To solve the simple harmonic oscillator equation, we make the ansatz

\[
x(t) = x_w + (x_0 - x_w) \cos(\beta t) + b \sin(\beta t)
\]

(16.172)

and find the expressions for \( b \) and \( \beta \) satisfying the boundary conditions. We have

\[
\begin{align*}
\frac{dx}{dt} &= -\beta(x_0 - x_w) \sin(\beta t) + b\beta \cos(\beta t) \\
\frac{d^2x}{dt^2} &= -\beta^2(x_0 - x_w) \cos(\beta t) - b\beta^2 \sin(\beta t) \\
&= -\beta^2(x - x_w)
\end{align*}
\]

(16.173) (16.174)

from which we deduce that

\[
\begin{align*}
\beta &= \sqrt{k} \\
b &= \frac{v_x}{\beta}.
\end{align*}
\]

(16.175) (16.176)

The solution to the complete ray equation can thus be stated as

\[
\begin{align*}
x(t) &= x_w + (x_0 - x_w) \cos(\sqrt{k}t) + \frac{v_x}{\sqrt{k}} \sin(\sqrt{k}t) \\
y(t) &= y_0 + v_y t \\
z(t) &= z_w + (z_0 - z_w) \cos(\sqrt{k'}t) + \frac{v_z}{\sqrt{k'}} \sin(\sqrt{k'}t),
\end{align*}
\]

(16.177) (16.178) (16.179)

where \( r_0 = (x_0, y_0, z_0) \) is the ray’s initial position, \( v_0 = (v_x, v_y, v_z) \) the initial velocity and

\[
\begin{align*}
k &= \frac{c^2 A}{n_c} \\
k' &= \frac{c^2 B}{n_c}.
\end{align*}
\]

(16.180) (16.181)

Let us now define a focal point \( F \), where all rays launched will meet. The obvious choice is a focal point along the tube’s center line \( (x_w, z_w) \). Without specifying the y-coordinate of this point yet, we ask the question: at what times \( t \) and \( t' \) will \( x(t) = x_w \) and \( z(t') = z_w \) for the first time after launch? From the ray equation solutions we get

\[
\begin{align*}
t &= \frac{\arctan[-\sqrt{k}(x_0 - x_w)/v_x]}{\sqrt{k}} \\
t' &= \frac{\arctan[-\sqrt{k'}(z_0 - z_w)/v_z]}{\sqrt{k'}}
\end{align*}
\]

(16.182) (16.183)

where the angular region of interest for the tangent is \( 0 \leq \theta \leq \pi \). It is interesting that there is both a lower and an upper limit to these times. While the lower limit of \( t \to 0 \) makes sense (if we shoot the ray with infinite velocity towards the tube’s center) the upper limit of \( t \to \pi/\sqrt{k} \) is less intuitive; there is no escape from the tube no matter what initial outward velocity we give the ray due to increasing pulling back force towards the center as the ray moves away from it. If the velocity component is zero we have \( t = \pi/(2\sqrt{k}) \).
In general, both times $t$ and $t'$ will be different. What this means is that the ray will not hit the focal point. For that to occur, both times must be equal, hence we must have

$$\frac{\arctan[-\sqrt{k}(x_0 - x_w)/v_y]}{\sqrt{k}} = \frac{\arctan[-\sqrt{k}(z_0 - z_w)/v_z]}{\sqrt{k'}}. \tag{16.184}$$

Given $A$ and $B$ values (and thus $k$ and $k'$ values via equations 16.180 and 16.181) defining the tube’s electron number density layout, this last relation imposes a condition among the variables $x_0, z_0, v_x, v_z$. Only if this condition is fulfilled will a focal point at $(x_w, y_F, z_w)$ be reached. The $y$-coordinate $y_F$ of the focal point is then obtained from the $y$-solution 16.178 of the ray equation by inserting the value of the time $t$ obtained through either side of 16.184. For the unit test we must identify $y_F$ to be sitting on the domain boundary. Hence when launching a series of ray’s for the unit test, all $y_F$ must be the same. A first scheme I thus emerges for setting up a very general unit test:

**SCHEME I**

- Fix the center $(x_w, z_w)$ of the ellipsoidal tube cross section, set up the x- and z-dimensions of the domain, and assign a value to the electron number density $n_w$ at the center.
- Set up the ellipsoidal tube parameters $A$ and $B$, as shown in 16.166 and 16.167, and choose a constant $v_y$ velocity component.
- For one trial ray, given the initial position and velocity x-components $x_0$ and $v_x$, determine the time $t$ to reach the focal point $F$ using equation 16.182.
- Using $t$ and $v_y$, calculate $y_F$ from 16.178, and set up the y-dimensions of the domain.
- Set up the initial x-component positions $x_0$ for the whole set of rays that are going to be launched.
- Using $t$, determine the corresponding x-component velocities $v_x$ for all rays from equation 16.182.
- For each ray, choose a $z_0$ location and determine the needed velocity component $v_z$ using the relation 16.184.
- Launch all rays and collect all the x- and z-components of the domain exit points $(x_{exit}, z_{exit})$.
- For each ray, check how close $(x_{exit}, z_{exit})$ is to the expected $(x_w, z_w)$.

A second simplified scheme II (and the one that is currently implemented in FLASH) has all rays launched vertically onto the xz-face of the domain. This means that $v_x = v_z = 0$. By looking at the time condition 16.184, we see that the only possibility for a focal point in this case is if $k = k'$, and the time associated with it is equal to $t = \pi/(2\sqrt{k})$. This means that we must have $A = B$ (a circular tube). Scheme II is set up as follows:

**SCHEME II**

- Fix the center $(x_w, z_w)$ of the circular tube cross section, set up the x- and z-dimensions of the domain, and assign a value to the electron number density $n_w$ at the center.
- Set up the circular tube parameter $A$, as shown in 16.166, and choose a constant $v_y$ velocity component.
- Calculate the time $t = (\pi/2c)\sqrt{n_e/A}$ to reach the focal point $F$.
- Using $t$ and $v_y$, calculate $y_F$ from 16.178, and set up the y-dimensions of the domain.
- Set up the initial positions $(x_0, z_0)$ for the whole set of rays that are going to be launched.
- Launch all rays and collect all the x- and z-components of the domain exit points $(x_{exit}, z_{exit})$.
- For each ray, check how close $(x_{exit}, z_{exit})$ is to the expected $(x_w, z_w)$.

Note, that for scheme II there is no restriction on the initial ray positions $(x_0, z_0)$ like in scheme I. Rays launched vertically from anywhere onto the xz-plane with the same $v_y$ velocity will reach the same focal point $F$. Their paths might be of different lengths, but the time it takes for them to reach $F$ is always the same.
16.4.10.2 Analytic Power Deposition Solution for the Ellipsoidal Quadratic Potential Tube

According to equation 16.56, the power remaining after the ray exits the ellipsoidal tube is

\[ P_{\text{exit}} = P_0 \exp \left\{ - \int_0^{t_{\text{cross}}} \nu_{ib}[r(t)] \, dt \right\}, \tag{16.185} \]

where \( P_0 \) and \( P_{\text{exit}} \) denote the ray’s power at the tube’s entry and exit point and \( t_{\text{cross}} \) is the time it takes for the ray to reach the focal point \( F \). From 16.44, the inverse-Bremsstrahlung rate is

\[ \nu_{ib}[r(t)] = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{e^4}{n_e k^{3/2}} \frac{Z[r(t)] n_e[r(t)]^2 \ln \Lambda[r(t)]}{T_e[r(t)]^{3/2}}. \tag{16.186} \]

Note the dependence of \( Z \) on position, since we are dealing with the entire domain and not with individual cells. In order to obtain an analytical integrand that is easy to integrate, the first assumption that we make is that both \( Z \) and \( \Lambda \) are independent of position. For convenience we chose:

\[ Z[r(t)] = 1, \tag{16.187} \]
\[ \ln \Lambda[r(t)] = 1. \tag{16.188} \]

To change the quadratic dependence on the electron number density to a linear dependency, we set the electron temperature as

\[ T_e[r(t)] = T_w \left[ \frac{n_e[r(t)]}{n_w} \right]^{2/3} \tag{16.189} \]

with a specific electron temperature \( T_w \) at the elliptical origin. Using these choices we obtain

\[ \nu_{ib}[r(t)] = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{e^4}{n_e k^{3/2} T_w^{3/2}} n_w[r(t)] \]
\[ = \nu_{ib,w} \frac{n_e[r(t)]}{n_w}, \tag{16.190} \]

where

\[ \nu_{ib,w} = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_w^2 e^4}{n_e k^{3/2} T_w^{3/2}} \tag{16.191} \]

is the inverse-Bremsstrahlung rate at the origin. The integral in the exponential becomes

\[ \int_0^{t_{\text{cross}}} \nu_{ib}[r(t)] \, dt = \frac{\nu_{ib,w}}{n_w} \int_0^{t_{\text{cross}}} n_e[r(t)] \, dt \]
\[ = \frac{\nu_{ib,w}}{n_w} \int_0^{t_{\text{cross}}} n_w + A(x[t] - x_w)^2 + B(z[t] - z_w)^2 \, dt. \tag{16.192} \]

Inserting the analytical path solutions 16.177 and 16.179, we can perform the integration. Let us concentrate on the x-component part only. We are lead to an integral of the form

\[ \int_0^{t_{\text{cross}}} (x[t] - x_w)^2 \, dt = \int_0^{t_{\text{cross}}} \left[ \left( x_0 - x_w \right) \cos(\sqrt{k} t) + \frac{v_x}{\sqrt{k}} \sin(\sqrt{k} t) \right]^2 \, dt \]
\[ = \frac{v_x (x_0 - x_w)}{2k} + \frac{v_x^2 + k(x_0 - x_w)^2}{2k} t_{\text{cross}}, \tag{16.193} \]

where \( t_{\text{cross}} \) is given by the expression in 16.182. For this integration, known sine and cosine integration rules have been used as well as the fact that \( \sin(\arctan a) = a/\sqrt{1 + a^2} \) and \( \cos(\arctan a) = 1/\sqrt{1 + a^2} \). The
16.4. ENERGY DEPOSITION UNIT

The current Energy Deposition unit test coded into FLASH is based on the analytic solutions for scheme II. The circular quadratic potential tube is defined through the following parameters and constants:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Constants in FLASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_c, z_c )</td>
<td>( m_e = 9.10938215 \times 10^{-28} \text{ g} )</td>
</tr>
<tr>
<td>( x_w, z_w )</td>
<td>( e = 4.80320427 \times 10^{-10} \text{ esu} )</td>
</tr>
<tr>
<td>( R )</td>
<td>( k = 1.3806504 \times 10^{-16} \text{ erg/K} )</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( c = 2.99792458 \times 10^{10} \text{ cm/sec} )</td>
</tr>
<tr>
<td>( v_y )</td>
<td>( c )</td>
</tr>
<tr>
<td>( T_w )</td>
<td>( 10 \text{ keV} )</td>
</tr>
<tr>
<td>( P_0 )</td>
<td>( 1 \text{ erg/sec} )</td>
</tr>
</tbody>
</table>

These values lead to the following additional variable values (in parenthesis are the equations used), completing the picture:

\[
\begin{align*}
n_e &= 1.1148542 \ldots \times 10^{21} \text{ cm}^{-3} \quad (16.35) \\
n_w &= \frac{n_e}{2} \\
t_{\text{cross}} &= \frac{5\pi}{\sqrt{2}} \text{ sec} \quad (16.182 \text{ with } v_x = 0) \\
y_F &= \frac{5\pi}{\sqrt{2}} \text{ cm} \quad (16.178) \\
\nu_{ib,w} &= 8.1002987 \ldots \times 10^8 \text{ sec}^{-1} \quad (16.191) \\
P_{\text{exit}} &= 0.7017811 \ldots \text{ erg/sec} \quad (16.195 \text{ and } 16.185)
\end{align*}
\]

Evaluation of each cell’s average electron number density is done by integrating equation 16.165 over the entire cell xz-area.

\[
\langle n_e \rangle = \frac{\int_{z_1}^{z_2} \int_{x_1}^{x_2} n_e(x,y) \, dx \, dz}{(x_2 - x_1)(z_2 - z_1)} = n_w + A(\tau - x_w)^2 + A(\tau - z_w)^2 + \frac{A}{3}(\Delta x^2 + \Delta z^2), \quad (16.197)
\]
where $x_1, z_1$ are the lower and $x_2, z_2$ the upper corresponding cell edge coordinates and

$$\tau = \frac{x_1 + x_2}{2}$$  \hspace{1cm} (16.198)

$$\Delta x = \frac{x_1 - x_2}{2}$$  \hspace{1cm} (16.199)

with corresponding equations for the z-part. The average cell’s electron temperature is evaluated directly from the average electron number density using 16.189

$$\langle T_e \rangle = T_w \left[ \frac{\langle n_e \rangle}{n_w} \right]^{2/3}.$$  \hspace{1cm} (16.200)

The gradients are obtained from the cell average values using the symmetrized derivative formula (shown here for the $i$-th cell x-component of the electron number density gradient)

$$\frac{\partial}{\partial x} n_e(i) = \frac{\langle n_e \rangle_{i+1} - \langle n_e \rangle_{i-1}}{d_{i-1,i+1}},$$  \hspace{1cm} (16.201)

where $d_{i-1,i+1}$ is twice the distance between neighboring cells on the x-axis.

**TEST I**

For test I, a series of 8 rays (from 8 beams with 1 ray each) with radial distances of 3 cm from $(x_w, z_w) = (5\,cm, 5\,cm)$ are launched with the following initial $(x_0, z_0)$ coordinates (all in cm):

<table>
<thead>
<tr>
<th>Ray</th>
<th>$x_0$</th>
<th>$z_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5 + $3/\sqrt{2}$</td>
<td>5 + $3/\sqrt{2}$</td>
</tr>
<tr>
<td>6</td>
<td>5 + $3/\sqrt{2}$</td>
<td>5 - $3/\sqrt{2}$</td>
</tr>
<tr>
<td>7</td>
<td>5 - $3/\sqrt{2}$</td>
<td>5 + $3/\sqrt{2}$</td>
</tr>
<tr>
<td>8</td>
<td>5 - $3/\sqrt{2}$</td>
<td>5 - $3/\sqrt{2}$</td>
</tr>
</tbody>
</table>

Results are shown only for the ray 1 and ray 5, the other ones are related to these two by symmetry. Also $z_{exit}$ is not given, because for ray 1 we have $z_{exit} = 5\,cm$ and for ray 5 we have $z_{exit} = x_{exit}$. The percentage errors were calculated as $|x_{exit} - x_w|/x_w$ and likewise for the power deposition.

<table>
<thead>
<tr>
<th>Refinement Level</th>
<th>$x_{exit}$</th>
<th>% error</th>
<th>$P_{exit}$</th>
<th>% error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.04894</td>
<td>0.98</td>
<td>0.69487</td>
<td>0.98</td>
</tr>
<tr>
<td>2</td>
<td>5.16343</td>
<td>3.27</td>
<td>0.69864</td>
<td>0.45</td>
</tr>
<tr>
<td>3</td>
<td>5.02737</td>
<td>0.55</td>
<td>0.70107</td>
<td>0.10</td>
</tr>
<tr>
<td>4</td>
<td>4.98971</td>
<td>0.21</td>
<td>0.70185</td>
<td>0.01</td>
</tr>
<tr>
<td>5</td>
<td>5.00004</td>
<td>0.00</td>
<td>0.70176</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>5.00234</td>
<td>0.05</td>
<td>0.70174</td>
<td>0.00</td>
</tr>
<tr>
<td>Ray 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.35171</td>
<td>7.03</td>
<td>0.69283</td>
<td>1.23</td>
</tr>
<tr>
<td>2</td>
<td>4.77272</td>
<td>4.55</td>
<td>0.70200</td>
<td>0.03</td>
</tr>
<tr>
<td>3</td>
<td>4.96221</td>
<td>0.76</td>
<td>0.70115</td>
<td>0.09</td>
</tr>
<tr>
<td>4</td>
<td>4.94433</td>
<td>1.11</td>
<td>0.70204</td>
<td>0.04</td>
</tr>
<tr>
<td>5</td>
<td>4.95092</td>
<td>0.98</td>
<td>0.70235</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>4.97987</td>
<td>0.40</td>
<td>0.70198</td>
<td>0.02</td>
</tr>
<tr>
<td>Analytical</td>
<td>5.00000</td>
<td></td>
<td>0.70178</td>
<td></td>
</tr>
</tbody>
</table>
TEST II

The second test is also based on scheme II and aims at launching a large number of rays using only one beam centered along the tube’s origin with a specific cross section power function, such that all rays reaching the focal point have an equal exit power. Since the power deposited depends on the ray path length, the cross section power function must be such that each ray gets the appropriate initial power to account for the differences in path lengths. For a specific ray we have for the exit power

\[ P_{\text{exit,ray}} = P \sum w_{\text{ray}} \exp \left\{ -\nu_{ib,w} t_{\text{cross}} \left[ 1 + \frac{AR_{\text{ray}}}{2n_w} \right] \right\} \],

(16.202)

where equations 16.195, 16.185 and 16.118 have been used and \( R_{\text{ray}} = (x_0 - x_w)^2 + (z_0 - z_w)^2 \) is the ray’s radial distance from the tube origin. If we set the ray weighting function equal to the inverse gaussian function

\[ w_{\text{ray}} = e^{-\frac{A\nu_{ib,w} t_{\text{cross}}}{2n_w} R_{\text{ray}}^2}, \]

(16.203)

we obtain

\[ P_{\text{exit,ray}} = \frac{P}{Z} e^{-\nu_{ib,w} t_{\text{cross}}}, \]

(16.204)

where we have used \( Z \) for the power partition function \( \sum w_{\text{ray}} \). For a particular number of rays per beam, \( Z \) is constant and therefore the exit power of all rays are equal. To get independent of the number of rays, the unit test records the partition scaled exit power \( P_{\text{exit,ray}}Z \) for each ray as well as their focal \((x_{\text{exit}},z_{\text{exit}})\) coordinates. A maximum square radial deviation

\[ \Delta R^2_{\text{focal}} = \max \left\{ (x_{\text{exit}} - x_w)^2 + (z_{\text{exit}} - z_w)^2 \right\} \]

(16.205)

from the focal point is then defined and recorded. The maximum absolute deviation in the partition scaled power is registered as

\[ \Delta P_{\text{focal}} = \max |P_{\text{exit,ray}}Z - \frac{P}{Z} e^{-\nu_{ib,w} t_{\text{cross}}}|. \]

(16.206)

Both quantities are then checked against tolerance values.

16.5 Heatexchange

The Heatexchange unit implements coupling among the different components (Ion, Electron, Radiation) as a simple relaxation law.

\[ \frac{\partial \rho_{E_{\text{ele}}}}{\partial t} = \kappa_{i,e}(T_{\text{Ion}} - T_{\text{Ele}}) + \kappa_{e,r}(T_{\text{Rad}} - T_{\text{Ele}}) \]

(16.207)
\[ \frac{\partial E_{\text{Rad}}}{\partial t} = -\kappa_{e,r}(T_{\text{Rad}} - T_{\text{Ele}}) \]

(16.208)
\[ \frac{\partial \rho_{E_{\text{ion}}}}{\partial t} = -\kappa_{i,e}(T_{\text{Ion}} - T_{\text{Ele}}) \]

(16.209)

This unit is required because FLASH operator splits the ion/electron equilibration term from the rest of the calculations. All realistic 3T HEDP simulations should include the Heatexchange unit since it is responsible for equilibrating the ion/electron temperatures over time. The radiation terms in (16.207) are only included when the legacy gray radiation diffusion unit is used. The more sophisticated multigroup radiation diffusion unit, described in Chapter 22, includes the emission and absorption internally.

There are three different implementations residing under Heatexchange unit,
• **Constant**: `hx_couplingConst12`, `hx_couplingConst13` and `hx_couplingConst23` can be used for setting the constant coupling terms.

• **ConstCoulomb**: Uses a constant coulomb logarithm to compute ion-electron equilibration. `hx_coulombLog` can be used to set this value.

• **Spitzer**: Uses a Spitzer ion-electron equilibration. `hx_ieTimeCoef` can be used to ion-electron equilibration time coefficient.

The **Constant** and **ConstCoulomb** implementations are really designed to be used for testing purposes. The **Spitzer** implementation should be used for simulations of actual HEDP experiments since it is more accurate and physically realistic. This is described in detail in **Section 16.5.1**.

`hx_relTol` runtime parameter affects the time step computed by Heatexchange_computeDt. Basically, if the max (abs) temperature adjustment that would be introduced in any nonzero component in any cell is less than `hx_relTol`, then the time step limit is relaxed. Set to a negative value to inherit the value of runtime parameter `eos_tolerance`.

**Usage**: To use any implementation of Heatexchange unit, a simulation should include the Heatexchange using an option like `-with-unit=physics/sourceTerms/Heatexchange/HeatexchangeMain/ConstCoulomb` on the setup line, or add `REQUIRES physics/sourceTerms/Heatexchange/HeatexchangeMain/ConstCoulomb` in the Config file. The same process can be repeated for other implementations. Constant implementation is set as the default.

![Energy exchange verification test](image)

Figure 16.16: Verification test for heat exchange code units, showing electrons and ions at different temperatures coming into mutual equilibrium, as expected from the form of the electron-ion coupling. The time axis unit is equal to the asymptotic value of the exponential decay time.

### 16.5.1 Spitzer Heat Exchange

The **Spitzer** implementation should be used to model ion-electron equilibration in realistic HEDP simulations. See **Section 25.7.5** for an example of how to use the **Spitzer Heatexchange** implementation in an HEDP simulation. Note, that this implementation only models ion-electron equilibration, and does not include any radiation related physics. In that sense it is designed to be used in conjunction with the multi-group radiation unit described in diffusion **Chapter 22**, since this unit includes the emission and absorption
terms internally. Thus the equations solved by the Spitzer implementation are:

\[
\frac{de_{\text{ion}}}{dt} = \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}) \\
\frac{de_{\text{ele}}}{dt} = \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}})
\]

where:

- \(T_{\text{ele}}\) is the electron temperature
- \(T_{\text{ion}}\) is the ion temperature
- \(e_{\text{ele}}\) is the electron specific internal energy
- \(e_{\text{ion}}\) is the ion specific internal energy
- \(c_{v,\text{ele}}\) is the electron specific internal energy
- \(\tau_{ei}\) is the ion/electron equilibration time

We then replace \(de\) with \(c_vdT\) for the ions and electrons to obtain:

\[
\frac{dT_{\text{ion}}}{dt} = \frac{m}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}) \\
\frac{dT_{\text{ele}}}{dt} = \frac{1}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}})
\]

where \(m = c_{v,\text{ele}}/c_{v,\text{ion}}\) is the ratio of specific heats. (16.211) can be solved by assuming that the specific heats and ion/electron equilibration times are constant over a time step, \(\Delta t\). The result is that the ion/electron temperatures equilibrate exponentially over a time step. This is shown in (16.212).

\[
T_{\text{ion}}^{n+1} = \left(\frac{T_{\text{ion}}^n + mT_{\text{ele}}^n}{1 + m}\right) - m \left(\frac{T_{\text{ele}}^n - T_{\text{ion}}^n}{1 + m}\right) \exp\left[-(1 + m)\frac{\Delta t}{\tau_{ei}}\right] \quad (16.212a)
\]

\[
T_{\text{ele}}^{n+1} = \left(\frac{T_{\text{ion}}^n + mT_{\text{ele}}^n}{1 + m}\right) + \left(\frac{T_{\text{ele}}^n - T_{\text{ion}}^n}{1 + m}\right) \exp\left[-(1 + m)\frac{\Delta t}{\tau_{ei}}\right] \quad (16.212b)
\]

The Spitzer implementation does not directly update the temperatures, however, since this would lead to energy conservation errors. Rather, the specific internal energies are directly updated according to:

\[
e_{\text{ele}}^{n+1} = e_{\text{ele}}^n + c_{v,\text{ele}}(T_{\text{ele}}^{n+1} - T_{\text{ele}}^n) \quad (16.213a)
\]

\[
e_{\text{ion}}^{n+1} = e_{\text{ion}}^n + c_{v,\text{ion}}(T_{\text{ion}}^{n+1} - T_{\text{ion}}^n) \quad (16.213b)
\]

The final temperatures are then computed using a call to the equation of state.

The subroutine hx_ieEquilTime is responsible for computing the ion/electron equilibration time, \(\tau_{ei}\). Currently, the formula used is consistent with that given in (Atzeni, 2004) and the NRL Plasma Formulary (Huba, 2011). The value of \(\tau_{ei}\) is shown in (16.214).

\[
\tau_{ei} = \frac{3k_B^{3/2}}{8\sqrt{2\pi}e^4} \left(\frac{m_{\text{ion}}T_{\text{ele}} + m_{\text{ele}}T_{\text{ion}}}{m_{\text{ele}}m_{\text{ion}}}\right)^{3/2} \bar{z}^2 n_{\text{ion}} \ln \Lambda_{ei}. \quad (16.214)
\]

where:

- \(k_B\) is the Boltzmann constant
- \(e\) is the electron charge
- \(m_{\text{ion}}\) is the average mass of an ion
• $m_{\text{ele}}$ is the mass of an electron
• $\bar{\xi}$ is the average ionization as computed by the EOS
• $n_{\text{ion}}$ is the ion number density in the plasma
• $\ln \Lambda_{ei}$ is the Coulomb Logarithm associated with ion/electron collision, further description below.

The Coulomb logarithm is the ratio of the maximum to minimum impact parameter for ion/electron collisions and, in general, should always be greater than one. Commonly in HED plasmas, this condition is violated. As a result, to allow the simulation to run, a floor of 1 is placed on the value of $\ln \Lambda_{ei}$. The expression used is given in (16.215) and is consistent with (Brysk, 1974).

$$
\ln \Lambda_{ei} = \ln \left[ \frac{\sqrt{k_B T_{\text{ele}}/4\pi e^2 n_{\text{ele}}}}{\max \left( \bar{\xi} e^2 / 3k_B T_{\text{ele}}, \frac{\hbar}{2\sqrt{3k_B T_{\text{ele}}}} n_{\text{ele}} \right)} \right] \quad (16.215)
$$

where $n_{\text{ele}}$ is the electron number density and $\hbar$ is Planck’s constant. The runtime parameter $hx_{\text{ieTimeCoef}}$ multiplies $\tau_{ei}$ and can be used to scale $\tau_{ei}$.

### 16.6 Flame

The Flame unit implements an artificial reaction front model based on a simplified single-variable reaction-diffusion model. This is intended to be used to model the behavior of under-resolved reaction fronts propagated spatially by thermal diffusion, commonly called flames or deflagration fronts. As the reaction-diffusion front is also carried with the fluid, this model is sometimes termed an advection-diffusion-reaction (ADR) model. The principal usage to date of this model has been during the deflagration phase of models of Type Ia Supernovae. See Townsley et al. (2007), from which some text in this section has been adapted, and works that reference it for examples.

#### 16.6.1 Reaction-Diffusion Forms

Generally, an ADR scheme characterizes the location of a flame front using a reaction progress variable, $\phi$, which increases monotonically across the front from 0 (fuel) to 1 (ash). Evolution of this progress variable is accomplished via an advection-diffusion-reaction equation of the form

$$
\frac{\partial \phi}{\partial t} + \vec{v} \cdot \nabla \phi = \kappa \nabla^2 \phi + \frac{1}{\tau} R(\phi) , \quad (16.216)
$$

where $\vec{v}$ is the local fluid velocity, and the reaction term, $R(\phi)$, timescale, $\tau$, and the diffusion constant, $\kappa$, are chosen so that the front propagates at the desired speed. Vladimirova et al. (2006) showed that the step-function reaction rate leads to a substantial amount of unwanted acoustic noise. They studied a suitable alternative, the Kolmogorov Petrovski Piskunov (KPP) reaction term which has an extensive history in the study of reaction-diffusion equations. In the KPP model the reaction term is given by

$$
R(\phi) = \frac{1}{4} \phi (1 - \phi) . \quad (16.217)
$$

The symmetric and low-order character of this function gives it very nice numerical properties, leading to amazingly little acoustic noise. Following Vladimirova et al. (2006), we adopt $\kappa \equiv s b \Delta x / 16$ and $\tau \equiv b \Delta x / 16 s$, where $\Delta x$ is the grid spacing, $s$ is the desired propagation speed, and $b$ sets the desired front width scaled to represent approximately the number of zones.

The KPP reaction term, however, has two serious drawbacks. Formally, the flame speed is only single valued for initial conditions that are precisely zero (and stay that way) outside the burned region (Xin 2000), which cannot really be effected in a hydrodynamics simulation. This can lead to an unbounded increase of the propagation speed, which is precisely the property we wish to have under good control. Secondly, the progress variable $\phi$ takes an infinite amount of time to actually reach 1 (complete consumption of fuel).
16.6. FLAME

While not a fatal flaw like the flame speed problem, this is a problem for our simulations in which we would like to have a localized flame front so that fully-burned ash can be treated as pure NSE material.

Both of these drawbacks can be ameliorated by a slight modification of the reaction term (S. Asida, private communication) to

\[ R(\phi) = f \frac{1}{4}(\phi - \epsilon_0)(1 - \phi + \epsilon_1), \]

where \( 0 < \epsilon_0, \epsilon_1 \ll 1 \) and \( f \) is an additional factor that depends on \( \epsilon_0 \) and \( \epsilon_1 \) and the flame width so that the flame speed is preserved with the same constants as for KPP above. This “sharpened” KPP (sKPP) has truncated tails in both directions (thus being sharpened), making the flame front fully localized, and is a bi-stable reaction rate and thus gives a unique flame speed (Xin 2000). The price paid is that since \( R(\phi) = 0 \) for \( \phi \leq 0 \) and \( \phi \geq 1 \), (16.218) is discontinuous, adding some noise to the solution. Since the suppression of the tails is stronger for higher \( \epsilon_0 \) and \( \epsilon_1 \), we adjusted \( \epsilon_0 \) and \( \epsilon_1 \) so that for a particular flame width we could meet noise goals (Townsley et al. 2007). The parameter the default parameter values obtained are \( \epsilon_0 = \epsilon_1 = 10^{-3}, f = 1.309, \) and \( b = 3.2 \). The noise properties of these choices are discussed in Townsley et al. (2007).

Diffusive flames are known to be subject to a curvature effect that affects the flame speed when the radius of curvature is similar to the flame thickness, a frequent circumstance with modestly-resolved flame front structure. In testing, the curvature effect of the step-function reaction rate proved surprisingly strong, likely due to the exponential “nose” that the flame front possesses (Vladimirova et al. 2006). Both KPP and sKPP show significantly better curvature properties.

16.6.2 Unit Structure

The unit is divided into several subunits to allow flexibility in how the propagation speed is determined and how the energy release of the reaction front takes place. The main subunit FlameMain just implements the actual the reaction and diffusion terms defined above. There is currently only one implementation, RDSplit5point, that uses operator split steps for the reaction and diffusion, and a 5 point stencil for the Laplacian operator.

The order of operations occurs as follows, for each block of the mesh in sequence: The progress variable, \( \phi \), stored in FLAM_MSCALAR is extracted, the Unit-internal function fl_flameSpeed() is called to fill a temporary array with the flame speed evaluated for each cell in the current block. The mesh scalar FLAM_MSCALAR is then updated based on this flame speed while saving temporarily the effective \( \dot{\phi} \) for this time step. This information is then passed on to the internal function fl_effects() which performs any energy release or other desired side effects of the propagation of the RD front. Note both the temporary arrays storing the flame speed and the \( \dot{\phi} \) for the current block are destroyed before processing the next block. However some implementations of fl_flameSpeed() may save this information in the main grid arrays for informational or other purposes.

16.6.2.1 Flame Speed

The flame speed is generally determined by various physical quantities on the grid. However, since the flame is generally an artificial reaction front, many aspects other than the composition may influence the speed. The simplest alternative implementation for determining the flame speed, which is the default one, is Flame/FlameSpeed/Constant. This simply sets a constant flame speed everywhere on the grid.

The flame speed can also include a turbulence-flame interaction (TFI). The implemented options are discussed in Jackson, Townsley, & Calder (2014, submitted). This is accomplished by including one of the implementations under, for example Flame/FlameSpeed/Constant/TFI. Since the TFI always modifies some base flame speed, it is generally included as a implementation of that type of flame speed, in this case a constant flame speed. In order to facilitate sharing of the TFI components, they are stored in their own implementation directories under Flame/FlameSpeed/turbulent and included as needed. All of the TFI options require the inclusion of the Turb unit which provides a measurement of the local turbulence strength.
16.6.2.2 Flame Effects

The flame effects subunit performs functions related to the actual energy deposition or changes in abundances caused by advance of the reaction front being modeled by the reaction-diffusion progress variable \( \phi \). The simplest implementation, included under Flame/FlameEffects/EIP, implements fixed energy input and change in \( \bar{A} \) and \( \bar{Z} \) across the reaction front. This is constructed to utilize the fully ionized electron-ion plasma EOS called the "Helmholtz" EOS in Flash.

16.7 Turbulence Measurement

The Turb unit implements a method to measure the local turbulence strength at the grid scale. The implementation is based on the operator \( OP_2 \) in Colin et al. (2000),

\[
v' = OP_2(\vec{v}) = c_2 \Delta_x^2 \nabla^2 (\nabla \times \vec{v}) \quad (16.219)
\]

Here \( c_2 \) is a calibration constant, set with the turb_c2 runtime variable. The default value, 0.9, was determined for the PPM hydodynamics method by Jackson et al. (2014, submitted) based on simulations of stirred turbulence. \( \Delta_x \) is the grid spacing and \( \vec{v} \) is the velocity field on the grid. Two options are implemented for sampling the velocity field, each of which uses a 5-point stencil for the Laplacian operator. If turb_stepSize is set to 1, the stencil samples adjacent cells, if set to 2 every other cell is sampled, for an effective stencil size of 9 points. The latter option requires a guardcell fill between the curl and Laplacian operators. The result of equation 16.219 is stored in the TURB mesh variable.
Chapter 17

Diffusive Terms

Figure 17.1: The organizational structure of the Diffuse unit.

The physics/Diffuse unit implements diffusive effects, such as heat conduction, viscosity, and mass diffusivity.

17.1 Diffuse Unit

The Diffuse unit contains four things:

1. Flux-Based Diffusion Solvers: These are explicit diffusion solvers used to compute fluxes for specific processes, such as thermal conduction or magnetic resistivity. These fluxes can then be added to the total fluxes in the hydrodynamic solvers (see Section 17.1.1).

2. General Implicit Diffusion Solvers: The Diffuse unit also contains explicit diffusion solvers that are general purpose. These subroutines can be used to implicitly solve scalar diffusion problems in FLASH (see Section 17.1.2).

3. Flux Limiters: The Diffuse unit contains a general purpose subroutine which modifies diffusion coefficients to incorporate a flux limiters (see Section 17.1.3).

4. Electron Thermal Conduction: The Diffuse unit contains code which uses the general purpose implicit diffusion solver to solve an equation which models electron thermal conduction in a 3T plasma (see Section 17.1.4)
CHAPTER 17. DIFFUSIVE TERMS

17.1.1 Diffuse Flux-Based implementations

There are some important differences between other physics units and the flux-based Diffuse implementations:

- **DiffuseFluxBased** does not operate by modifying solution data arrays (like UNK, etc.) directly. **DiffuseFluxBased** modifies flux arrays instead.

- **DiffuseFluxBased** therefore depends on another physics unit to
  1. define and initialize the flux arrays (before **DiffuseFluxBased** calls add to them), and to
  2. apply the fluxes to actually update the solution data in UNK.

- **DiffuseFluxBased** calls that implement the diffusive effects are not made from **Driver.evolveFlash** as for other physics units and the **DiffuseMain** subunit. Rather, those **DiffuseFluxBased** calls need to be made from cooperating code unit that defines, initializes, and applies the flux arrays. As of FLASH4, among the provided unit implementations only the **PPM** implementation of the **Hydro** unit does this (by calls in the **hy.ppm_sweep** routine).

- The **DiffuseFluxBased** routines that implement the diffusive effects are called separately for each flux direction for each block of cells.

Other Hydro implementations, in particular the MHD implementations, currently implement some diffusive effects in their own flux-based way that does not use the **DiffuseFluxBased** unit.

To use **DiffuseFluxBased** routines of the Diffuse unit, a simulation should

- include the Diffuse unit using an option like `-with-unit=physics/Diffuse/DiffuseFluxBased` on the setup line, or **REQUIRES** `physics/Diffuse/DiffuseFluxBased` in a Config file;

- include a unit that makes Diffuse calls, currently the PPM implementation of Hydro;

- set `useXXX` runtimes parameters appropriately (see below).

The appropriate calls to **DiffuseFluxBased** routines will then be made by the following Hydro code (which can be found in **hy.ppm_sweep.F90**):

```fortran
    call Diffuse_therm(sweepDir, igeom, blockList(blk), numCells,&
                      blkLimits, blkLimitsGC, primaryLeftCoord, &
                      primaryRghtCoord, tempFlx, tempAreaLeft)

    call Diffuse_visc (sweepDir, igeom, blockList(blk), numCells,&
                       blkLimits, blkLimitsGC, primaryLeftCoord,primaryRghtCoord,&
                       tempFlx, tempAreaLeft,secondCoord,thirdCoord)

    !! call Diffuse_species(sweepDir, igeom, blockList(blk), numCells,&
    !!    blkLimits, blkLimitsGC, primaryLeftCoord,primaryRghtCoord,&
    !!    tempFlx, tempFly, tempFlz)
```

To use the Diffuse unit for heat conduction in a simulation, the runtime parameters `useDiffuse` and `useDiffuseTherm` must be set to `.true.`; and to use the Diffuse unit for viscosity effects in a simulation, the runtime parameters `useDiffuse` and `useDiffuseVisc` must be set to `.true.`.

17.1.2 General Implicit Diffusion Solver

FLASH contains a general purpose implicit diffusion solver that can be used to include physics such as electron thermal conduction and multigroup diffusion. The actual code for these solvers is located in **GridSolvers**. Two implementations of these solvers exist in the Diffuse unit. An older and less useful version lies under the split directory. A newer solver which uses the HYPRE library is contained in the unsplit directory. As the name suggests, the split solver is directionally split (default: Strang) to interact
with the pencil grid infrastructure of the GridSolvers/Pfft implementation of the Grid unit. The runtime parameter useDiffuse must be set to .true. to activate the general purpose implicit solver.

The typical diffusion equation that can be solved by this unit takes the form,

\[ A \frac{\partial f}{\partial t} + Cf = \nabla \cdot B \nabla f + D, \]  

(17.1)

where \( f \) is the variable to be diffused in time, and \( A, B, C, D \) are parameters with spatial variation. The infrastructure is provided for a general diffusion equation and can be customized to solve specific equations. The API function Diffuse_solveScalar provides a comprehensive interface to solve different types of equations.

In the unsplit solver the equation is implicitly advanced as follows,

\[ A^n \frac{f^{n+1} - f^n}{\delta t} + C^n f^{n+1} = \theta (\nabla \cdot B^n \nabla f^{n+1}) + (1 - \theta) (\nabla \cdot B^n \nabla f^n) + D, \]  

(17.2)

In the split solver, the solution is advanced along each direction separately. i.e,

\[ A^n \frac{f^{n+\frac{1}{2}} - f^n}{\delta t} + \frac{1}{3} C^n f^{n+\frac{1}{2}} = \theta \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^{n+\frac{1}{2}}) + (1 - \theta) \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^n) + \frac{1}{3} D, \]  

(17.3)

\[ A^n \frac{f^{n+\frac{3}{2}} - f^{n+\frac{1}{2}}}{\delta t} + \frac{1}{3} C^n f^{n+\frac{3}{2}} = \theta \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{3}{2}}) + (1 - \theta) \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{1}{2}}) + \frac{1}{3} D, \]  

(17.4)

\[ A^n \frac{f^{n+1} - f^{n+\frac{3}{2}}}{\delta t} + \frac{1}{3} C^n f^{n+1} = \theta \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+1}) + (1 - \theta) \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+\frac{3}{2}}) + \frac{1}{3} D, \]  

(17.5)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>( \theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>Explicit</td>
</tr>
<tr>
<td>0.5</td>
<td>Crank Nicholson</td>
</tr>
<tr>
<td>1.0</td>
<td>Backward Euler</td>
</tr>
</tbody>
</table>

**Implicitness:** It should be noted that the parameters \( A, B, C \) are time lagged and evaluated at \( t^n \). Whereas keeping with the spirit of the implicit scheme these should have been evaluated at \( t^{n+1} \). Evaluating them at \( t^{n+1} \) makes the problem non-linear for non-constant values of \( A, B \) and \( C \). These equations are much more complicated to solve. Although time lagged coefficient essentially linearized the non-linear problem, the trade off is the scheme is no longer completely implicit. The scheme is fully implicit only in the constant case. This could possibly impact the overall stability characteristics of the scheme.

The Laplacian term is discretized using a central difference scheme as follows,

\[ \nabla \cdot B \nabla f = \frac{B_{i,j,k} (f_{i+1,j,k} - f_{i-1,j,k}) - B_{i-1,j,k} (f_{i,j,k} - f_{i-1,j,k})}{\delta x^2} + \frac{B_{i,j+1,k} (f_{i,j+1,k} - f_{i,j,k}) - B_{i,j,k} (f_{i,j,k} - f_{i,j-1,k})}{\delta y^2} + \frac{B_{i,j,k+1} (f_{i,j,k+1} - f_{i,j,k}) - B_{i,j,k} (f_{i,j,k} - f_{i,j,k-1})}{\delta z^2}, \]  

(17.6)

Where, \( f \) and \( B \) are cell centered variables, and

\[ B_{i+\frac{1}{2},j,k} = \frac{B_{i+1,j,k} + B_{i,j,k}}{2}, \]  

(17.7)

\[ B_{i-\frac{1}{2},j,k} = \frac{B_{i-1,j,k} + B_{i,j,k}}{2}, \]  

(17.8)

\[ B_{i,j+\frac{1}{2},k} = \frac{B_{i,j+1,k} + B_{i,j,k}}{2}, \]  

(17.9)

\[ B_{i,j-\frac{1}{2},k} = \frac{B_{i,j-1,k} + B_{i,j,k}}{2}, \]  

(17.10)

\[ B_{i,j,k+\frac{1}{2}} = \frac{B_{i,j,k+1} + B_{i,j,k}}{2}, \]  

(17.11)

\[ B_{i,j,k-\frac{1}{2}} = \frac{B_{i,j,k-1} + B_{i,j,k}}{2}. \]  

(17.12)
**Paramesh:** In coming up with a numerical scheme in AMR, it is important to ensure that fluxes are conserved at fine-coarse boundary. This is typically done in FLASH using `Grid_conservefluxes` routine. However this is an option which is not available to the implicit solver. Since it uses HYPRE to solve the linear system, the flux conservation should be embedded in the system being solved by HYPRE.

Looking at standard diffusion equation mentioned earlier, for simplicity we set $C = D = 0$.

So,

$$ A \frac{\partial f}{\partial t} = \nabla \cdot B \nabla f, \quad (17.13) $$

We introduce a new variable $\vec{F}$ defined as,

$$ \vec{F} = B \nabla f, \quad (17.14) $$

$$ A \frac{\partial f}{\partial t} = \nabla \cdot \vec{F}, \quad (17.15) $$

We discretize the equation as follows,

$$ A^v f^{n+1} - f^n \equiv \theta (\vec{F}^{n+1} \cdot \partial s) + (1 - \theta)(\vec{F}^n \cdot \partial s) \quad (17.16) $$

The idea is then to compute $\vec{F} \cdot \partial s$ on the fine coarse boundary so as to conserve flux. In a 1D problem the $\vec{F} \cdot \partial s$ on a fine-coarse boundary is computed as,

$$ (F^L)_{i+\frac{1}{2}} = (F^R)_{i+\frac{1}{2}} = B_{i+\frac{1}{2}} \frac{f_{i+1} - f_i}{\delta x_{i+\frac{1}{2}}}, \quad (17.18) $$

$$ (F^L)_{i-\frac{1}{2}} = (F^R)_{i-\frac{1}{2}} = B_{i-\frac{1}{2}} \frac{f_i - f_{i-1}}{\delta x_{i-\frac{1}{2}}} \quad (17.19) $$

The idea is similarly extended to 2D where coarse cell flux is set to sum of computed fine cell fluxes of its neighbors.

$$ F_c ds_c = F_1 ds_1 + F_2 ds_2 \quad (17.21) $$
17.1. DIFFUSE UNIT

Table 17.2: General Implicit Diffusion Solver Boundary Conditions

<table>
<thead>
<tr>
<th>Name</th>
<th>Integer</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neumann</td>
<td>GRID_PDE_BND_NEUMANN</td>
<td>$\nabla f \cdot n = 0$</td>
</tr>
<tr>
<td>Dirichlet</td>
<td>GRID_PDE_BND_DIRICHLET</td>
<td>$f = c$</td>
</tr>
<tr>
<td>Vacuum</td>
<td>VACUUM</td>
<td>$\frac{1}{2} f + \frac{1}{2} \nabla f \cdot n = 0$</td>
</tr>
</tbody>
</table>

Where,

$$F_1 = B_{face} \frac{f_1 - f_c}{\delta x_1}$$

$$F_2 = B_{face} \frac{f_2 - f_c}{\delta x_2}$$

(17.22)

(17.23)

The value of $B_{face}$ can then be evaluated from surrounding cells using paramesh interpolation.

Usage: To use the Split/Unsplit implementation of Diffuse unit, a simulation should include the Diffuse unit using an option like `-with-unit=source/physics/Diffuse/DiffuseMain/Split` on the setup line, or add `REQUIRES physics/Diffuse/DiffuseMain/Split` in the Config file. The same process can be repeated for Unsplit solver. Split and Unsplit solver are mutually exclusive implementations and cannot exist in the same simulation.

Table 17.1: comparison of capabilities of Split and Unsplit diffusion.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Split</th>
<th>Unsplit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paramesh</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Uniform grid</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Cartesian 1D, 2D</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Cartesian 3D</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Cylindrical 1D</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Cylindrical 2D (R-Z)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Spherical 1D</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

17.1.2.1 Boundary Conditions

Unlike much of the rest of FLASH, the general purpose implicit diffusion solver does not implement boundary conditions explicitly through guard cell fills. Instead the diffusion matrix is modified to directly include boundary conditions. The `bcTypes` and `bcValues` arguments to `Diffuse_solveScalar` are used to indicate the boundary condition to use. Each of these arguments is an array of length six corresponding to the six faces of the domain: `imin, imax, jmin, jmax, kmin, kmax` in that order. The `bcTypes` argument is used to set the type of boundary condition to use. Several integer values are acceptable that are defined in the “constants.h” header file. The allowed integer values and the mathematical effect are shown in Table 17.2. When a Dirichlet (constant value) boundary conditions is required, `bcTypes` stores the value to use.

17.1.3 Flux Limiters

The `Diffuse` unit contains the subroutine `Diffuse_fluxLimiter` which modifies a mesh variable representing a diffusion coefficient over the entire mesh to include a flux-limit. The subroutine modifies the
diffusion coefficient to smoothly transition to some maximum flux, $q_{\text{max}}$. The subroutine takes the following arguments:

```fortran
subroutine Diffuse_fluxLimiter(idcoef, ifunc, ifl, mode, blkcnt, blklst)
```

where:

- `idcoef` [integer,input] Mesh variable representing the diffusion coefficient (e.g. `COND_VAR` for electron thermal conduction)
- `ifunc` [integer,input] Mesh variable representing the function being diffused (e.g. `TELE_VAR` for thermal conduction)
- `ifl` [integer,input] Mesh variable representing the flux limit (e.g. `FLLM_VAR` for electron thermal conduction)
- `mode` [integer,input] The type of flux-limiter to apply
- `blkcnt` [integer,input] Number of blocks
- `blklst` [integer array, input] List of blocks

The subroutine modifies the mesh variable whose index is given by `idcoef` by applying a flux-limit. Many methods of incorporating flux-limits into the diffusion coefficient exist. `Diffuse_fluxLimiter` currently supports three of these: “harmonic”, “min/max”, “Larsen”, and “none”. The `mode` argument selects the specific method to use. The modes are integers defined in the “constants.h” header file. String representations also exist and can be converted to the integer values using the `RuntimeParameters_mapStrToInt` subroutine. The string representations can be used to create runtime parameters which represent a flux-limiter mode. Table 17.3 shows the integer constants used as arguments for `Diffuse_fluxLimiter` as well as the string representations of these constants.

Each flux limiter mode modifies the existing diffusion coefficient to incorporate the flux limiter in a different way. If $D$ is the original diffusion coefficient and $D_{fl}$ is the flux-limited diffusion coefficient then:

- the “none” flux limiter sets:
  \[ D_{fl} = D \]  \hspace{1cm} (17.24)

- the “harmonic” flux limiter sets:
  \[ D_{fl} = \frac{1}{D + \frac{|\nabla f|}{q_{\text{max}}}} \]  \hspace{1cm} (17.25)

- the “Larsen” flux limiter sets:
  \[ D_{fl} = \left( \frac{1}{D} \right)^2 + \left( \frac{|\nabla f|}{q_{\text{max}}} \right)^2 \right)^{1/2} \]  \hspace{1cm} (17.26)

- the “min/max” flux limiter sets:
  \[ D_{fl} = \min \left( D, \frac{q_{\text{max}}}{|\nabla f|} \right) \]  \hspace{1cm} (17.27)

where $f$ is the value of the function being diffused. The gradient of $f$ is computed using a second order central difference.

### Table 17.3: String and Integer Representations of Flux-Limiter Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Integer Constant</th>
<th>String Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>FL_NONE</td>
<td>&quot;fl_none&quot;</td>
</tr>
<tr>
<td>Harmonic</td>
<td>FL_HARMONIC</td>
<td>&quot;fl_harmonic&quot;</td>
</tr>
<tr>
<td>Min/Max</td>
<td>FL_MINMAX</td>
<td>&quot;fl_minmax&quot;</td>
</tr>
<tr>
<td>Larsen</td>
<td>FL_LARSEN</td>
<td>&quot;fl_larsen&quot;</td>
</tr>
</tbody>
</table>

The subroutine takes the following arguments:

```fortran
subroutine Diffuse_fluxLimiter(idcoef, ifunc, ifl, mode, blkcnt, blklst)
```
17.1.4 Stand-Alone Electron Thermal Conduction

The Diffuse unit also contains a subroutine which solves the diffusion equation for the case of electron thermal conduction. This code is in the `diff_advancetherm` subroutine that is called by Diffuse. The diffusion equation has the form:

\[ \rho c_{v,\text{ele}} \frac{\partial T_{\text{ele}}}{\partial t} = \nabla \cdot K_{\text{ele}} \nabla T_{\text{ele}} \]  

(17.28)

where \( \rho \) is the mass density, \( c_{v,\text{ele}} \) is the electron specific heat, and \( K_{\text{ele}} \) is the electron thermal conductivity. The conductivity is computed by the Conductivity unit (see Section 21.1 for more information). This equation is solved implicitly over a time step to avoid overly restricted time step constraints. In many cases a flux-limiter is needed in situations where overly large temperature gradients are present. The flux-limit used for electron thermal conduction, \( q_{\text{max,ele}} \), is defined as:

\[ q_{\text{max,ele}} = \alpha_{\text{ele}} n_{\text{ele}} k_B T_{\text{ele}} \sqrt{k_B T_{\text{ele}} \rho_{\text{ele}}} \]  

(17.29)

where:

- \( \alpha_{\text{ele}} \) is the electron conductivity flux-limiter coefficient
- \( n_{\text{ele}} \) is the electron number density
- \( k_B \) is the Boltzmann constant
- \( m_{\text{ele}} \) is the electron mass

The Diffuse.solveScalar subroutine is used for the implicit solve. The use of the electron thermal conduction solver is demonstrated in several simulations. See Section 25.8.1 and Section 25.7.5 for examples. For electron thermal conduction to function, the `useConductivity` and `useDiffuse` runtime parameters must be set to .true.. The following runtime parameters control the behavior of the electron thermal conduction package:

- `diff_useEleCond`: Set to .true. to activate electron thermal conduction
- `diff_eleFlMode`: String indicating which flux-limiter mode to use for electron conduction. Valid values are described in Section 17.1.3.
- `diff_eleFlCoef`: Sets the value of \( \alpha_{\text{ele}} \)
- `diff_thetaImplct`: A number between 0 and 1. When set to 0, an explicit treatment is used for the diffusion solve. When set to 1, a fully implicit solve is used.

The runtime parameters:

- `diff_eleXlBoundaryType`
- `diff_eleXrBoundaryType`
- `diff_eleYlBoundaryType`
- `diff_eleYrBoundaryType`
- `diff_eleZlBoundaryType`
- `diff_eleZrBoundaryType`

are strings which are used to set the boundary conditions for electron conduction on each of the six faces of the domain. Acceptable values are:

- “DIRICHLET”, “Dirichlet”, or “dirichlet” to indicate a Dirichlet boundary condition where \( T_{\text{ele}} \) is set to zero on the boundary
- “OUTFLOW”, “neumann”, “zero-gradient”, “outflow” to indicate a Neumann boundary condition
Chapter 18
Gravity Unit

18.1 Introduction

The Gravity unit supplied with FLASH4 computes gravitational source terms for the code. These source terms can take the form of the gravitational potential $\phi(\mathbf{x})$ or the gravitational acceleration $\mathbf{g}(\mathbf{x})$,

$$\mathbf{g}(\mathbf{x}) = -\nabla \phi(\mathbf{x}) .$$

(18.1)

The gravitational field can be externally imposed or self-consistently computed from the gas density via the Poisson equation,

$$\nabla^2 \phi(\mathbf{x}) = 4\pi G \rho(\mathbf{x}) ,$$

(18.2)
where $G$ is Newton’s gravitational constant. In the latter case, either periodic or isolated boundary conditions can be applied.

## 18.2 Externally Applied Fields

The FLASH distribution includes three externally applied gravitational fields. Each provides the acceleration vector $g(x)$ directly, without using the gravitational potential $\phi(x)$.

When building an application that uses an external, time-independent Gravity implementation, no additional storage in unk for holding gravitational potential or accelerations is needed or defined.

### 18.2.1 Constant Gravitational Field

This implementation creates a spatially and temporally constant field parallel to one of the coordinate axes. The magnitude and direction of the field can be set at runtime. This unit is called Gravity/GravityMain/-Constant.

### 18.2.2 Plane-parallel Gravitational field

This PlanePar version implements a time-constant gravitational field that is parallel to one of the coordinate axes and falls off with the square of the distance from a fixed location. The field is assumed to be generated by a point mass or by a spherically symmetric mass distribution. A finite softening length may optionally be applied.

This type of gravitational field is useful when the computational domain is large enough in the direction radial to the field source that the field is not approximately constant, but the domain’s dimension perpendicular to the radial direction is small compared to the distance to the source. In this case the angular variation of the field direction may be ignored. The PlanePar field is cheaper to compute than the PointMass field described below, since no fractional powers of the distance are required. The acceleration vector is parallel to one of the coordinate axes, and its magnitude drops off with distance along that axis as the inverse distance squared. Its magnitude and direction are independent of the other two coordinates.

### 18.2.3 Gravitational Field of a Point Mass

This PointMass implementation describes the gravitational field due to a point mass at a fixed location. A finite softening length may optionally be applied. The acceleration falls off with the square of the distance from a given point. The acceleration vector is everywhere directed toward this point.

## 18.3 Self-gravity

The self-consistent gravity algorithm supplied with FLASH computes the Newtonian gravitational field produced by the matter. The produced potential function satisfies Poisson’s equation (18.2). This unit’s implementation can also return the acceleration field $g(x)$ computed by finite-differencing the potential using the expressions

\[
\begin{align*}
g_{x;ijk} &= \frac{1}{2\Delta x} (\phi_{i-1,j,k} - \phi_{i+1,j,k}) + O(\Delta x^2) \\
g_{y;ijk} &= \frac{1}{2\Delta y} (\phi_{i,j-1,k} - \phi_{i,j+1,k}) + O(\Delta y^2) \\
g_{z;ijk} &= \frac{1}{2\Delta z} (\phi_{i,j,k-1} - \phi_{i,j,k+1}) + O(\Delta z^2). \tag{18.3}
\end{align*}
\]

In order to preserve the second-order accuracy of these expressions at jumps in grid refinement, it is important to use quadratic interpolants when filling guard cells at such locations. Otherwise, the truncation error of the interpolants will produce unphysical forces at these block boundaries.

Two algorithms are available for solving the Poisson equations: Gravity/GravityMain/Multipole and Gravity/GravityMain/Multigrid. The initialization routines for these algorithms are contained in the
Gravity unit, but the actual implementations are contained below the Grid unit due to code architecture constraints.

The multipole-based solver described in Section 8.10.2.1 for self gravity is appropriate for spherical or nearly-spherical mass distributions with isolated boundary conditions. For non-spherical mass distributions higher order moments of the solver must be used. Note that storage and CPU costs scale roughly as the square of number of moments used, so it is best to use this solver only for nearly spherical matter distributions.

The multigrid solver described in Section 8.10.2.6 is appropriate for general mass distributions and can solve problems with more general boundary conditions.

The tree solver described in 8.10.2.4 is appropriate for general mass distributions and can solve problems with both isolated and periodic boundary conditions set independently in individual directions.

18.3.1 Coupling Gravity with Hydrodynamics

The gravitational field couples to the Euler equations only through the momentum and energy equations. If we define the total energy density as

$$\rho E \equiv \frac{1}{2} \rho v^2 + \rho \epsilon,$$

(18.4)

where $\epsilon$ is the specific internal energy, then the gravitational source terms for the momentum and energy equations are $\rho g$ and $\rho \mathbf{v} \cdot \mathbf{g}$, respectively. Because of the variety of ways in which different hydrodynamics schemes treat these source terms, the gravity module only supplies the potential $\phi$ and acceleration $\mathbf{g}$, leaving the implementation of the fluid coupling to the hydrodynamics module. Finite-difference and finite-volume hydrodynamic schemes apply the source terms in their advection steps, sometimes at multiple intermediate timesteps and sometimes using staggered meshes for vector quantities like $\mathbf{v}$ and $\mathbf{g}$.

For example, the PPM algorithm supplied with FLASH uses the following update steps to obtain the momentum and energy in cell $i$ at timestep $n+1$

$$\begin{align*}
(\rho v)_i^{n+1} &= (\rho v)_i^n + \frac{\Delta t}{2} g_i^{n+1} (\rho v_i^n + \rho v_i^{n+1}) \\
(\rho E)_i^{n+1} &= (\rho E)_i^n + \frac{\Delta t}{2} g_i^{n+1} (\rho v_i^n + \rho v_i^{n+1}) (v_i^n + v_i^{n+1}).
\end{align*}$$

(18.5)

Here $g_i^{n+1}$ is obtained by extrapolation from $\phi_i^{n-1}$ and $\phi_i^n$. The Poisson gravity implementation supplies a mesh variable to contain the potential from the previous timestep; future releases of FLASH may permit the storage of several time levels of this quantity for hydrodynamics algorithms that require more steps. Currently, $\mathbf{g}$ is computed at cell centers.

Note that finite-volume schemes do not retain explicit conservation of momentum and energy when gravity source terms are added. Godunov schemes such as PPM, require an additional step in order to preserve second-order time accuracy. The gravitational acceleration component $g_i$ is fitted by interpolants along with the other state variables, and these interpolants are used to construct characteristic-averaged values of $\mathbf{g}$ in each cell. The velocity states $v_{L,i+1/2}$ and $v_{R,i+1/2}$, which are used as inputs to the Riemann problem solver, are then corrected to account for the acceleration using the following expressions

$$\begin{align*}
v_{L,i+1/2} &\to v_{L,i+1/2} + \frac{\Delta t}{4} \left( g_{L,i+1/2}^+ + g_{L,i+1/2}^- \right) \\
v_{R,i+1/2} &\to v_{R,i+1/2} + \frac{\Delta t}{4} \left( g_{R,i+1/2}^+ + g_{R,i+1/2}^- \right).
\end{align*}$$

(18.6)

Here $g_{X,i+1/2}^\pm$ is the acceleration averaged using the interpolant on the $X$ side of the interface ($X = L, R$) for $v \pm c$ characteristics, which bring material to the interface between cells $i$ and $i + 1$ during the timestep.

18.3.2 Tree Gravity

Tree gravity unit in physics/Gravity/GravityMain/Poisson/BHTree is supposed to be used together with the tree solver unit Grid/GridSolvers/BHTree/Wunsch for calculation of the gravitational potential. It provides subroutines for building and walking the tree called by the tree solver. In this version, only monopole moments (node masses) are used for the potential calculation.
It also defines new multipole acceptance criteria (MACs) that estimate the error in gravitational acceleration of a contribution of a single node to the potential (hereafter partial error) much better than purely geometrical MAC defined in the tree solver. They are: (1) the approximate partial error (APE), and (2) the maximum partial error (MPE). The first one is based on an assumption that the partial error is proportional to multipole moment of the node. The node is accepted for calculation of the potential if

$$D^{m+2} > \frac{GM S_{node}^m}{\Delta a_{p, APE}}$$  \hspace{1cm} (18.7)

where $D$ is distance between the point-of-calculation and the node, $M$ is the node mass, $S_{node}$ is the node size, $m$ is a degree of the multipole approximation and $\Delta a_{p, APE}$ is the required error in acceleration (controlled by runtime parameter $\text{grv\_bhAccErr}$). Since only monopole moments are used for the potential calculation, the most reasonable choice of $m$ seems to be $m = 2$. This MAC is similar to the one used in Gadget2 (see Springer, 2005, MNRAS, 364, 1105).

The second MAC (maximum partial error, MPE) calculates the error in acceleration of a single node contribution $\Delta a_{p, MPE}$ according to formula 9 from Salmon & Warren (1994; see this paper for details):

$$\Delta a_{p, MPE} \leq \frac{1}{D^2} \left( \frac{1}{1 - S_{node}/D} \right)^2 \left( \frac{3|B_2|}{D^2} - \frac{2|B_3|}{D^3} \right)$$ \hspace{1cm} (18.8)

where $B_n = \sum_i m_i |\vec{r}_i - \vec{r}_0|^n$ where $m_i$ and $\vec{r}_i$ are masses and positions of individual grid cells within the node and $\vec{r}_0$ is the node mass centre position. Moment $B_2$ can be easily determined during the tree build, moment $B_3$ can be estimated as $B_3^2 \geq B_2^3/M$. The maximum allowed partial error in gravitational acceleration is controlled by runtime parameters $\text{grv\_bhAccErr}$ and $\text{grv\_bhUseRelAccErr}$ (see 18.4.1).

During the tree walk, subroutines $\text{Gravity\_bhNodeContrib}$ and $\text{Gravity\_bhBotNodeContrib}$ add to the gravitational potential contributions

$$\Phi = \frac{GM}{|\vec{r}|}$$ \hspace{1cm} (18.9)

in the case of isolated boundary conditions, and

$$\Phi = GM f_{EF}(\vec{r})$$ \hspace{1cm} (18.10)

in the case of periodic or mixed boundary conditions. In the above two formulae, $G$ is the constant of gravity, $M$ is the node mass, $\vec{r}$ is the position vector between point-of-calculation and the node mass centre and $f_{EF}$ is the Ewald field (see below).

Boundary conditions can be isolated or periodic, set independently for each direction. If they are periodic at least in one direction, the Ewald method is used for the potential calculation (Ewald, P. P., 1921, Ann. Phys. 64, 253). The original Ewald method is an efficient method for computing gravitational field for periodic boundary conditions in three directions. Ewald speeded up evaluation of the gravitational potential by splitting it into two parts, $Gm/\vec{r} = Gm \text{erf}(\alpha \vec{r})/r + Gm \text{erfc}(\alpha \vec{r})/r$ ($\alpha$ is an arbitrary constant) and then by applying Poisson summation formula on erfc terms, gravitational field at position $\vec{r}$ can be written in the form

$$\phi(\vec{r}) = -G \sum_{a=1}^{N} m_a \left( \sum_{i_1, i_2, i_3} A_S(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3}) + A_L(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3}) \right),$$ \hspace{1cm} (18.11)

the first sum runs over whole computational domain, where at position $\vec{r}_a$ is mass $m_a$. Second sum runs over all neighbouring computational domains, which are at positions $\vec{l}_{i_1, i_2, i_3}$ and $A_S(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3})$ and $A_L(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3})$ are short, resp. long-range contributions. It is sufficient to take into account only few terms in eq. 18.11.

We modified Ewald method for problems with periodic boundary conditions in two directions and isolated boundary conditions in the third direction and for problems with periodic boundary conditions in one direction and isolated in two directions.
18.4 Usage

To include the effects of gravity in your FLASH executable, include the option

```
-with-unit=physics/Gravity
```
on your command line when you configure the code with `setup`. The default implementation is `Constant`, which can be overridden by including the entire path to the specific implementation in the command line or `Config` file. The other available implementations are `Gravity/GravityMain/PlanePar`, `Gravity/GravityMain/Pointmass` and `Gravity/GravityMain/Poisson`. The `Gravity` unit provides accessor functions to get gravitational acceleration and potential. However, none of the external field implementations of Section 18.2 explicitly compute the potential, hence they inherit the null implementation from the API for accessing potential. The gravitational acceleration can be obtained either on the whole domain, a single block or a single row at a time.

When building an application that solves the Possion equation for the gravitational potential, additional storage is needed in `unk` for holding the last, as well as (usually) the previous, gravitational potential field; and, depending on the Possion solver used, additional variables may be needed. The variables `GPOT_VAR` and `GPOT_VAR`, and others as needed, will be automatically defined in `Flash.h` in those cases. See `Gravity_potentialListOfBlocks` for more information.

18.4.1 Tree Gravity Unit Usage

Calculation of gravitational potential can be enabled by compiling in this unit and setting the runtime parameter `useGravity` true. The constant of gravity can be set independently by runtime parameter `grv_bhNewton`; if it is smaller than 1 the constant `newton` from the internal Flash unit database is used. If parameters `gr_bhPhysMACTW` or `gr_bhPhysMACComm` are set, the gravity unit MAC is used and it can be chosen by setting `grv_bhMAC` to either `ApproxPartialErr` or `MaxPartialErr`. If the first one is used, the order of the multipole approximation is given by `grv_bhMPDegree`.

The maximum allowed partial error in gravitational acceleration is set by runtime parameter `grv_bhAccErr`. It has either a meaning of an error in absolute acceleration or in relative acceleration normalized by the acceleration from the previous time-step. The latter is used if `grv_bhUseRelAccErr` is set to True, and in this case the first call of the tree solver calculates the potential using purely geometrical MAC (because the acceleration from the previous time-step does not exist).

Boundary conditions are set by the runtime parameter `grav_boundary_type` and they can be `isolated`, `periodic` or `mixed`. In the case of mixed boundary conditions, runtime parameters `grav_boundary_type_i`, `grav_boundary_type_y` and `grav_boundary_type_z` specify along which coordinate are boundary conditions periodic and isolated (possible values are `periodic` or `isolated`). Arbitrary combination of these values is permitted, thus suitable for problems with planar resp. linear symmetry. It should work for computational domain with arbitrary dimensions. The highest accuracy is reached with blocks of cubic physical dimensions.

If runtime parameter `grav_boundary_type` is `periodic` or `mixed`, then the Ewald field for appropriate symmetry is calculated at the beginning of the simulation. Range in which run indices $i_1, i_2, i_3$ in (eq. 18.11) is controlled by `grv_bhEwaldSeriesN`. The Ewald field is then stored in a nested set of grids, the first of them corresponds in size to full computational domain, and each following grid is half the size (in each direction) of the previous grid. Number of nested grids is controlled by variable `grv_bhEwaldNRef`. If `grv_bhEwaldNRef` is too low to cover origin (where is the Ewald field discontinous), then the run is terminated. Each grid is composed of `grv_bhEwaldFieldNx x grv_bhEwaldFieldNy x grv_bhEwaldFieldNz` points. Whenever during run, when evaluation of the Ewald Field at particular point is needed, its value is found by interpolation in suitable level of grid. Linear or semi-quadratic interpolation can be chosen by runtime parameter `grv_bhLinearInterpolOnly` (option true corresponds to linear interpolation). Semi-quadratic interpolation is recommended only in the case when there are periodic boundary conditions in two directions.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grv_bhNewton</td>
<td>real</td>
<td>-1.0</td>
<td>constant of gravity; if (\neq 0), it is obtained from internal physical constants database</td>
</tr>
<tr>
<td>grv_bhMAC</td>
<td>string</td>
<td>&quot;ApproxPartialErr&quot;</td>
<td>MAC, other option: &quot;MaxPartialErr&quot;</td>
</tr>
<tr>
<td>grv_bhMFDegree</td>
<td>integer</td>
<td>2</td>
<td>degree of multipole in error estimate in APE MAC</td>
</tr>
<tr>
<td>grv_bhUseRelAccErr</td>
<td>logical</td>
<td>.false.</td>
<td>if .true., grv_bhAccErr has meaning of relative error, otherwise absolute</td>
</tr>
<tr>
<td>grv_bhAccErr</td>
<td>real</td>
<td>0.1</td>
<td>maximum allowed error in gravitational acceleration</td>
</tr>
<tr>
<td>grav_boundary_type</td>
<td>string</td>
<td>&quot;isolated&quot;</td>
<td>or &quot;periodic&quot; or &quot;mixed&quot;</td>
</tr>
<tr>
<td>grav_boundary_type_x</td>
<td>string</td>
<td>&quot;isolated&quot;</td>
<td>or &quot;periodic&quot;</td>
</tr>
<tr>
<td>grav_boundary_type_y</td>
<td>string</td>
<td>&quot;isolated&quot;</td>
<td>or &quot;periodic&quot;</td>
</tr>
<tr>
<td>grav_boundary_type_z</td>
<td>string</td>
<td>&quot;isolated&quot;</td>
<td>or &quot;periodic&quot;</td>
</tr>
<tr>
<td>grv_bhEwaldFieldNx</td>
<td>integer</td>
<td>32</td>
<td>size of the Ewald field grid in x-direction</td>
</tr>
<tr>
<td>grv_bhEwaldFieldNy</td>
<td>integer</td>
<td>32</td>
<td>size of the Ewald field grid in y-direction</td>
</tr>
<tr>
<td>grv_bhEwaldFieldNz</td>
<td>integer</td>
<td>32</td>
<td>size of the Ewald field grid in z-direction</td>
</tr>
<tr>
<td>grv_bhEwaldSeriesN</td>
<td>integer</td>
<td>10</td>
<td>number of terms in the Ewald series</td>
</tr>
<tr>
<td>grv_bhEwaldAlwaysGenerate</td>
<td>boolean</td>
<td>true</td>
<td>whether Ewald field should be regenerated</td>
</tr>
<tr>
<td>grv_bhEwaldFName</td>
<td>string</td>
<td>&quot;ewald_field&quot;</td>
<td>name of the file to store the Ewald field</td>
</tr>
<tr>
<td>grv_bhEwaldNRef</td>
<td>integer</td>
<td>-1</td>
<td>number of refinement levels (nested grids) for Ewald field, if (\neq 0), set automatically</td>
</tr>
<tr>
<td>grv_bhLinearInterpolOnly</td>
<td>logical</td>
<td>.true.</td>
<td>if .false., semi-quadratic interpolation is used for interpolation in the Ewald field</td>
</tr>
</tbody>
</table>

### 18.5 Unit Tests

There are two unit tests for the gravity unit. **Poisson3** is essentially the Maclaurin spheroid problem described in Section 25.3.4. Because an analytical solution exists, the accuracy of the gravitational solver can be quantified. The second test, **Poisson3_active**, is a modification of **Poisson3** to test the mapping of particles in *Grid.mapParticlesToMesh*. Some of the mesh density is redistributed onto particles, and the particles are then mapped back to the mesh, using the analytical solution to verify completeness. This test is similar to the simulation **PoisParticles** discussed in Section 25.4.3. **PoisParticles** is based on the Huang-Greengard Poisson gravity test described in Section 25.3.3.
Chapter 19

Particles Unit

Figure 19.1: The \texttt{Particles} unit main subunit.
The support for particles in FLASH4 comes in two flavors, active and passive. Active particles are further classified into two categories; massive and charged. The active particles contribute to the dynamics of the simulation, while passive particles follow the motion of Lagrangian tracers and make no contribution to the dynamics. Particles are dimensionless objects characterized by positions \( x_i \), velocities \( v_i \), and sometimes other quantities such as mass \( m_i \) or charge \( q_i \). Their characteristic quantities are considered to be defined at their positions and may be set by interpolation from the mesh or may be used to define mesh quantities by extrapolation. They move relative to the mesh and can travel from block to block, requiring communication patterns different from those used to transfer boundary information between processors for mesh-based data.

Passive particles acquire their kinematic information (velocities) directly from the mesh. They are meant to be used as passive flow tracers and do not make sense outside of a hydrodynamical context. The governing equation for the \( i \)th passive particle is particularly simple and requires only the time integration of interpolated mesh velocities.

\[
\frac{dx_i}{dt} = v_i \tag{19.1}
\]

Active particles experience forces and may themselves contribute to the problem dynamics (e.g., through long-range forces or through collisions). They may additionally have their own motion independent of the grid, so an additional motion equation of

\[
v_i^{n+1} = v_i^n + a_i^n \Delta t^n . \tag{19.2}
\]

may come into play. Here \( a_i \) is the particle acceleration. Solving for the motion of active particles is also referred to as solving the N-body problem. The equations of motion for the \( i \)th active particle include the equation (19.1) and another describing the effects of forces.

\[
m_i \frac{dv_i}{dt} = F_{lr,i} + F_{sr,i} \tag{19.3}
\]

Here, \( F_{lr,i} \) represents the sum of all long-range forces (coupling all particles, except possibly those handled by the short-range term) acting on the \( i \)th particle and \( F_{sr,i} \) represents the sum of all short-range forces (coupling only neighboring particles) acting on the particle.

For both types of particles, the primary challenge is to integrate (19.1) forward through time. Many alternative integration methods are described in Section 19.1 below. Additional information about the mesh to particle mapping is described in Section 19.2. An introduction to the particle techniques used in FLASH is given by R. W. Hockney and J. W. Eastwood in *Computer Simulation using Particles* (Taylor and Francis, 1988).
19.1. TIME INTEGRATION

FLASH Transition

Please note that the particles routines have not been thoroughly tested with non-Cartesian coordinates; use them at your own risk!

New since FLASH3.1

Since release 3.1 of FLASH, a single simulation can have both active and passive particles defined. FLASH3 and FLASH2 allowed only active or passive particles in a simulation. Because of the added complexity, new Config syntax and new setup script syntax is necessary for Particles. See Section 5.2 for command line options, Section 6.6 for Config syntax, and Section 19.3 below for more details.

FLASH4 includes support for sink particles. These are a special kind of (massive) active particles, with special rules for creation, mass accretion, and interaction with fluid variables and other particles. See Section 19.4 below for information specific to sink particles.

19.1 Time Integration

The active and passive particles have many different time integration schemes available. The subroutine Particles_advance handles the movement of particles through space and time. Because FLASH4 has support for including different types of both active and passive particles in a single simulation, the implementation of Particles_advance may call several helper routines of the form pt_advanceMETHOD (e.g., pt_advanceLeapfrog, pt_advanceEuler_active, pt_advanceCustom), each acting on an appropriate subset of existing particles. The METHOD here is determined by the ADVMETHOD part of the PARTICLETYPE Config statement (or the ADV par of a -particlemethods setup option) for the type of particle. See the Particles_advance source code for the mapping from ADVMETHOD keyword to pt_advanceMETHOD subroutine call.

19.1.1 Active Particles (Massive)

The active particles implementation includes different time integration schemes, long-range force laws (coupling all particles), and short-range force laws (coupling nearby particles). The attributes listed in Table 19.1 are provided by this subunit. A specific implementation of the active portion of Particles_advance is selected by a setup option such as -with-unit=Particles/ParticlesMain/active/massive/Leapfrog, or by specifying something like REQUIRES Particles/ParticlesMain/active/massive/Leapfrog in a simulation’s Config file (or by listing the path in the Units file if not using the -auto configuration option). Further details are given in Section 19.3 below.

Available time integration schemes for active particles include

- **Forward Euler.** Particles are advanced in time from \( t^n \) to \( t^{n+1} = t^n + \Delta t^n \) using the following difference equations:

\[
\begin{align*}
\mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \\
\mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n.
\end{align*}
\]  

(19.4)

Here \( \mathbf{a}_i \) is the particle acceleration. Within FLASH4, this scheme is implemented in Particles/ParticlesMain/active/massive/Euler. This Euler scheme (as well as the Euler scheme for the passive particles) is first-order accurate and is included for testing purposes only. It should not be used in a production run.
• **Variable-timestep leapfrog.** Particles are advanced using the following difference equations

\[
\begin{align*}
    x_i^1 &= x_i^0 + v_i^0 \Delta t^0 \\
    v_i^{1/2} &= v_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
    v_i^{n+1/2} &= v_i^{n-1/2} + C_n a_i^n + D_n a_i^{n-1} \\
    x_i^{n+1} &= x_i^n + v_i^{n+1/2} \Delta t^n.
\end{align*}
\]  

(19.5)

The coefficients \( C_n \) and \( D_n \) are given by

\[
\begin{align*}
    C_n &= \frac{1}{2} \Delta t^n + \frac{1}{3} \Delta t^{n-1} + \frac{1}{2} \left( \frac{\Delta t^{n-2}}{\Delta t^{n-1}} \right) \\
    D_n &= \frac{1}{6} \left( \Delta t^{n-1} - \frac{\Delta t^{n-2}}{\Delta t^{n-1}} \right).
\end{align*}
\]  

(19.6)

By using time-centered velocities and stored accelerations, this method achieves second-order time accuracy even with variable timesteps. Within FLASH4, this scheme is implemented in \texttt{Particles/ParticlesMain/active/massive/Leapfrog}.

• **Cosmological variable-timestep leapfrog.** (\texttt{Particles/ParticlesMain/active/massive/LeapfrogCosmo})

The coefficients in the leapfrog update are modified to take into account the effect of cosmological redshift on the particles. The particle positions \( x \) are interpreted as comoving positions, and the particle velocities \( v \) are interpreted as comoving peculiar velocities (\( \dot{v} = \dot{x} \)). The resulting update steps are

\[
\begin{align*}
    x_i^1 &= x_i^0 + v_i^0 \Delta t^0 \\
    v_i^{1/2} &= v_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
    v_i^{n+1/2} &= v_i^{n-1/2} \left[ 1 - \frac{A^n}{2} \Delta t^n + \frac{1}{3!} \Delta t^n A^n - \frac{1}{6} \Delta t^n A^{n-1} \right] + a_i^n \left[ \frac{\Delta t^{n-1}}{2} + \frac{1}{6} \Delta t^{n-1} A^{n-1} \right] \\
    &\quad + a_i^{n-1} \left[ \frac{\Delta t^{n-2}}{6} - \frac{1}{12} \Delta t^{n-2} A^{n-2} \right] + \frac{1}{2} a_i^n \Delta t^n \Delta t^{n-1} \\
    x_i^{n+1} &= x_i^n + v_i^{n+1/2} \Delta t^n.
\end{align*}
\]

Here we define \( A \equiv -2 \dot{a}/a \), where \( a \) is the scale factor. Note that the acceleration \( a_i^{n-1} \) from the previous timestep must be retained in order to obtain second order accuracy. Using the \texttt{Particles/ParticlesMain/passive/LeapfrogCosmo} time integration scheme only makes sense if the \texttt{Cosmology} module is also used, since otherwise \( a \equiv 1 \) and \( \dot{a} \equiv 0 \).

• **Sink particles** have their own implementation of several advancement methods (with time subcycling), implemented under \texttt{Particles/ParticlesMain/active/Sink}, see description below in Section 19.4.

The leapfrog-based integrators implemented under \texttt{Particles/ParticlesMain/active/massive} supply the additional particle attributes listed in Table 19.2.

### Table 19.1: Particle attributes provided by active particles.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS_PART_PROP</td>
<td>REAL</td>
<td>Particle mass</td>
</tr>
<tr>
<td>ACCX_PART_PROP</td>
<td>REAL</td>
<td>x-component of particle acceleration</td>
</tr>
<tr>
<td>ACCY_PART_PROP</td>
<td>REAL</td>
<td>y-component of particle acceleration</td>
</tr>
<tr>
<td>ACCZ_PART_PROP</td>
<td>REAL</td>
<td>z-component of particle acceleration</td>
</tr>
</tbody>
</table>
Table 19.2: Particle attributes provided by leapfrog time integration.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OACX_PART_PROP</td>
<td>REAL</td>
<td>(x)-component of particle acceleration at previous timestep</td>
</tr>
<tr>
<td>OACY_PART_PROP</td>
<td>REAL</td>
<td>(y)-component of particle acceleration at previous timestep</td>
</tr>
<tr>
<td>OACZ_PART_PROP</td>
<td>REAL</td>
<td>(z)-component of particle acceleration at previous timestep</td>
</tr>
</tbody>
</table>

19.1.2 Charged Particles - Hybrid PIC

Collisionless plasmas are often modeled using fluid magnetohydrodynamics (MHD) models. However, the MHD fluid approximation is questionable when the gyroradius of the ions is large compared to the spatial region that is studied. On the other hand, kinetic models that discretize the full velocity space, or full particle in cell (PIC) models that treat ions and electrons as particles, are computationally very expensive. For problems where the time scales and spatial scales of the ions are of interest, hybrid models provide a compromise. In such models, the ions are treated as discrete particles, while the electrons are treated as a (often massless) fluid. This means that the time scales and spatial scales of the electrons do not need to be resolved, and enables applications such as modeling of the solar wind interaction with planets. For a detailed discussion of different plasma models, see Ledina et al. (2008).

19.1.2.1 The hybrid equations

In the hybrid approximation, ions are treated as particles, and electrons as a massless fluid. In what follows we use SI units. We have \(N_I\) ions at positions \(\mathbf{r}_i(t)\) [m] with velocities \(\mathbf{v}_i(t)\) [m/s], mass \(m_i\) [kg] and charge \(q_i\) [C], \(i = 1, \ldots, N_I\). By spatial averaging we can define the charge density \(\rho_I(\mathbf{r}, t)\) [C/m\(^3\)] of the ions, their average velocity \(\mathbf{u}_I(\mathbf{r}, t)\) [m/s], and the corresponding current density \(\mathbf{J}_I(\mathbf{r}, t) = \rho_I \mathbf{u}_I\) [C/m\(^2\) s\(^{-1}\)]. Electrons are modeled as a fluid with charge density \(\rho_e(\mathbf{r}, t)\) and current density \(\mathbf{J}_e(\mathbf{r}, t) = \rho_e \mathbf{u}_e\). The electron number density is \(n_e = -\rho_e/e\), where \(e\) is the elementary charge. If we assume that the electrons are an ideal gas, then \(p_e = n_e kT_e\), so the pressure is directly related to temperature (\(k\) is Boltzmann’s constant).

The trajectories of the ions are computed from the Lorentz force,

\[
\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i, \quad \frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i} (\mathbf{E} + \mathbf{v}_i \times \mathbf{B}), \quad i = 1, \ldots, N_I
\]

where \(\mathbf{E} = \mathbf{E}(\mathbf{r}, t)\) is the electric field, and \(\mathbf{B} = \mathbf{B}(\mathbf{r}, t)\) is the magnetic field. The electric field is given by

\[
\mathbf{E} = \frac{1}{\rho_I} \left( -\mathbf{J}_I \times \mathbf{B} + \mu_0^{-1} (\nabla \times \mathbf{B}) \times \mathbf{B} \right) - \nabla p_e,
\]

where \(\rho_I\) is the ion charge density, \(\mathbf{J}_I\) is the ion current, \(p_e\) is the electron pressure, and \(\mu_0 = 4\pi \cdot 10^{-7}\) is the magnetic constant. Here we assume that \(p_e\) is adiabatic. Then the relative change in electron pressure is related to the relative change in electron density by

\[
\frac{p_e}{p_{e0}} = \left( \frac{n_e}{n_{e0}} \right)^{\gamma},
\]

where the zero subscript denotes reference values (here the initial values at \(t = 0\)). Then Faraday’s law is used to advance the magnetic field in time,

\[
\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}.
\]
### 19.1.2.2 A cell-centered finite difference hybrid PIC solver

We use a cell-centered representation of the magnetic field on a uniform grid. All spatial derivatives are discretized using standard second order finite difference stencils. Time advancement is done by a predictor-corrector leapfrog method with subcycling of the field update, denoted cyclic leapfrog (CL) by Matthew (1994). An advantage of the discretization is that the divergence of the magnetic field is zero, down to round off errors. The ion macroparticles (each representing a large number of real particles) are deposited on the grid by a cloud-in-cell method (linear weighting), and interpolation of the fields to the particle positions are done by the corresponding linear interpolation. Initial particle positions are drawn from a uniform distribution, and initial particle velocities from a Maxwellian distribution. Further details of the algorithm can be found in Holmström, M. (2011) and references therein, where an extension of the solver that include inflow and outflow boundary conditions was used to model the interaction between the solar wind and the Moon. In what follows we describe the FLASH implementation of the hybrid solver with periodic boundary conditions.

### 19.1.2.3 Hybrid solver implementation

The two basic operations needed for a PIC code are provided as standard operations in FLASH:

- Deposit charges and currents onto the grid: call `Grid_mapParticlesToMesh()`
- Interpolate fields to particle positions: call `Grid_mapMeshToParticles()`

At present the solver is restricted to a Cartesian uniform grid, since running an electromagnetic particle code on an adaptive grid is not straight forward. Grid refinement/coarsening must be accompanied by particle split/join operations. Also, jumps in the cell size can lead to reflected electromagnetic waves.

The equations are stated in SI units, so all parameters should be given in SI units, and all output will be in SI units. The initial configuration is a spatial uniform plasma consisting of two species. The first species, named `pt_picPname_1` consists of particles of mass `pt_picPmass_1` and charge `pt_picPcharge_1`. The initial (at $t = 0$) uniform number density is `pt_picPdensity_1`, and the velocity distribution is Maxwellian with a drift velocity of ($pt_picPvelx_1$, $pt_picPvely_1$, $pt_picPvelz_1$), and a temperature of `pt_picPtemp_1`. Each model macro-particle represents `pt_picPweight_1` real particles. So this parameter will control the total number of macro particles of species 1 in the simulation. At the start of the simulation the whole domain is filled with macro-particles of the given weight, randomly according to a uniform distribution, until the specified number density is reached. The velocity of each macro-particle is randomly drawn from the above Maxwellian distribution.

All the above parameters are also available for a second species, e.g., `pt_picPmass_2`, which is initialized in the same way as the first species. The grid is initialized with the uniform magnetic field (`sim_bx`, `sim_by`, `sim_bz`).

Now for grid quantities. The cell averaged mass density is stored in `pden`, and the charge density in `cden`. The magnetic field is represented by (`grbx`, `grby`, `grbz`). A background field can be set during initialization in (`gtx`, `gty`, `gtz`). We then solve for the deviation from this background field. Care must be take so that the background field is divergence free, using the discrete divergence operator. The easiest way to ensure this is to compute the background field as the rotation of a potential. The electric field is stored in (`gex`, `grey`, `gexz`), the current density in (`gjx`, `gjy`, `gjz`), and the ion current density in (`gjix`, `gjiy`, `gjiz`). A resistivity is stored in `gres`, thus it is possible to have a non-uniform resistivity in the simulation domain, but the default is to set the resistivity to the constant value of `sim_resistivity` everywhere. For post processing, separate fields are stored for charge density and ion current density for species 1 and 2.

Regarding particle properties. Each computational meta-particles is labeled by a positive integer, `specie` and has a mass and charge. As all FLASH particles they also each have a position $r_i = (posx, posy, posz)$ and a velocity $v_i = (velx, vely, velz)$. To be able to deposit currents onto the grid, each particle stores the current density corresponding to the particle, $J_i$. For the movement of the particles by the Lorentz force, we also need the electric and magnetic fields at the particle positions, $E(r_i)$ and $B(r_i)$, respectively.

Regarding the choice of time step. The timestep must be constant, $\Delta t = dtinit = dtmin = dtmax$, since the leap frog solver requires this. For the solution to be stable the time step, $\Delta t$, must be small enough. We
Table 19.3: Runtime parameters for the hybrid solver. Initial values are at $t = 0$. For each parameter for species 1, there is a corresponding parameter for species 2 (named with 2 instead of 1), e.g., $pt\_picPvelx_2$.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pt_picPname_1$</td>
<td>STRING</td>
<td>$&quot;H^+&quot;$</td>
<td>Species 1 name</td>
</tr>
<tr>
<td>$pt_picPmass_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Species 1 mass, $m_i$ [amu]</td>
</tr>
<tr>
<td>$pt_picPcharge_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Species 1 charge, $q_i$ [e]</td>
</tr>
<tr>
<td>$pt_picPdensity_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Initial $n_i$ species 1 [m$^{-3}$]</td>
</tr>
<tr>
<td>$pt_picPtemp_1$</td>
<td>REAL</td>
<td>1.5e5</td>
<td>Initial $T_i$ species 1 [K]</td>
</tr>
<tr>
<td>$pt_picPvelx_1$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $u_i$ species 1 [m/s]</td>
</tr>
<tr>
<td>$pt_picPvely_1$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$pt_picPvelz_1$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$pt_picPweight_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Real particles per macro-particle of species 1</td>
</tr>
<tr>
<td>$sim_bx$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $B$ (at $t = 0$) [T]</td>
</tr>
<tr>
<td>$sim_by$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$sim_bz$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$sim_te$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $T_e$ [K]</td>
</tr>
<tr>
<td>$sim_gam$</td>
<td>REAL</td>
<td>-1.0</td>
<td>Adiabatic exponent for electrons</td>
</tr>
<tr>
<td>$sim_nsub$</td>
<td>INTEGER</td>
<td>3</td>
<td>Number of CL B-field update subcycles (must be odd)</td>
</tr>
<tr>
<td>$sim_rng_seed$</td>
<td>INTEGER</td>
<td>1</td>
<td>Seed for the random number generator (must be $&gt; 0$)</td>
</tr>
</tbody>
</table>

will try and quantify this, and here we assume that the grid cells are cubes, $\Delta x = \Delta y = \Delta z$, and that we have a single species plasma.

First of all, since the time advancement is explicit, there is the standard CFL condition that no particle should travel more than one grid cell in one time step, i.e. $\Delta t \max_i(|v_i|) < \Delta x$. This time step is printed to standard output by FLASH (as $dt\_Part$), and can thus be monitored.

Secondly, we also have an upper limit on the time step due to Whistler waves (Pritchett 2000),

$$\Delta t < \frac{\Omega_i^{-1}}{\pi} \left( \frac{\Delta x}{\delta_i} \right)^2 \sim \frac{n}{B} (\Delta x)^2, \quad \delta_i = \frac{1}{|q_i|} \sqrt{\frac{m_i}{\mu_0 n}},$$

where $\delta_i$ is the ion inertial length, and $\Omega_i = |q_i|B/m_i$ is the ion gyrofrequency.

Finally, we also want to resolve the ion gyro motion by having several time steps per gyration. This will only put a limit on the time step if, approximately, $\Delta x > 5 \delta_i$, and we then have that $\Delta t < \Omega_i^{-1}$.

All this are only estimates that does not take into account, e.g., the initial functions, or the subcycling of the magnetic field update. In practice one can just reduce the time step until the solution is stable. Then for runs with different density and/or magnetic field strength, the time step will need to be scaled by the change in $n_i/B$, e.g., if $n_j$ is doubled, $\Delta t$ can be doubled. The factor $(\Delta x)^2$ implies that reducing the cell size by a factor of two will require a reduction in the time step by a factor of four.

19.1.3 Passive Particles

Passive particles may be moved using one of several different methods available in FLASH4. With the exception of Midpoint, they are all single-step schemes. The methods are either first-order or second-order accurate, and all are explicit, as described below. In all implementations, particle velocities are obtained by mapping grid-based velocities onto particle positions as described in Section 19.2.

Numerically solving Equation (19.1) for passive particles means solving a set of simple ODE initial value problems, separately for each particle, where the velocities $v_i$ are given at certain discrete points in time by the state of the hydrodynamic velocity field at those times. The time step is thus externally given and
Table 19.4: The grid variables for the hybrid solver that are most important for a user. For each property for species 1, there is a corresponding variable for species 2 (named with 2 instead of 1), e.g., \( cde2 \).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cden</td>
<td>PER_VOLUME</td>
<td>Total charge density, ( \rho ) [C/m(^3)]</td>
</tr>
<tr>
<td>grbx</td>
<td>GENERIC</td>
<td>Magnetic field, ( B ) [T]</td>
</tr>
<tr>
<td>grby</td>
<td>GENERIC</td>
<td></td>
</tr>
<tr>
<td>grbz</td>
<td>GENERIC</td>
<td></td>
</tr>
<tr>
<td>grex</td>
<td>GENERIC</td>
<td>Electric field, ( E ) [V/m]</td>
</tr>
<tr>
<td>grey</td>
<td>GENERIC</td>
<td></td>
</tr>
<tr>
<td>grez</td>
<td>GENERIC</td>
<td></td>
</tr>
<tr>
<td>grjx</td>
<td>PER_VOLUME</td>
<td>Current density, ( J ) [A/m(^2)]</td>
</tr>
<tr>
<td>grjy</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
<tr>
<td>grjz</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
<tr>
<td>gjix</td>
<td>PER_VOLUME</td>
<td>Ion current density, ( J_I ) [A/m(^2)]</td>
</tr>
<tr>
<td>gjiy</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
<tr>
<td>gjiz</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
<tr>
<td>cde1</td>
<td>PER_VOLUME</td>
<td>Species 1 charge density, ( \rho_I ) [C/m(^3)]</td>
</tr>
<tr>
<td>jix1</td>
<td>PER_VOLUME</td>
<td>Species 1 ion current density, ( J_I ) [A/m(^2)]</td>
</tr>
<tr>
<td>jiy1</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
<tr>
<td>jiz1</td>
<td>PER_VOLUME</td>
<td></td>
</tr>
</tbody>
</table>

cannot be arbitrarily chosen by a particle motion ODE solver\(^1\). Statements about the order of a method in this context should be understood as referring to the same method if it were applied in a hypothetical simulation where evaluations of velocities \( v_i \) could be performed at arbitrary times (and with unlimited accuracy). Note that FLASH4 does not attempt to provide a particle motion ODE solver of higher accuracy than second order, since it makes little sense to integrate particle motion to a higher order than the fluid dynamics that provide its inputs.

In all cases, particles are advanced in time from \( t^n \) (or, in the case of Midpoint, from \( t^{n-1} \)) to \( t^{n+1} = t^n + \Delta t^n \) using one of the difference equations described below. The implementations assume that at the time when \texttt{Particles\_advance} is called, the fluid fields have already been updated to \( t^{n+1} \), as is the case with the \texttt{Driver\_evolveFlash} implementations provided with FLASH4. A specific implementation of the passive portion of \texttt{Particles\_advance} is selected by a setup option such as \texttt{-with-unit=Particles/ParticlesMain/passive/Euler}, or by specifying something like \texttt{REQUIRES Particles/ParticlesMain/passive/Euler} in a simulation’s \texttt{Config} file (or by listing the path in the \texttt{Units} file if not using the \texttt{-auto} configuration option). Further details are given is Section 19.3 below.

- **Forward Euler (Particles/ParticlesMain/passive/Euler).** Particles are advanced in time from \( t^n \) to \( t^{n+1} = t^n + \Delta t^n \) using the following difference equation:

\[
\text{x}_i^{n+1} = \text{x}_i^n + v^n_i \Delta t^n .
\]  

(19.7)

Here \( v^n_i \) is the velocity of the particle, which is obtained using particle-mesh interpolation from the grid at \( t = t^n \).

Note that this evaluation of \( v^n_i \) cannot be deferred until the time when it is needed at \( t = t^{n+1} \), since at that point the fluid variables have been updated and the velocity fields at \( t = t^n \) are not available any more. Particle velocities are therefore interpolated from the mesh at \( t = t^n \) and stored as particle attributes. Similar concerns apply to the remaining methods but will not be explicitly mentioned every time.

\(^1\)Even though it is possible to do so, see \texttt{Particles\_computeDt}, one does not in general wish to let particles integration dictate the time step of the simulation.
Table 19.5: Important particle properties for the hybrid solver. Note that this is for the computational macro-particles.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>specie</td>
<td>REAL</td>
<td>Particle type (an integer number 1,2,3,...)</td>
</tr>
<tr>
<td>mass</td>
<td>REAL</td>
<td>Mass of the particle, $m_i$ [kg]</td>
</tr>
<tr>
<td>charge</td>
<td>REAL</td>
<td>Charge of the particle, $q_i$ [C]</td>
</tr>
<tr>
<td>jx</td>
<td>REAL</td>
<td>Particle ion current, $J_{ix} = q_i v_i$ [A m]</td>
</tr>
<tr>
<td>jy</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>jz</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>bx</td>
<td>REAL</td>
<td>Magnetic field at particle, $B_i(r, t)$ [T]</td>
</tr>
<tr>
<td>by</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>bz</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>ex</td>
<td>REAL</td>
<td>Electric field at particle, $E_i(r, t)$ [V/m]</td>
</tr>
<tr>
<td>ey</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>ez</td>
<td>REAL</td>
<td></td>
</tr>
</tbody>
</table>

- **Two-Stage Runge-Kutta (Particles/ParticlesMain/passive/RungeKutta).** This 2-stage Runge-Kutta scheme is the preferred choice in FLASH4. It is also the default which is compiled in if particles are included in the setup but no specific alternative implementation is requested. The scheme is also known as Heun’s Method:

  \[
  x_i^{n+1} = x_i^n + \frac{\Delta t^n}{2} \left[ v_i^n + v_i^{*,n+1} \right],
  \]

  where

  \[
  v_i^{*,n+1} = v(x_i^{*,n+1}, t^{n+1}),
  \]

  \[
  x_i^{*,n+1} = x_i^n + \Delta t^n v_i^n.
  \]

  Here $v(x, t)$ denotes evaluation (interpolation) of the fluid velocity field at position $x$ and time $t$; $v_i^{*,n+1}$ and $x_i^{*,n+1}$ are intermediate results; and $v_i^n = v(x_i^n, t^n)$ is the velocity of the particle, obtained using particle-mesh interpolation from the grid at $t = t^n$ as usual.

- **Midpoint (Particles/ParticlesMain/passive/Midpoint).** This Midpoint scheme is a two-step scheme. Here, the particles are advanced from time $t^{n-1}$ to $t^{n+1} = t^{n-1} + \Delta t^{n-1} + \Delta t^n$ by the equation

  \[
  x_i^{n+1} = x_i^{n-1} + v_i^n (\Delta t^{n-1} + \Delta t^n) .
  \]

  The scheme is second order if $\Delta t^n = \Delta t^{n-1}$.

  To get the scheme started, an Euler step (as described for passive/Euler) is taken the first time `Particles/ParticlesMain/passive/Midpoint/pt_advancePassive` is called.

  The Midpoint alternative implementation uses the following additional particle attributes:

  PARTICLEPROP pos2PrevX REAL # two previous x-coordinate
  PARTICLEPROP pos2PrevY REAL # two previous y-coordinate
  PARTICLEPROP pos2PrevZ REAL # two previous z-coordinate

- **Estimated Midpoint with Correction (Particles/ParticlesMain/passive/EstiMidpoint2).** The scheme is second order even if $\Delta t^n = \Delta t^{n+1}$ is not assumed. It is essentially the EstiMidpoint or “Predictor-Corrector” method of previous releases, with a correction for non-constant time steps by using additional evaluations (at times and positions that are easily available, without requiring more particle attributes).

\[2\]They can be considered “predicted” positions and velocities.
Particle advancement follows the equation

\[ \mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t^n \mathbf{v}_i^{\text{comb}}, \quad \tag{19.10} \]

where

\[ \mathbf{v}_i^{\text{comb}} = c_1 \mathbf{v}(\mathbf{x}_i^{n+\frac{1}{2}}, t^n) + c_2 \mathbf{v}(\mathbf{x}_i^{n+\frac{1}{2}}, t^{n+1}) + c_3 \mathbf{v}(\mathbf{x}_i^n, t^n) + c_4 \mathbf{v}(\mathbf{x}_i^n, t^{n+1}) \quad \tag{19.11} \]

is a combination of four evaluations (two each at the previous and the current time),

\[ \mathbf{x}_i^{n+\frac{1}{2}} = \mathbf{x}_i^n + \frac{1}{2} \Delta t^{n-1} \mathbf{v}_i^n \]

are estimated midpoint positions as before in the Estimated Midpoint scheme, and the coefficients

\[ c_1 = c_1(\Delta t^{n-1}, \Delta t^n), \]
\[ c_2 = c_2(\Delta t^{n-1}, \Delta t^n), \]
\[ c_3 = c_3(\Delta t^{n-1}, \Delta t^n), \]
\[ c_4 = c_4(\Delta t^{n-1}, \Delta t^n) \]

are chosen dependent on the change in time step so that the method stays second order when \( \Delta t^{n-1} \neq \Delta t^n \).

Conditions for the correction can be derived as follows: Let \( \Delta t^n_\ast = \frac{1}{2} \Delta t^{n-1} \) the estimated half time step used in the scheme, let \( t^{n+\frac{1}{2}}_\ast = t^n + \Delta t^n_\ast \) the estimated midpoint time, and \( t^{n+\frac{1}{2}} = t^n + \frac{1}{2} \Delta t^n \) the actual midpoint of the \([t^n, t^{n+1}]\) interval. Also write \( \mathbf{x}_i^{\ast,n+\frac{1}{2}} = \mathbf{x}_i^n + \frac{1}{2} \Delta t^n \mathbf{v}_i^n \) for first-order (Euler) approximate positions at the actual midpoint time \( t^{n+\frac{1}{2}} \), and we continue to denote with \( \mathbf{x}_i^{*,n+\frac{1}{2}} \) the estimated positions reached at the estimated midpoint time \( t^{n+\frac{1}{2}}_\ast \).

Assuming reasonably smooth functions \( \mathbf{v}(\mathbf{x}, t) \), we can then write for the exact value of the velocity field at the approximate positions evaluated at the actual midpoint time

\[ \mathbf{v}(\mathbf{x}_i^{\ast,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) = \mathbf{v}(\mathbf{x}_i^n, t^n) + \mathbf{v}_i(\mathbf{x}_i^n, t^n) \frac{1}{2} \Delta t^n + (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) \frac{1}{2} \Delta t^n + O(\frac{1}{2} (\Delta t^n)^2) \quad \tag{19.12} \]

by Taylor expansion. It is known that the propagation scheme \( \mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}(\mathbf{x}_i^{\ast,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) \Delta t \) using these velocities is second order (this is known as the modified Euler method).

On the other hand, expansion of (19.11) gives

\[ \mathbf{v}_i^{\text{comb}} = (c_1 + c_2 + c_3 + c_4) \mathbf{v}(\mathbf{x}_i^n, t^n) + (c_2 + c_4) \mathbf{v}_i(\mathbf{x}_i^n, t^n) \Delta t + (c_1 + c_2) (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) \Delta t^n_\ast + \text{higher order terms in } \Delta t \text{ and } \Delta t^n_\ast. \]

After introducing a time step factor \( f \) defined by \( \Delta t^n_\ast = f \Delta t^n \), this becomes

\[ \mathbf{v}_i^{\text{comb}} = (c_1 + c_2 + c_3 + c_4) \mathbf{v}(\mathbf{x}_i^n, t^n) + (c_2 + c_4) \mathbf{v}_i(\mathbf{x}_i^n, t^n) \Delta t + (c_1 + c_2) (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) f \Delta t + O((\Delta t)^2). \quad \tag{19.13} \]

One can derive conditions for second order accuracy by comparing (19.13) with (19.12) and requiring that

\[ \mathbf{v}_i^{\text{comb}} = \mathbf{v}(\mathbf{x}_i^{\ast,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) + O(\frac{1}{2} (\Delta t)^2) \quad \tag{19.14} \]
It turns out that the coefficients have to satisfy three conditions in order to eliminate from the theoretical difference between numerical and exact solution all $O(\Delta t^n - 1)$ and $O(\Delta t^n)$ error terms:

\[
\begin{align*}
    c_1 + c_2 + c_3 + c_4 &= 1 \quad \text{(otherwise the scheme will not even be of first order)}, \\
    c_2 + c_4 &= \frac{1}{2} \quad \text{(and thus also } c_1 + c_3 = \frac{1}{2}), \\
    c_1 + c_2 &= \frac{\Delta t^n}{\Delta t^{n-1}}.
\end{align*}
\]

The provided implementation chooses $c_4 = 0$ (this can be easily changed if desired by editing in the code). All four coefficients are then determined:

\[
\begin{align*}
    c_1 &= \frac{\Delta t^n}{\Delta t^{n-1}} - \frac{1}{2}, \\
    c_2 &= \frac{1}{2}, \\
    c_3 &= 1 - \frac{\Delta t^n}{\Delta t^{n-1}}, \\
    c_4 &= 0.
\end{align*}
\]

Note that when the time step remains unchanged we have $c_1 = c_2 = \frac{1}{2}$ and $c_3 = c_4 = 0$, and so (19.10) simplifies significantly.

An Euler step, as described for passive/Euler in (19.7), is taken the first time when Particles/ParticlesMain/passive/EstiMidpoint2/pt_advancePassive is called and when the time step has changed too much. Since the integration scheme is tolerant of time step changes, it should usually not be necessary to apply the second criterion; even when it is to be employed, the criteria should be less strict than for an uncorrected EstiMidpoint scheme. For EstiMidPoint2 the timestep is considered to have changed too much if either of the following is true:

\[
\begin{align*}
    \Delta t^n &> \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq \text{pt_dtChangeToleranceUp} \times \Delta t^{n-1} \\
    \Delta t^n &< \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq \text{pt_dtChangeToleranceDown} \times \Delta t^{n-1}
\end{align*}
\]

where pt_dtChangeToleranceUp and pt_dtChangeToleranceDown are runtime parameter specific to the EstiMidPoint2 alternative implementation.

The EstiMidpoint2 alternative implementation uses the following additional particle attributes for storing the values of $x^{n+\frac{1}{2}}$ and $v^{n+\frac{1}{2}}$ between the Particles_advance calls at $t^n$ and $t^{n+1}$:

\[
\begin{align*}
    \text{PARTICLEPROP velPredX REAL} \\
    \text{PARTICLEPROP velPredY REAL} \\
    \text{PARTICLEPROP velPredZ REAL} \\
    \text{PARTICLEPROP posPredX REAL} \\
    \text{PARTICLEPROP posPredY REAL} \\
    \text{PARTICLEPROP posPredZ REAL}
\end{align*}
\]

The time integration of passive particles is tested in the ParticlesAdvance unit test, which can be used to examine the convergence behavior, see Section 19.3.4.

## 19.2 Mesh/Particle Mapping

Particles behave in a fundamentally different way than grid-based quantities. Lagrangian, or passive particles are essentially independent of the grid mesh and move along with the velocity field. Active particles may be located independently of mesh refinement. In either case, there is a need to convert grid-based quantities into similar attributes defined on particles, or vice versa. The method for interpolating mesh quantities
to tracer particle positions must be consistent with the numerical method to avoid introducing systematic error. In the case of a finite-volume methods such as those used in FLASH4, the mesh quantities have cell-averaged rather than point values, which requires that the interpolation function for the particles also represent cell-averaged values. Cell averaged quantities are defined as

\[
f_i (x) = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} f (x') \, dx'
\]  
(19.15)

where \( i \) is the cell index and \( \Delta x \) is the spatial resolution. The mapping back and forth from the mesh to the particle positions must be consistent. In particular, to derive the mapping function for the particles, one requires the mapping back and forth from the mesh to the particle positions to be consistent with the numerical method to avoid introducing systematic error. The mapping back and forth from the mesh to the particle positions is defined in the routines \texttt{Particles\_mapFromMesh} and \texttt{Particles\_mapToMeshOneBlk}.

Specifying the desired mapping method is accomplished by designating the \texttt{MAPMETHOD} in the Simulation Config file for each type of particle. See Section 6.6.1 for more details.

### 19.2.1 Quadratic Mesh Mapping

The quadratic mapping package defines an interpolation back and forth to the mesh which is second order. This implementation is primarily meant to be used with passive tracer particles.

To derive it, first consider a second-order interpolation function of the form

\[
f (x) = A + B (x - x_i) + C (x - x_i)^2.
\]  
(19.16)

Then integrating gives

\[
f_{i-1} = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x - x_i)^2 \right]_{x_{i-3/2}}^{x_{i-1/2}} + \frac{1}{3} C (x - x_i)^3 \bigg|_{x_{i-3/2}}^{x_{i-1/2}}
\]  
(19.17)

\[
f_i = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x - x_i)^2 \right]_{x_{i-1/2}}^{x_{i+1/2}} + \frac{1}{3} C (x - x_i)^3 \bigg|_{x_{i-1/2}}^{x_{i+1/2}}
\]  
(19.18)

and

\[
f_{i+1} = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x - x_i)^2 \right]_{x_{i+1/2}}^{x_{i+3/2}} + \frac{1}{3} C (x - x_i)^3 \bigg|_{x_{i+1/2}}^{x_{i+3/2}}
\]  
(19.19)

We may write these as

\[
\begin{bmatrix}
  f_{i+1} \\
  f_i \\
  f_{i-1}
\end{bmatrix} =
\begin{bmatrix}
  1 & -1 & \frac{13}{12} \\
  1 & 0 & \frac{13}{12} \\
  1 & 1 & \frac{13}{12}
\end{bmatrix}
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2
\end{bmatrix}.
\]  
(19.20)

Inverting this gives expressions for \( A \), \( B \), and \( C \),

\[
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2
\end{bmatrix} =
\begin{bmatrix}
  -\frac{1}{2} & \frac{13}{12} & -\frac{1}{2} \\
  \frac{1}{2} & 0 & \frac{1}{2} \\
  \frac{1}{2} & 1 & \frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
  f_{i+1} \\
  f_i \\
  f_{i-1}
\end{bmatrix}.
\]  
(19.21)

In two dimensions, we want a second-order interpolation function of the form

\[
f (x, y) = A + B (x - x_i) + C (x - x_i)^2 + D (y - y_j) + E (y - y_j)^2 + F (x - x_i) (y - y_j).
\]  
(19.22)
In this case, the cell averaged quantities are given by

\[ f_{i,j} (x, y) = \frac{1}{A} \Delta y \int_{x_{i-1/2}}^{x_{i+1/2}} dx' \int_{y_{j-1/2}}^{y_{j+1/2}} dy' f (x', y') . \] (19.23)

Integrating the 9 possible cell averages gives, after some algebra,

\[
\begin{bmatrix}
  f_{i-1,j-1} \\
  f_{i,j-1} \\
  f_{i+1,j-1} \\
  f_{i-1,j} \\
  f_{i,j} \\
  f_{i+1,j} \\
  f_{i-1,j+1} \\
  f_{i,j+1} \\
  f_{i+1,j+1}
\end{bmatrix} =
\begin{bmatrix}
  1 & -1 & 13 & -1 & 13 & 13 & 1 \\
  1 & 0 & 13 & -1 & 13 & 0 & 0 \\
  1 & 1 & 13 & -1 & 13 & -1 & 13 \\
  1 & -1 & 13 & 0 & 13 & 0 & 0 \\
  1 & 0 & 13 & 0 & 13 & 0 & 0 \\
  1 & 1 & 13 & 0 & 13 & -1 & 13 \\
  1 & 0 & 13 & 1 & 13 & 0 & 0 \\
  1 & 1 & 13 & 1 & 13 & 1 & 13 \\
  1 & 1 & 13 & 1 & 13 & 1 & 13
\end{bmatrix}
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2 \\
  D \Delta y \\
  E \Delta y^2 \\
  F \Delta x \Delta y
\end{bmatrix} .
\] (19.24)

At this point we note that there are more constraints than unknowns, and we must make a choice of the constraints. We chose to ignore the cross terms and take only the face-centered cells next to the cell containing the particle, giving

\[
\begin{bmatrix}
  f_{i,j-1} \\
  f_{i-1,j} \\
  f_{i,j} \\
  f_{i+1,j} \\
  f_{i,j+1}
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 13 & -1 & 13 \\
  1 & -1 & 13 & 0 & 13 \\
  1 & 0 & 13 & 0 & 13 \\
  1 & 1 & 13 & 0 & 13 \\
  1 & 1 & 13 & 1 & 13 \\
\end{bmatrix}
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2 \\
  D \Delta y \\
  E \Delta y^2 \\
\end{bmatrix} .
\] (19.25)

Inverting gives

\[
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2 \\
  D \Delta y \\
  E \Delta y^2 \\
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 13 & -1 & 13 \\
  1 & -1 & 13 & 0 & 13 \\
  1 & 0 & 13 & 0 & 13 \\
  1 & 1 & 13 & 0 & 13 \\
  1 & 1 & 13 & 1 & 13 \\
\end{bmatrix}
\begin{bmatrix}
  f_{i,j-1} \\
  f_{i-1,j} \\
  f_{i,j} \\
  f_{i+1,j} \\
  f_{i,j+1}
\end{bmatrix} .
\] (19.26)

Similarly, in three dimensions, the interpolation function is

\[ f (x, y, z) = A + B (x - x_i) + C (x - x_i)^2 + D (y - y_j) + E (y - y_j)^2 + F (z - z_k) + G (z - z_k)^2 . \] (19.27)

and we have

\[
\begin{bmatrix}
  A \\
  B \Delta x \\
  C \Delta x^2 \\
  D \Delta y \\
  E \Delta y^2 \\
  F \Delta z \\
  G \Delta z^2 \\
\end{bmatrix} =
\begin{bmatrix}
  -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} \\
  0 & 0 & -\frac{1}{2} & 0 & -\frac{1}{2} & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 & 0 \\
  \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  f_{i,j,k-1} \\
  f_{i,j-1,k} \\
  f_{i,j,k} \\
  f_{i,j+1,k} \\
  f_{i,j,k+1}
\end{bmatrix} .
\] (19.28)

Finally, the above expressions apply only to Cartesian coordinates. In the case of cylindrical \((r, z)\) coordinates, we have

\[ f (r, z) = A + B (r - r_i) + C (r - r_i)^2 + D (z - z_j) + E (z - z_j)^2 + F (r - r_i) (z - z_j) . \] (19.29)
and

\[
\begin{bmatrix}
A \\
B \Delta r \\
C \Delta f^2 \\
D \Delta z^2 \\
E \Delta z^2
\end{bmatrix} = \\
\begin{bmatrix}
-\frac{1}{24} & -\frac{h_1-1}{24} & \frac{7}{8} & \frac{h_1-1}{24} & -\frac{1}{24} \\
0 & \frac{7}{8} & \frac{2h_1}{3h_2} & \frac{7}{8} & 0 \\
0 & \frac{(12h_1^2+12h_1-1)(h_1-1)}{h_1 h_2} & -2\frac{12h_1^2-13}{h_2} & \frac{(12h_1^2+12h_1-1)(h_1-1)}{h_1 h_2} & 0 \\
-\frac{1}{2} & 0 & 0 & \frac{7}{2} & 0 \\
0 & \frac{7}{2} & 0 & \frac{7}{2} & 0
\end{bmatrix}
\begin{bmatrix}
f_{i,j-1} \\
f_{i-1,j} \\
f_{i,j} \\
f_{i+1,j} \\
f_{i,j+1}
\end{bmatrix}.
\] (19.30)

19.2.2 Cloud in Cell Mapping

Other interpolation routines can be defined that take into account the actual quantities defined on the grid. These “mesh-based” algorithms are represented in FLASH4 by the Cloud-in-Cell mapping, where the interpolation to/from the particles is defined as a simple linear weighting from nearby grid points. The weights are defined by considering only the region of one “cell” size around each particle location; the proportional volume of the particle “cloud” corresponds to the amount allocated to/from the mesh. The CIC method can be used with both types of particles. When using it with active particles the MapToMesh methods should also be selected. In order to include the CIC method with passive particles, the setup command line option is \texttt{-with-unit=Particles/ParticlesMapping/CIC}. Two additional command line options \texttt{-with-unit=Particles/ParticlesMapping/MapToMesh} and \texttt{-with-unit=Grid/GridParticles/MapToMesh} are necessary when using the active particles. All of these command line options can be replaced by placing the appropriate \texttt{REQUIRES/REQUESTS} directives in the Simulation \texttt{Config} file.

19.3 Using the Particles Unit

The Particles unit encompasses nearly all aspects of Lagrangian particles. The exceptions are input/output the movement of related data structures between different blocks as the particles move from one block to another, and mapping the particle attributes to and from the grid.

Beginning with release of version 4 it is possible to add particles to a simulation during evolution, a new function \texttt{Particles.addNew} has been added to the unit’s API for this purpose. It has been possible to include multiple different types of particles in the same simulation since release FLASH3.1. Particle types must be specified in the \texttt{Config} file of the Simulations unit setup directory for the application, and the syntax is explained in Section 6.6. At configuration time, the setup script parses the \texttt{PARTICLETYPE} specifications in the Config file, and generates an F90 file \texttt{Particles.specifyMethods.F90} that populates a data structure \texttt{gr.ptTypeInfo}. This data structure contains information about the method of initialization and interpolation methods for mapping the particle attributes to and from the grid for each included particle type. Different time integration schemes are applied to active and passive particles. However, in one simulation, all active particles are integrated using the same scheme, regardless of how many active types exists. Similarly, only one passive integration scheme is used. The added complexity of multiple particle types allows different methods to be used for initialization of particles positions and their mapping to and from the grid quantities. Because several different implementations of each type of functionality can co-exist in one simulation, there are no defaults in the \texttt{Particles unit Config} files. These various functionalities are organized into different subunits; a brief description of each subunit is included below and further expanded in subsections in this chapter.

- The \texttt{ParticlesInitialization} subunit distributes a given set of particles through the spatial domain at the simulation startup. Some type of spatial initialization is always required; the functionality is provided by \texttt{Particles.initPositions}. The users of active particles typically have their own custom initialization. The following two implementations of initialization techniques are included in the FLASH4 distribution (they are more likely to used with the passive tracer particles):
Lattice distributes particles regularly along the axes directions throughout a subsection of the physical grid.

WithDensity distributes particles randomly, with particle density being proportional to the grid gas density.

Users have two options for implementing custom initialization methods. The two files involved in the process are: Particles_initPositions and pt_initPositions. The former does some housekeeping such as allowing for inclusion of one of the available methods along with the user specified one, and assigning tags at the end. A user wishing to add one custom method with no constraints on tags etc is advised to implement a custom version of the latter. This approach allows the user to focus the implementation on the placement of particles only. Users desirous of refining the grid based on particles count during initialization should see the setup PoisParticles for an example implementation of the Particles_initPositions routine. If more than one implementation of pt_initPositions is desired in the same simulation then it is necessary to implement each one separately with different names (as we do for tracer particles: pt_initPositionsLattice and pt_initPositionsWithDensity) in their simulation setup directory. In addition, a modified copy of Particles_initPostions, which calls these new routines in the loop over types, must also be placed in the same directory.

- The ParticlesMain subunit contains the various time-integration options for both active and passive particles. A detailed overview of the different schemes is given in Section 19.1.

- The ParticlesMapping subunit controls the mapping of particle properties to and from the grid. FLASH currently supplies the following mapping schemes:

Cloud-in-cell (ParticlesMapping/meshWeighting/CIC), which weights values at nearby grid cells; and

Quadratic (ParticlesMapping/Quadratic), which performs quadratic interpolation.

Some form of mapping must always be included when running a simulation with particles. As mentioned in Section 19.2 the quadratic mapping scheme is only available to map from the grid quantities to the corresponding particle attributes. Since active particles require the same mapping scheme to be used in mapping to and from the mesh, they cannot use the quadratic mapping scheme as currently implemented in FLASH4. The CIC scheme may be used by both the active and passive particles.

For active particles, we use the mapping routines to assign particles’ mass to the particle density grid-based solution variable (PDEN_VAR). This mapping is the initial step in the particle-mesh (PM) technique for evaluating the long range gravitational force between all particles. Here, we use the particle mapping routine Particles_mapToMeshOneBlk to “smear” the particles’ attribute over the cells of a temporary array. The temporary array is an input argument which is passed from the grid mapping routine Grid_mapParticlesToMesh. This encapsulation means that the particle mapping routine is independent of the current state of the grid, and is not tied to a particular Grid implementation. For details about the task of mapping the temporary array values to the cells of the appropriate block(s), please see Section 8.9.2. New schemes can be created that “smear” the particle across many more cells to give a more accurate PDEN_VAR distribution, and thus a higher quality force approximation between particles. Any new scheme should implement a customized version of the pt_assignWeights routine, so that it can be used by the Particles_mapToMeshOneBlk routine during the map.

- The ParticlesForces subunit implements the long and short range forces described in Equation (19.3) in the following directories:

  - longRange collects different long-range force laws (requiring elliptic solvers or the like and dependent upon all other particles);
  - shortRange collects different short-range force laws (directly summed or dependent upon nearest neighbors only).
Currently, only one long-range force law (gravitation) with one force method (particle-mesh) is included with FLASH. Long-range force laws are contained in the Particles/ParticlesForces/longRange, which requires that the Gravity unit be included in the code. In the current release, no shortRange implementation of ParticlesForces is supplied with FLASH. However, note that the sink particle implementation described below in Section 19.4 includes directly computed particle–particle forces.

After particles are moved during time integration or by forces, they may end up on other blocks within or without the current processor. The redistribution of particles among processors is handled by the GridParticles subunit, as the algorithms required vary considerably between the grid implementations. The boundary conditions are also implemented by the GridParticles unit. See Section 8.9 for more details of these redistribution algorithms. The user should include the option -with-unit=Grid/GridParticles on the setup line, or REQUIRES Grid/GridParticles in the Config file.

In addition, the input-output routines for the Particles unit are contained in a subunit IOParticles. Particles are written to the main checkpoint files. If the user desires, a separate output file can be created which contains only the particle information. See Section 19.3.3 below as well as Section 9.2.3 for more details. The user should include the option -with-unit=IO/IOParticles on the setup line, or REQUIRES IO/IOParticles in the Config file.

In FLASH4, the initial particle positions can be used to construct an appropriately refined grid, i.e. more refined in places where there is a clustering of particles. To use this feature the flash.par file must include: refine_on_particle_count=.true. and max_particles_per_blk=[some value]. Please be aware that FLASH will abort if the criterion is too demanding. To overcome the abort, specify a less demanding criterion, or increase the value of lrefine_max.

19.3.1 Particles Runtime Parameters

There are several general runtime parameters applicable to the Particles unit, which affect every implementation. The variable useParticles obviously must be set equal to .true. to utilize the Particles unit. The time stepping is controlled with pt_dtFactor; a value less than one ensures that particles will not step farther than one entire cell in any given time interval. The Lattice initialization routines have additional parameters. The number of evenly spaced particles is controlled in each direction by pt_numX and similar variables in Y and Z. The physical range of initialization is controlled by pt_initialXMin and the like. Finally, note that the output of particle properties to special particle files is controlled by runtime parameters found in the IO unit. See Section 9.2.3 for more details.

19.3.2 Particle Attributes

By default, particles are defined to have eight real properties or attributes: 3 positions in x,y,z; 3 velocities in x,y,z; the current block identification number; and a tag which uniquely identifies the particle. Additional properties can be defined for each particle. For example, active particles usually have the additional properties of mass and acceleration (needed for the integration routines, see Table Table 19.1). Depending upon the simulation, the user can define particle properties in a manner similar to that used for mesh-based solution variables. To define a particle attribute, add to a Config file a line of the form

    PARTICLEPROP property-name

For attributes that are meant to merely sample and record the state of certain mesh variables along trajectories, FLASH can automatically invoke interpolation (or, in general, some map method) to generate attribute values from the appropriate grid quantities. (For passive tracer particles, these are typically the only attributes beyond the default set of eight mentioned above.) The routine Particles_updateAttributes is invoked by FLASH at appropriate times to effect this mapping, namely before writing particle data to checkpoint and particle plot files. To direct the default implementation of Particles_updateAttributes to act as desired for tracer attributes, the user must define the association of the particle attribute with the appropriate mesh variable by including the following line in the Config file:

    PARTICLEMAP TO property-name FROM VARIABLE variable-name
These particle attributes are carried along in the simulation and output in the checkpoint files. At runtime, the user can specify the attributes to output through runtime parameters `particle_attribute_1`, `particle_attribute_2`, etc. These specified attributes are collected in an array by the `Particles_init` routine. This array in turn is used by `Particles_updateAttributes` to calculate the values of the specified attributes from the corresponding mesh quantities before they are output.

### 19.3.3 Particle I/O

Particle data are written to and read from checkpoint files by the I/O modules (Section 9.1). For more information on the format of particle data written to output files, see Section 9.9.1 and Section 9.9.2.

Particle data can also be written out to the `flash.dat` file. The user should include a local copy of `IO_writeIntegralQuantities` in their Simulation directory. The Orbit test problem supplies an example `IO_writeIntegralQuantities` routine that is useful for writing individual particle trajectories to disk at every timestep.

There is also a utility routine `Particles_dump` which can be used to dump particle output to a plain text file. An example of usage can be found in `Particles_unitTest`. Output from this routine can be read using the fidlr routine `particles_dump.pro`.

### 19.3.4 Unit Tests

The unit tests provided for `Particles` exercise the `Particles_advance` methods for tracer particles. Tests under `Simulation/SimulationMain/unitTest/ParticlesAdvance` can be used to examine and compare convergence behavior of various time integration schemes. The tests compare numerical and analytic solutions for a problem (with a given velocity field) where analytic solutions can be computed.

Currently only one `ParticlesAdvance` test is provided. It is designed to be easily modified by replacing a few source files that contain implementations of the equation and the analytic solution. The use the test, configure it with a command like

```
./setup -auto -1d unitTest/ParticlesAdvance/HomologousPassive \ 
  -unit=Particles/ParticlesMain/passive/EstiMidpoint2
```

and replace `EstiMidpoint2` with one of the other available methods (or omit the option to get the default method), see Section 19.1.3. Add other options as desired.

For `unitTest/ParticlesAdvance/HomologousPassive`,

```
./setup -auto -1d unitTest/ParticlesAdvance/HomologousPassive +ug -nxb=80
```

is recommended to get started.

When varying the test, the following runtime parameters defined for `Simulation/SimulationMain/unitTest/ParticlesAdvance` will probably need to be adjusted:

**PARAMETER** `sim_schemeOrder` `INTEGER 2` — The order of the integration scheme. This should probably always be either 1 or 2.

**PARAMETER** `sim_maxTolCoeff0` `REAL 1.0e-8` — Zero-th order error coefficient $C_0$, used for convergence criterion if `sim_schemeOrder` = 0.

**PARAMETER** `sim_maxTolCoeff1` `REAL 0.0001` — First order error coefficient $C_1$, used for convergence criterion if `sim_schemeOrder` = 1.

**PARAMETER** `sim_maxTolCoeff2` `REAL 0.01` — Second order error coefficient $C_2$, used for convergence criterion if `sim_schemeOrder` = 2.

A test for order $k$ is considered successful if the following criterion is satisfied:

$$\text{maxError} \leq C_k \times \text{maxActualDt}^k,$$

where `maxError` is the maximum absolute error between numerical and analytic solution for any particle that was encountered during a simulation run, and `maxActualDt` is the maximum time step $\Delta t$ used in the run.
The appropriate runtime parameters of various units, in particular Driver, Particles, and Grid, should be used to control the desired simulation run. In particular, it is recommended to vary \texttt{dtmax} by several orders of magnitude (over a range where it directly determines \texttt{maxActualDt}) for a given test in order to examine convergence behavior.

19.4 Sink Particles

19.4.1 Basics of Sink Particles

Sink particles are required in collapse simulations to model dense core, star, or black hole formation and accretion. Using sink particles solves two main problems in collapse calculations:

1. The physical length scale associated with the collapse, the Jeans length,

   \[ \lambda_J = \sqrt{\frac{\pi c_s^2}{G \rho}} \]  

   decreases with increasing gas density \( \rho \) (here, \( G \) is the gravitational constant and \( c_s \) the sound speed).

To avoid artificial fragmentation, the Jeans length must be resolved with at least 4 grid cells (Truelove et al. 1997, ApJ 489, L179). More recent calculations show that a minimum of 32 grid cells is required to resolve the kinetic energy in rotational motions inside the Jeans volume and to resolve minimum turbulent MHD dynamo action (Sur et al. 2010, ApJ 721, L134; Federrath et al. 2011, ApJ 731, 62). The FLASH sink particle module automatically refines the AMR grid based on the Jeans length (controlled by runtime parameters \texttt{refineOnJeansLength}, \texttt{jeans_ncells_ref}, and \texttt{jeans_ncells_deref}). However, once the highest level of the AMR hierarchy is reached, sink particles must be created to avoid artificial fragmentation when the Jeans length drops further during the collapse.

2. The typical timescale for collapsing objects is the freefall time, \( t_{ff} = \sqrt{3\pi/(32G\rho)} \). With increasing gas density, this timescale becomes shorter and shorter, which means that the code will literally grind to a halt during runaway collapse, as time steps become shorter and shorter. The first object going into freefall collapse thus determines the time step of the whole calculation, such that following the formation of a star cluster would be impossible. To avoid these problems, the gas in regions that are in a state of runaway collapse are replaced by sink particles.

19.4.2 Using the Sink Particle Unit

To include sink particles, specify \texttt{REQUIRES} Particles/ParticlesMain/active/Sink in the simulation \texttt{Config} file. This automatically includes all required subunits for the sink particles. Currently, sink particles require a Paramesh 4 Grid implementation and Poisson gravity. Currently, various subunits and implementation directories of the Particles unit and are also included automatically. Table 19.6 lists all runtime parameters of the sink particle module. The default values of these parameters should be ok in most simulations, except \texttt{sink_density_thresh}, \texttt{sink_accretion_radius}, and \texttt{sink_softening_radius}. Those three parameters have to be adopted to the resolution and physics of a given simulation.

First, the sink particle accretion radius, \( r_{acc} \) (\texttt{sink_accretion_radius}) should be calculated, based on the minimum cell size \( \Delta x \) for a given simulation. The recommended value is \( r_{acc} = 2.5\Delta x \). If sink particles are included, the code will print a message to the standard output, stating the number of cells that a chosen sink accretion radius corresponds to. The recommended value for the sink particle softening radius (\texttt{sink_softening_radius}) is to set it equal to the accretion radius.

In order to avoid artificial fragmentation, the gas density on the AMR grid must not exceed a critical density \( \rho_{thresh} \) in regions of gravitational collapse (Truelove et al. 1997). This density is related to the smallest resolvable Jeans length \( \lambda_J \) on the highest level of the AMR hierarchy. The density threshold \( \rho_{thresh} \) (\texttt{sink_density_thresh}) is obtained by solving the definition of the Jeans length, Equation (19.31) for \( \rho \),

\[ \rho_{thresh} = \frac{\pi c_s^2}{4G r_{acc}^2} \]

(19.32)
This equation relates the sink particle accretion radius to the sink particle density threshold and depends on the thermodynamics (sound speed) of a given simulation. Since the Jeans length should be resolved with at least 4 grid cells (Truelove et al. 1997), the sink particle accretion radius $r_{\text{acc}}$ must not be smaller than 2 grid cells. The accretion radius is thus determined by the smallest linear cell size $\Delta x$ of the AMR grid. As recommended above, setting $r_{\text{acc}} \simeq 2.5 \Delta x$ satisfies the Truelove criterion, because the Jeans length $\lambda_J = 2r_{\text{acc}}$ is thus resolved with 5 grid cells on the top level of the AMR hierarchy.

Please cite Federrath et al. (2010, ApJ 713, 269) if you are using this unit.

![Table 19.6: Runtime parameters for the sink particle module.](image)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useSinkParticles</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>switch sinks on/off</td>
</tr>
<tr>
<td>sink_density_thresh</td>
<td>REAL</td>
<td>1.0e-14</td>
<td>density threshold for sink creation and accretion</td>
</tr>
<tr>
<td>sink_accretion_radius</td>
<td>REAL</td>
<td>1.0e14</td>
<td>creation and accretion radius</td>
</tr>
<tr>
<td>sink_softening_radius</td>
<td>REAL</td>
<td>1.0e14</td>
<td>gravitational softening radius</td>
</tr>
<tr>
<td>sink_softening_type_gas</td>
<td>STRING</td>
<td>&quot;linear&quot;</td>
<td>sink–gas softening type (options: &quot;linear&quot;, &quot;spline&quot;)</td>
</tr>
<tr>
<td>sink_softening_type_sinks</td>
<td>STRING</td>
<td>&quot;spline&quot;</td>
<td>sink–sink softening type (options: &quot;linear&quot;, &quot;spline&quot;)</td>
</tr>
<tr>
<td>sink_integrator</td>
<td>STRING</td>
<td>&quot;leapfrog&quot;</td>
<td>sink particle time integrator (options: &quot;euler&quot;, &quot;leapfrog&quot;, &quot;leapfrog_cosmo&quot;)</td>
</tr>
<tr>
<td>sink_dt_factor</td>
<td>REAL</td>
<td>0.5</td>
<td>time step safety factor ($\leq 0.5$)</td>
</tr>
<tr>
<td>sink_subdt_factor</td>
<td>REAL</td>
<td>0.01</td>
<td>time step safety factor for sink–sink subcycling ($\leq 0.5$)</td>
</tr>
<tr>
<td>sink_convergingFlowCheck</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>creation check for converging gas flow</td>
</tr>
<tr>
<td>sink_potentialMinCheck</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>creation check for gravitational potential minimum</td>
</tr>
<tr>
<td>sink_jeansCheck</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>creation check for Jeans instability</td>
</tr>
<tr>
<td>sink_negativeEtotCheck</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>creation check for gravitationally bound gas</td>
</tr>
<tr>
<td>sink_GasAccretionChecks</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>check for bound and converging state before gas accretion</td>
</tr>
<tr>
<td>sink_merging</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>switch for sink particle merging</td>
</tr>
<tr>
<td>pt_maxSinksPerProc</td>
<td>INTEGER</td>
<td>100</td>
<td>number of sinks per processor</td>
</tr>
<tr>
<td>refineOnSinkParticles</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>sinks must be on highest AMR level</td>
</tr>
<tr>
<td>refineOnJeansLength</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>switch for refinement on Jeans length</td>
</tr>
<tr>
<td>jeans_ncells_ref</td>
<td>REAL</td>
<td>32.0</td>
<td>number of cells for Jeans length refinement</td>
</tr>
<tr>
<td>jeans_ncells_deref</td>
<td>REAL</td>
<td>64.0</td>
<td>number of cells for Jeans length de-refinement</td>
</tr>
</tbody>
</table>

### 19.4.3 The Sink Particle Method

For technical details and tests of the FLASH sink particle unit, see Federrath et al. (2010, ApJ 713, 269) and Federrath et al. (2011, IAUS 270, 425). We only summarize here the most important aspects of the implementation.
Before a sink particle is created, we define a spherical control volume with the accretion radius \( r_{\text{acc}} \) around each cell that exceeds the resolution-dependent density threshold, \( \rho_{\text{thresh}} \) given by Equation (19.32), and check whether the gas in this control volume

- is on the highest level of refinement,
- is converging (\( \nabla \cdot \mathbf{v} < 0 \); i.e., has negative radial velocity),
- has a central gravitational potential minimum,
- is gravitationally bound (\( |E_{\text{grav}}| > E_{\text{int}} + E_{\text{kin}} + E_{\text{mag}} \)),
- is gravitationally unstable (Jeans-unstable),
- and is not within the accretion radius of an existing sink particle.

These checks are designed to avoid spurious sink particle formation and to trace only truly collapsing and star-forming gas. As soon as a sink particle is created, it can gain further mass by accretion from the AMR grid, if this gas is inside the sink particle accretion radius and is bound and collapsing towards the sink particle. If the accretion radii of multiple sink particles overlap, the gas of a given cell is accreted onto the sink particle to which the gas is most strongly bound.

We take all contributions to the gravitational interactions between the gas on the grid and the sink particles into account. Those interactions are

1. GAS–GAS (handled by either the multigrid or the tree Poisson solver)
2. GAS–SINKS (computed by direct summation over all particles and grid cells)
3. SINKS–GAS (computed by direct summation over all particles and grid cells)
4. SINKS–SINKS (computed by direct \(N\)-body summation over all sink particles).

We note that all interactions including sink particles (GAS–SINKS, SINKS–GAS, SINKS–SINKS) do not require the Poisson solver or any particle–grid mapping procedure. Since these are all computed by direct summation, computational costs can become quite expensive once the number of sink particles reaches thousands and more, but is significantly more accurate than mapping procedures. In fact, a previous implementation used interpolation of sink particle masses back onto the grid and employing the Poisson solver to compute all interactions. This resulted in very smooth gravitational interactions that made it impossible to follow close orbits and highly eccentric encounters of multiple (two and more) sink particles, in particular for the sink–sink interactions. Thus, the current implementation of sink particles is designed to follow a maximum of a few thousand particles. Linear and spline softening of the gravitational forces for extremely close encounters of multiple sink particles are implemented. By default, linear softening is used for gas–sinks and sinks–gas interactions (sink softening type gas), motivated by an almost uniform gas density inside the sink particle radius. Spline softening is used for sink–sink interactions (sink softening type sinks), as it is more suitable to follow \(N\)-body dynamics. Softening is only applied inside the sink softening radius (sink softening radius). A second-order accurate leapfrog integrator is used to advance the sink particles. For cosmological simulations, a cosmological version of the leapfrog integrator is available (see sink integrator). The sink particles are fully integrated into the MPI parallelization of the FLASH code. Any split or unsplit solver for hydro or MHD (in which case the bound state check includes the contribution of the magnetic energy) is supported.

### 19.4.4 Sink Particle Unit Test

To invoke the sink particle unit test, use \( ./\text{setup unitTest/SinkMomTest} \ -\text{auto} \ -3d \). This initializes a momentum conservation test. An initially uniform cloud with radius 0.025 pc and density \( 10^{-18} \text{g cm}^{-3} \) at rest starts collapsing and forms a sink particle. An initial sink particle with 0.1 \( M_{\odot} \) and initial momentum \( p_y = 4 \times 10^{36} \text{g cm s}^{-1} \) in positive \( y \)-direction is also present. The purpose of this test is to see how well the total \( x \)-momentum \( p_x \) and the total \( y \)-momentum \( p_y \) are conserved. This test is particularly suitable, because it involves gas collapse, sink particle creation, accretion, more than one sink
particle, and thus all gravitational interactions (gas–gas, gas–sinks, sinks–gas, sinks–sinks), as well as close eccentric orbits of the two sink particles. Figure 19.3 shows $p_x$ and $p_y$ as a function of time for both sink particles and gas separately. The symmetry between sink and gas momenta shows that momentum is well conserved for several orbits. Figure 19.3 can be reproduced with the IDL and Python tools in /source/Simulation/SimulationMain/unitTest/SinkMomTest/utils/ by running the IDL script plot_mom_sinks.pro.
Chapter 20

Cosmology Unit

Figure 20.1: The Cosmology unit tree.

The Cosmology unit solves the Friedmann equation for the scale factor in an expanding universe, applies a cosmological redshift to the hydrodynamical quantities, and supplies library functions for various routine cosmological calculations needed by the rest of the code for initializing, performing, and analyzing cosmological simulations.

20.1 Algorithms and Equations

The Cosmology unit makes several assumptions about the interpretation of physical quantities that enable any hydrodynamics or materials units written for a non-expanding universe to work unmodified in a cosmological context. All calculations are assumed to take place in comoving coordinates \( x = r/a \), where \( r \) is a proper position vector and \( a(t) \) is the time-dependent cosmological scale factor. The present epoch is defined to correspond to \( a = 1 \); in the following discussion we use \( t = t_0 \) to refer to the age of the Universe at the present epoch. The gas velocity \( \mathbf{v} \) is taken to be the comoving peculiar velocity \( \mathbf{x} \). The comoving gas
density, pressure, temperature, and internal energy are defined to be

\[ \rho = a^3 \tilde{\rho} \]
\[ p = a \tilde{p} \]
\[ T = \frac{T}{a^2} \]
\[ \rho \epsilon = a \tilde{\rho} \tilde{\epsilon} . \]  

(20.1)

The quantities marked with a tilde, such as \( \tilde{\rho} \), are the corresponding “proper” or physical quantities. Note that, in terms of comoving quantities, the equation of state has the same form as for the proper quantities in noncomoving coordinates. For example, the perfect-gas equation of state is

\[ \rho \epsilon = \frac{p}{\gamma - 1} = \frac{\rho k T}{(\gamma - 1) \mu} . \]  

(20.2)

With these definitions, the Euler equations of hydrodynamics can be written in the form

\[ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \]  

(20.3)

\[ \frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla p + 2 \frac{\dot{a}}{a} \rho \mathbf{v} + \rho \mathbf{v} \mathbf{\nabla} \phi = 0 \]  

(20.4)

\[ \frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p)v] + \frac{\dot{a}}{a}(3\gamma - 1)\rho \epsilon + 2\rho v^2] + \rho \mathbf{v} \cdot \nabla \phi = 0 \]  

(20.5)

\[ \frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + p)v] - \mathbf{v} \cdot \nabla p + \frac{\dot{a}}{a}(3\gamma - 1)\rho \epsilon = 0 . \]  

(20.6)

Here \( E \) is the specific total energy, \( \epsilon = \frac{1}{2} v^2 \), and \( \gamma \) is the effective ratio of specific heats. The Cosmology unit applies the terms involving \( \dot{a} \) via the Cosmology\_redshiftHydro routine.

The comoving potential \( \phi \) in the above equations is the solution to the Poisson equation in the form

\[ \nabla^2 \phi = \frac{4\pi G}{a^3} (\rho - \tilde{\rho}) , \]  

(20.7)

where \( \tilde{\rho} \) is the comoving mean matter density. Note that, because of the presence of \( a \) in (20.7), the gravity units must explicitly divide their source terms by \( a^3 \).

Units like the Gravity unit, which require the scale factor or the redshift \( z (a = (1 + z)^{-1}) \), can obtain the redshift via Cosmology\_getRedshift, and use the previous relation to obtain the scaling factor. The time represented by a cosmological redshift can be obtained by a call to Cosmology\_redshiftToTime and passing it a cosmological redshift. Note also that if a collisionless matter component (e.g., particles) is also present, its density must be added to the gas density on the right-hand side of (20.7). Accounting for particle masses in density is handled by the Gravity unit.

The comoving mean matter density is defined in terms of the critical density \( \rho_{\text{crit}} \) by

\[ \tilde{\rho} = \Omega_m \rho_{\text{crit}} \]
\[ \rho_{\text{crit}} = \frac{3H^2}{8\pi G} . \]  

(20.8)

The Hubble parameter \( H(t) \) [to be distinguished from the Hubble “constant” \( H_0 \equiv H(t_0) \)] is given by the Friedman equation

\[ H^2(t) \equiv \left( \frac{\dot{a}}{a} \right)^2 = H_0^2 \left( \frac{\Omega_m}{a^3} + \frac{\Omega_r}{a^4} + \Omega_\Lambda - \frac{\Omega_c}{a^2} \right) . \]  

(20.9)

Here \( \Omega_m, \Omega_r, \) and \( \Omega_\Lambda \) are the present-day densities, respectively, of matter, radiation, and cosmological constant, divided by \( \rho_{\text{crit}} \). The contribution of the overall spatial curvature of the universe is given by

\[ \Omega_c \equiv \Omega_m + \Omega_r + \Omega_\Lambda - 1 . \]  

(20.10)
The \texttt{Cosmology\_solveFriedmannEqn} routine numerically solves the Friedmann equation to obtain the scale factor and its rate of change as functions of time. In principle, any good ODE integrator can be used; the \texttt{csm\_integrateFriedmann} subroutine uses a fourth-order Runge-Kutta method to integrate the Friedmann equation under the assumption that $\Omega_r = 0$. Subunits can also use analytic solutions where appropriate.

Redshift terms for particles are handled separately by the appropriate time integration subunits of the \texttt{Particles} unit. For an example, see the \texttt{LeapfrogCosmo} implementation of the \texttt{ParticlesMain} subunit in Section 19.1.1.

### 20.2 Using the Cosmology unit

To include cosmological expansion in your FLASH executable, include the line

```
REQUESTS physics/Cosmology/
```

in your setup’s \texttt{Config} file. At present the Cosmology unit in FLASH4 is built around the \texttt{MatterLambdaKernel}. This kernel assumes the contribution of radiation to be negligible in comparison with those of matter and the cosmological constant.

The runtime parameters available with the \texttt{Cosmology} unit are described in Table 20.1. Note that the total effective mass density is not explicitly specified but is inferred from the sum of the \texttt{OmegaMatter}, \texttt{OmegaRadiation}, and \texttt{CosmologicalConstant} parameters. The \texttt{MaxScaleChange} parameter sets the maximum allowed fractional change in the scale factor $a$ during a single timestep. This behavior is enforced by the \texttt{Cosmology\_computeDt} routine. The default value is set to the system’s \texttt{HUGE} value for a double precision real floating point value to avoid interfering with non-cosmological simulations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useCosmology</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>True if cosmology is to be used in this simulation</td>
</tr>
<tr>
<td>OmegaMatter</td>
<td>REAL</td>
<td>0.3</td>
<td>Ratio of total mass density to critical density at the present epoch ($\Omega_m$)</td>
</tr>
<tr>
<td>OmegaBaryon</td>
<td>REAL</td>
<td>0.05</td>
<td>Ratio of baryonic (gas) mass density to critical density at the present epoch; must be $\leq$ \texttt{OmegaMatter} ($\Omega_b$)</td>
</tr>
<tr>
<td>CosmologicalConstant</td>
<td>REAL</td>
<td>0.7</td>
<td>Ratio of the mass density equivalent in the cosmological constant to the critical density at the present epoch ($\Omega_\Lambda$)</td>
</tr>
<tr>
<td>OmegaRadiation</td>
<td>REAL</td>
<td>$5 \times 10^{-5}$</td>
<td>Ratio of the mass density equivalent in radiation to the critical density at the present epoch ($\Omega_r$)</td>
</tr>
<tr>
<td>HubbleConstant</td>
<td>REAL</td>
<td>$2.1065 \times 10^{-18}$</td>
<td>Value of the Hubble constant $H_0$ in sec$^{-1}$</td>
</tr>
<tr>
<td>MaxScaleChange</td>
<td>REAL</td>
<td>\texttt{HUGE}(1.)</td>
<td>Maximum permitted fractional change in the scale factor during each timestep</td>
</tr>
</tbody>
</table>

The \texttt{MatterLambdaKernel} supplies a number of functions and routines that are helpful in initializing, performing, and analyzing cosmological simulations. They should be accessed through the wrapper functions shown below.

- \texttt{Cosmology\_cdmPowerSpectrum}
  - Return the present-day cold dark matter power spectrum as a function of a given wavenumber. The \texttt{MatterLambdaKernel} provides a fit to this power spectrum from Bardeen et al. (1986), which assumes baryons do not make a significant contribution to the mass density. Other fits are available; see e.g., Hu and Sugiyama (1996) or Bunn and White (1997).
CHAPTER 20. COSMOLOGY UNIT

- **Cosmology.computeVariance**
  Given an array of comoving length scales and a processed power spectrum, compute the linear variance \((\delta M/M)^2\) at the present epoch. A top-hat filter is applied in Fourier-space as a smoothing kernel.

- **Cosmology.computeDeltaCrit**
  This subroutine computes the linear overdensity at turnaround in the spherical collapse model. For more information, see the appendix of Lacey and Cole (1993).

- **Cosmology.redshiftToTime**
  Compute the age of the Universe corresponding to a cosmological redshift.

- **Cosmology.massToLength**
  Given a mass scale, return the corresponding comoving diameter of a sphere containing the given amount of mass.

20.3 Unit Test

FLASH provides a unit test for checking the basic functionality of the Cosmology module. It tests the unit’s generated cosmological scaling factor, cosmological redshift, and the time calculated from that redshift against an analytical solution of these quantities.

The test is run with the following parameters:

$$Omega_{Matter} = 1.0$$  
$$Omega_{Lambda} = 0.0$$  
$$Omega_{Baryon} = 1.0$$  
$$Omega_{Radiation} = 0.0$$  
$$HubbleConstant = 1.62038 \times 10^{-18} \text{sec}^{-1} (50 \text{ km/s/Mpc})$$

The Cosmological scaling factor is related to time by the equation:

$$a(t) = \left( \frac{t}{t_0} \right)^{2/3}$$

where $$t_0 = \frac{2}{3H_0}$$, $$H_0$$ is the HubbleConstant and is related to the cosmological redshift by the equation $$z(t) = \frac{1}{a(t)} - 1$$. The change in time is a uniform step, and by comparing the analytical and code results at time $$t$$, we can see if the Friedmann equations are correctly integrated by the Cosmology unit, and that the results fall within a tolerance set in Cosmology_unitTest.
Chapter 21

Material Properties Units

Figure 21.1: The materialProperties unit with Opacity, MagneticResistivity and Viscosity subunits.

Figure 21.2: The materialProperties unit with MassDiffusivity and Conductivity subunits.
In this release, FLASH’s implementation of the material properties units is minimal. For Heat Conductivity and Viscosity, we provide implementations for effects with constant coefficients; these can be used as models for implementing effects that follow other laws. For MassDiffusivity, only no-operation stubs are provided. A routine that calculates constant magnetic resistivity and viscosity is provided in the MagneticResistivity unit and can be used in non-ideal magnetohydrodynamics simulations. Several add-on capabilities are being made available to the users from the Download Page on Flash Website.

21.1 Thermal Conductivity

The Conductivity unit implements a prescription for computing thermal conductivity coefficients used by the Hydro PPM, the unsplit hydro and MHD solvers. The FLASH4 release provides three implementations:

- **Constant** for heat conduction with a constant isochoric conductivity;
- **Constant-diff** for heat conduction with a constant coefficient of diffusion.
- **SpitzerHighZ** which is used for electron thermal conduction. This is the most useful option for realistic HEDP simulations. See Section 25.7.5 for an example of how the SpitzerHighZ implementation is used in a realistic simulation. The Spitzer conductivity implemented here is shown in (21.1). It is consistent with the value given in (Atzeni, 2004).

\[
K_{ele} = \left(\frac{8}{\pi}\right)^{3/2} \frac{k_B^{7/2} e^4}{\pi m_{ele}} \left(\frac{1}{1 + 3.3/\bar{z}}\right) \frac{T_{ele}^{5/2}}{\bar{z} \ln \Lambda_{ei}}
\]  

(21.1)

where:
- \(K_{ele}\) is the electron conductivity
- \(k_B\) is the Boltzmann constant
- \(e\) is the electron charge
- \(m_{ele}\) is the mass of an electron
- \(\bar{z}\) is the average ionization as computed by the EOS
- \(T_{ele}\) is the electron temperature
- \(\ln \Lambda_{ei}\) is the Coulomb logarithm associated with ion/electron collisions and is shown in (16.215).

To use thermal conductivity in a FLASH4 simulation, the runtime parameter `useConductivity` must be set to `.true.`

21.2 Magnetic Resistivity

The magnetic resistivity unit `source/physics/materialProperties/MagneticResistivity` provides routines that computes magnetic resistivity \(\eta\) (and thus viscosity \(\nu_m\)) for a mixture of fully ionized gases used by the MHD solvers. The relationship between magnetic resistivity and viscosity is \(\nu_m = \frac{\eta}{4\pi}\) in CGS and \(\nu_m = \frac{1}{m} \eta\) in SI. The default top level routines return zero values for resistivity (a stub functionality). Specific routines for constant and variable resistivity are provided in the low level subdirectories `/MagneticResistivityMain/Constant` and `/MagneticResistivityMain/SpitzerHighZ`. By default, all routines return results in non-dimensional units (hence without \(4\pi\) or \(\mu_0\) coefficients). However they provide an option to return results either in CGS or SI unit.
21.2.1 Constant resistivity

This subunit returns constant magnetic resistivity. The unit declares a runtime parameter, resistivity, that is the constant resistivity. The default value is zero. The magnetic resistivity routine reads in resistivity ($\eta$) and returns it to the calling routine with proper scalings depending on unit system. For example, $c^2/4\pi\eta$ is returned in CGS unit, $1/\mu_0\eta$ in SI, and simply $\eta$ in non-dimensional unit.

FLASH Transition

In previous implementations, there used to be two runtime parameters: magnetic resistivity and magnetic viscosity. They respectively refer $\eta$ and $c^2/4\pi\eta$ in CGS (or $1/\mu_0\eta$ in SI, where $\mu_0 = 4\pi \times 10^{-7}$ henry/meter). What it was done in the old way was to initialize magnetic viscosity (e.g., $c^2/4\pi\eta$) using the magnetic resistivity, $\eta$. As of FLASH3.1, such distinctions between the magnetic resistivity and magnetic viscosity has been removed and we only use magnetic resistivity with proper scalings depending on unit system.

21.2.2 Spitzer HighZ resistivity

This subunit returns the Spitzer resistivity for the induction equation and is the most useful option for HEDP simulations. The implementation follows the prescription of Braginskii (1965), defining a perpendicular and parallel magnetic resistivity, with respect to the field line, as

$$ \eta_{\text{perp}} = \frac{m_{\text{ele}}}{e^2 n_{\text{ele}} \tau_{\text{ele}}}, \quad (21.2) $$

$$ \eta_{\text{par}} = \eta_{\text{perp}}/1.96. \quad (21.3) $$

Here we denote with $m_{\text{ele}}$, $e$, $n_{\text{ele}}$, $\tau_{\text{ele}}$ the mass, charge, number density and the collision time of electrons, respectively. In its current form, the implementation /MagneticResistivityMain/SpitzerHighZ/MagneticResistivity returns only the parallel component when calculating the resistive fluxes. This can be modified by calling the /MagneticResistivityMain/SpitzerHighZ/MagneticResistivity_fullState implementation and requesting the optional perpendicular component. To activate the Spitzer HighZ resistivity implementation, simply add the appropriate path in the Config file of your simulation directory.

21.3 Viscosity

The Viscosity unit implements a prescription for computing viscosity coefficients used by the Hydro PPM, the unsplit hydro and MHD solvers. In this release the unit provides support for either constant dynamic viscosity or constant kinematic viscosity, where the choice between the two is made with the runtime parameter visc_whichCoefficientIsConst.

To use viscosity in a FLASH4 simulation, the runtime parameter useViscosity must be set to .true.

21.4 Opacity

The Opacity unit, which resides in physics/materialProperties/Opacity, exists to provide opacities for multigroup radiation diffusion to the RadTrans unit. Thus, the Opacity unit does not have an API for providing continuous opacities. The group structure is specified using parameters which are described in Chapter 22. Opacities are accessed by the RadTrans unit using the Opacity subroutine.

```
call Opacity(soln, ngrp, opacityAbsorption, opacityEmission, opacityTransport)
```
The first argument is an input and provides the complete state within a cell as provided by the routine \texttt{Grid\_getBlPtr}. The second input argument is the group number. The last three arguments are outputs which return the absorption, emission, and transport opacities in units of $1/cm$ using the conditions provided by the \texttt{soln} argument. See Chapter 22 for a description of these three opacities. There are currently two \texttt{Opacity} implementations: Constant and Multispecies which are described below.

### 21.4.1 Constant Implementation

The Constant implementation is very simple and is useful for running test problems. It sets the absorption, emission, and transport opacities to constant values that are specified using the runtime parameters \texttt{op\_absorbConst}, \texttt{op\_emissConst}, and \texttt{op\_transConst}. Users can provide custom implementations of the Constant \texttt{Opacity} subroutine to test different formulas for opacities in FLASH.

The Constant opacity implementation can be included with the following setup option:

\texttt{-with-unit=physics/materialProperties/Opacity/OpacityMain/Constant}

### 21.4.2 Multispecies Implementation

In general, the opacity is a strong function of the material (or species in FLASH parlance), frequency (or group number), and state of the fluid. The Multispecies \texttt{Opacity} unit implementation provides the flexibility to specify different opacities to use as a function of species and opacity type (either absorption, emission, or transport). It can be included in the simulation using the following setup option:

\texttt{-with-unit=physics/materialProperties/Opacity/OpacityMain/Multispecies}

The opacities for each species are averaged together based on the relative number densities to produce an average opacity of each type for a given cell. For each species and type, the Multispecies implementation allows the user must specify an opacity model to use. There are currently three commonly used opacity models in FLASH:

- **constant**: This model simply returns a constant value for the opacity in units of $1/cm$
- **constcm2g**: This model also returns a constant opacity but uses units of $cm^2/g$
- **tabulated**: This model allows the user to specify the names of files which store opacity tables. These tables store opacities as functions electron temperature and density

The first two opacity models, constant and constcm2g, are fairly self-explanatory. The tabulated opacity model is useful for modeling realistic opacities in a general way. Each table is tabulates an opacity at $N_D$ discrete density points and $N_T$ discreet electron temperature points. The table for each species and opacity type can have different a temperature/density grid. However, each table must use the same energy group structure which is consistent with the group structure used by the \texttt{RadTrans} unit.

Extraction (interpolation) of opacities from the stored tables is currently done using the bilinear form. For a temperature/density pair $(t, d)$ inside the grid, the code determines the quadrant corners $T_1 \leq t \leq T_2$ and $D_1 \leq d \leq D_2$ and the corresponding four opacities $\kappa_{xy} = \kappa(T_x, D_y); x, y = 1, 2$ and performs the bilinear interpolation:

\[
\begin{align*}
\tau_1 &= \frac{T_2 - t}{T_2 - T_1} \\
\tau_2 &= \frac{t - T_1}{T_2 - T_1} \\
\delta_1 &= \frac{D_2 - d}{D_2 - D_1} \\
\delta_2 &= \frac{d - D_1}{D_2 - D_1} \\
\kappa(t, d) &= \sum_{i=1}^{2} \sum_{j=1}^{2} \tau_i \delta_j \kappa_{ij}. 
\end{align*}
\]
In case the target \((t, d)\) lays outside the grid boundaries, the corresponding boundary value(s) is(are) taken.
This is a temporary solution and will be supplemented with options for calculating more accurate opacities,
especially for lower cell temperatures. Two possible options are provided for performing the interpolation:
i) on the original tabulated opacities or ii) on the logarithmically (base 10) transformed tabulated opacities.

Currently, the tables must be stored in either the IONMIX or IONMIX4 formats. The exact specification
of this format can be found in Section 21.4.3. The IONMIX4 format is the most general since it allows for
arbitrary temperature/density points. Users who wish to use their own tabulated opacities may convert
their files into the IONMIX4 format for use in FLASH. As Section 21.4.3 describes, IONMIX4 files contain
group opacities in units of \(cm^2/g\). Each file contains three different group opacities: a Planck Absorption,
a Planck Emission, and a Rosseland opacity. The user must specify which of these opacities to use for each
species/opacity type in FLASH.

### 21.4.2.1 Runtime Parameters for the Multispecies Opacity

Use of the Multispecies opacity implementation requires the Multispecies unit. At least one species must
be defined in the simulation to use the Multispecies opacity. Furthermore, the species must all be specified
using the species setup variable and cannot be defined using the SPECIES keyword in a Config
file. Please see Section 11.4 for more details.

Once the species are specified, the Multispecies opacity implementation will automatically create a set of
runtime parameters to allow users to control how the opacity is computed for each species and opacity type.
These runtime parameters are shown in Table 21.1. The symbol \(<\text{spec}>\) in the table should be replaced by
the species names as specified using the species setup variable.

<table>
<thead>
<tr>
<th>Runtime Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op_&lt;spec&gt;Absorb</td>
<td>Model to use for the absorption opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;Emiss</td>
<td>Model to use for the emission opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;Trans</td>
<td>Model to use for the transport opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;AbsorbConstant</td>
<td>Constant value to use for absorption opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;EmissConstant</td>
<td>Constant value to use for emission opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;TransConstant</td>
<td>Constant value to use for transport opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;FileName</td>
<td>Name of IONMIX file for tabulated opacity</td>
</tr>
<tr>
<td>op_&lt;spec&gt;FileType</td>
<td>Tabulated opacity file type, either “IONMIX” or “IONMIX4”</td>
</tr>
</tbody>
</table>

When any of the models for a species is set to “op_tabpa”, “op_tabpe”, or “op_tabro”, FLASH will attempt
to read a file which contains the tabulated opacities. The file name, for each species, is given by the
runtime parameter op_<spec>FileName. The type of the file is set using the parameter op_<spec>FileType.
Currently, FLASH only supports reading opacity files in the IONMIX or IONMIX4 format, but this will likely change in the future.

The code segment below shows an example of how one would specify the runtime parameters for the Multispecies opacity in a simulation with two species named cham and targ (this simulation would have been set up using the species=cham,targ setup argument):

```plaintext
# Specify opacity models for the species cham:
op_chamAbsorb = "op_tabpa"  # Use tabulated Planck Absorption opacity for 
                               # the absorption term

# Use the Planck Emission opacity for the emission 
# term

# Use the Rosseland opacity for the transport term

# Specify tabulated opacity file name
op_chamFileName = "he-imx-005.cn4"

# Set tabulated opacity file type to IONMIX4

# Specify opacity models for the species targ:
op_targAbsorb = "op_constcm2g" # Use a constant opacity for the absorption term

# Use a constant opacity for the emission term

# Use a constant opacity for the transport term

# Set targ species absorption opacity to 10 cm^2/g
op_targAbsorbConst = 10.0

# Set targ species absorption opacity to zero
# This species won't emit radiation

# Set targ species transport opacity to a large 
# value Radiation will diffuse very slowly through 
# this material

# This species won't emit radiation
```

In this example species cham uses a tabulated opacity while species targ uses constant opacities. The opacities for targ are chosen so that the material will not emit any radiation, and to suppress the transport of radiation. The LaserSlab simulation shows a comprehensive example of how the Multispecies opacity is used in a realistic HEDP simulation.

### 21.4.3 The IONMIX EOS/Opacity Format

FLASH reads tabulated opacity and Equation Of State (EOS) files in the IONMIX, IONMIX4, and IONMIX6 formats. The IONMIX4 and IONMIX6 formats are very similar and are more flexible than the IONMIX format. Thus, only the IONMIX4 and IONMIX6 formats will be documented here. These three formats are not particularly user friendly and future FLASH releases will likely include support for EOS and opacity file formats which are easier to handle. Nevertheless, by manipulating their data into the IONMIX4 or IONMIX6 formats, users can use their own EOS/opacity data in FLASH simulations. Each file contains information for a single material. All EOS/opacity information is defined on a temperature/density grid. The densities are actually ion number densities.

Below, the IONMIX4 and IONMIX6 formats are defined. They are very similar: the latter is identical to the former but also includes electron specific entropy information. Certain information is ignored by FLASH, meaning that it is read in, but not used for any calculations (right now). Other information is used by the EOS unit alone, or by the opacity unit alone.

1. Number of temperature points
2. Number of ion number density points
3. Line containing information about the atomic number of each element in this material (ignored by FLASH)
4. Line containing information about the relative fraction of each element in this material (ignored by FLASH)
5. Number of radiation energy groups
6. List of temperatures
7. List of ion number densities
8. Average ionization ($\bar{z}$) for each temperature/density (only used for tabulated EOS in FLASH)
9. $d\bar{z}/dT_e [1/eV]$ for each temperature/density (ignored by FLASH)
10. Ion pressure ($P_i [J/cm^3]$ for each temperature/density (only used for tabulated EOS in FLASH)
11. Electron pressure ($P_e [J/cm^3]$) for each temperature/density (only used for tabulated EOS in FLASH)
12. $dP_i/dT_i [J/cm^3/eV]$ for each temperature/density (ignored by FLASH)
13. $dP_e/dT_e [J/cm^3/eV]$ for each temperature/density (ignored by FLASH)
14. Ion specific internal energy $e_i [J/g]$ for each temperature/density (only used for tabulated EOS in FLASH)
15. Electron specific internal energy $e_e [J/g]$ for each temperature/density (only used for tabulated EOS in FLASH)
16. $d e_i/dT_i [J/g/eV]$ for each temperature/density point (ignored by FLASH)
17. $d e_e/dT_e [J/g/eV]$ for each temperature/density point (ignored by FLASH)
18. $d e_i/dn_i [J/g/cm^3]$ for each temperature/density point (ignored by FLASH)
19. $d e_e/dn_e [J/g/cm^3]$ for each temperature/density point (ignored by FLASH)

**ONLY INCLUDE FOR IONMIX6 FORMAT**

Electron specific entropy $[J/g/eV]$ for each temperature/density point.

20. Each energy group boundary [eV]. There are $g+1$ boundaries for a simulation with $g$ groups.
21. Rosseland opacity $[cm^2/g]$ for each density/temperature/energy group (only used for tabulated opacity in FLASH)
22. Planck absorption opacity $[cm^2/g]$ for each density/temperature/energy group (only used for tabulated opacity in FLASH)
23. Planck emission opacity $[cm^2/g]$ for each density/temperature/energy group (only used for tabulated opacity in FLASH)

Below, the exact format of the IONMIX4 file is defined. This is done by providing code listing which shows how the data can be read in using FORTRAN read statements. Users can use this information to convert their own data into the IONMIX4 format so that it can be used by FLASH.

```fortran
! *** Write the header ***
! Write the number of temperature and density points:
write (header,923) ntemp, ndens
!
! Write the atomic number of each element in this material:
write (headr2,921) (izgas(l),l=1,ngases)
!
! Write the fraction (by number of ions) of each element in this material:
write (headr3,922) (fracsp(l),l=1,ngases)
!
write (10,980) header
write (10,980) headr2
write (10,980) headr3
!
! Write the number of radiation energy groups:
```
write (10,982) ngrups

! Write the temperature points (in eV):
write (10,991) (tplsma(it),it=1,ntemp)

! Write the number density points (in cm^-3):
write (10,991) (densnn(id),id=1,ndens)

! Write out the zbar at each temperature/density point (nele/nion):
write (10,991) ((densne(it,id)/densnn(id),it=1,ntemp),id=1,ndens)

! Write d(zbar)/dT:
write (10,991) ((dzdt(it,id),it=1,ntemp),id=1,ndens)

! Write the ion pressure (in Joules/cm^3):
write (10,991) ((densnn(id)*tplsma(it)*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write the electron pressure (in Joules/cm^3):
write (10,991) ((tplsma(it)*densne(it,id)*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write out d(pion)/dT (in Joules/cm^3/eV):
write (10,991) ((densnn(id)*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write out d(pele)/dT (in Joules/cm^3/eV):
write (10,991) & ((tplsma(it)*densnn(id)*dzdt(it,id) + densne(it,id))*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write out the ion specific internal energy (in Joules/gram):
write (10,991) ((enrgyion(it,id),it=1,ntemp),id=1,ndens)

! Write out the electron specific internal energy (in Joules/gram):
write (10,991) ((enrgy(it,id)-enrgyion(it,id),it=1,ntemp),id=1,ndens)

! Write out the ion specific heat (in Joules/gram/eV):
write (10,991) ((heatcpion(it,id),it=1,ntemp),id=1,ndens)

! Write out the electron specific heat (in Joules/gram/eV):
write (10,991) ((heatcp(it,id)-heatcpion(it,id),it=1,ntemp),id=1,ndens)

! Write out d(eion)/d(nion) (I think, but I'm not sure...)
write (10,991) ((dedden_ion(it,id)*condd*densnn(id)/tplsma(it)**2, it=1,ntemp),id=1,ndens)

! Write out d(eele)/d(nele) (I think, but I'm not sure...)
write (10,991) & (((dedden(it,id)-dedden_ion(it,id))*condd*densnn(id)/tplsma(it)**2, it=1,ntemp),id=1,ndens)

! ONLY FOR IONMIX6 FORMAT, IONMIX4 FORMAT DOES NOT INCLUDE ELECTRON
! SPECIFIC ENTROPY:
!
! Write out electron specific entropy
write (10,991) ((entrele(it,id),it=1,ntemp),id=1,ndens)

! Write out the energy group boundaries (in eV):
write (10,991) (engrup(ig),ig=1,ngrups+1)

! Write the Rosseland group opacities
write (10,991) (((orgp(it,id,ig),it=1,ntemp),id=1,ndens),ig=1,ngrups)

! Write the Planck absorption opacities
21.5. Mass Diffusivity

The MassDiffusivity unit implements a prescription for calculating a generic mass diffusivity that can be used by the Hydro PPM, and MHD solvers. In this release the unit only provides non-operational stub functionalities.
Chapter 22

Radiative Transfer Unit

The \texttt{RadTrans} unit is responsible for solving the radiative transfer equation

\[
\frac{1}{c} \frac{\partial I}{\partial t} + \hat{\Omega} \cdot \nabla I + \rho \kappa I = \eta, \tag{22.1}
\]

where, \( I(x, \hat{\Omega}, \nu, t) \) is the radiation intensity, \( c \) is the speed of light, \( \rho \) is the mass density, \( \kappa(x, \nu, t) \) is the opacity in units of \( \text{cm}^2/\text{g} \), \( \eta(x, \nu, t) \) is the emissivity, \( \nu \) is the radiation frequency, and \( \hat{\Omega} \) is the unit direction vector. This equation is coupled to the electron internal energy through

\[
\frac{\partial u_e}{\partial t} = \int_0^\infty d\nu \int_{4\pi} d\hat{\Omega} (\rho \kappa I - \eta), \tag{22.2}
\]

where \( u_e \) is the electron internal energy density.

The \texttt{RadTrans} unit is responsible for solving the radiative transfer equation and updating the electron energy using (22.2) over a single time step. This ensures that the total system energy is conserved. Radiation-hydrodynamics effects, such as work, are operator split and handled by the hydrodynamics unit. Currently, there is only a single \texttt{RadTrans} solver, \texttt{MGD}, which uses a simple multigroup diffusion approximation and is described in Section 22.1.

22.1 Multigroup Diffusion

The radiative transfer (22.1) and electron energy (22.2) equations can be simplified using a multigroup diffusion approximation. The frequency space is divided into \( N_g \) groups with \( N_g+1 \). Group \( g \) is defined by
the frequency range from $\nu_g$ to $\nu_{g+1}$. The plasma is assumed to emit radiation in a Planck spectrum with a given emission opacity. The multigroup diffusion equations solved by FLASH are

\[
\frac{1}{c} \frac{\partial u_g}{\partial t} - \nabla \cdot \left( \frac{1}{3\sigma_{t,g}} \nabla u_g \right) + \sigma_{a,g} u_g = \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} \left[ P(x_{g+1}) - P(x_g) \right]
\]
(22.3)

\[
\frac{\partial u_e}{\partial t} = \sum_g \left\{ \sigma_{a,g} u_g - \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} \left[ P(x_{g+1}) - P(x_g) \right] \right\}
\]
(22.4)

where $u_g$ is the radiation energy density, $\sigma_{t,g}$ is the transport opacity, $\sigma_{a,g}$ is the absorption opacity, $\sigma_{e,g}$ is the emission opacity, $a$ is the radiation constant, $T_e$ is the electron temperature, and $P(x)$ is the Planck integral, defined below. The argument to the Planck integral, is $x = h\nu/k_B T_e$ where $h$ is Planck’s constant and $k_B$ is the Boltzmann constant. The opacities in (22.3) and (22.4) are in units of $1/\text{cm}$ and $\sigma = \rho \kappa$. The Planck integral, $P(x)$ is defined as

\[
P(x) = \int_0^x \frac{(x')^3}{\exp(x') - 1} \, dx'
\]
(22.5)

The multigroup diffusion equation and electron internal energy equation are operator split using an explicit treatment. Thus, on each time step (22.4) is solved for each energy group using coefficients evaluated at time level $n$ using an implicit treatment for the diffusion solution itself. The equation below shows how the multigroup equations are discretized in time

\[
\frac{1}{c} \frac{u_g^{n+1} - u_g^n}{\Delta t} - \nabla \cdot (D_g^n \nabla u_g^{n+1}) + \sigma_{a,g} u_g^{n+1} = \sigma_{e,g} a(T_e^n)^4 \frac{15}{\pi^4} \left[ P(x_{g+1}^n) - P(x_g^n) \right]
\]
(22.6)

\[
\frac{u_e^{n+1} - u_e^n}{\Delta t} = \sum_g \left\{ \sigma_{a,g} u_g^{n+1} - \sigma_{e,g} a(T_e^n)^4 \frac{15}{\pi^4} \left[ P(x_{g+1}^n) - P(x_g^n) \right] \right\}
\]
(22.7)

where the $n$ superscript denotes the time level, and $\Delta t$ is the time step length. These equations are solved on each time step. The diffusion coefficient, $D_g^n$, has been introduced. When no flux-limiter is employed, $D_g^n = 1/3\sigma_{t,g}^n$. The flux-limiter limits the radiation flux in each group to the free streaming limit, $c u_g^n$. Several flux limiter options are available in FLASH. These are described in section 17.1.3. The string runtime parameter $\text{rt\_mgdFlMode}$ controls the flux limiter mode. The maximum flux is set to $q_{\text{max}} = \alpha_r c u_g^n$ for each energy group. The coefficient $\alpha_r$ is set to one by default, which is what is physically most realistic. However, it can be controlled using the runtime parameter $\text{rt\_mgdFlCoef}$.

The diffusion equation, (22.6), is an implicit equation and is solved using the general implicit diffusion solver described in section 17.1.2. It is recommended that users familiarize themselves with the general implicit diffusion solver to learn more about using MGD in FLASH.

### 22.1.1 Using Multigroup Radiation Diffusion

This section describes how to use MGD in FLASH. Section 25.7.5 shows an example of how to use MGD in a realistic HEDP simulation and is extremely informative.

Several setup options are needed to use MGD. The $\text{+mgd}$ setup shortcut will include the MGD unit in the simulation. In addition, storage needs to be created for storing the specific energy for each group. This is done using the $\text{mgd\_meshgroups}$ setup variable. The user must set $\text{mgd\_meshgroups}$ such that:

\[
(\text{mgd\_meshgroups}) N_{\text{mesh}} \geq N_g
\]
(22.8)

where $N_{\text{mesh}}$ represents the number of meshes (please see section 22.1.2 for more information). If you don’t know what this is, you can assume that $N_{\text{mesh}} = 1$. In this case, the constraint is that the $\text{mgd\_meshgroups}$ setup variable must be greater than or equal to the number of energy groups. The actual number of energy groups, $N_g$, is set at run time using the runtime parameter $\text{rt\_mgdNumGroups}$. Thus, as an example, the setup options for a 4-group simulation could be:

\[
\text{+mgd mgd\_meshgroups=6}
\]
and the runtime parameter file must contain \( \text{mgdNumGroups} = 4 \). No changes are needed to the Config file of the simulation.

The FLASH checkpoint files will contain the specific energy in each radiation energy group for each cell. The variables are named \( r### \) where \( ### \) is a three digit number from one to \( N_g \). Users can also add these variables to the list of plot variables so they will be included in plot files.

Several runtime parameters are used to control the MGD unit. The runtime parameter \( \text{useMGD} \) must be set to \( \text{true.} \) for MGD simulations. The energy group structure (values of the group boundaries) are specified, manually, using runtime parameters. A simulation with \( N_g \) groups will have \( N_g + 1 \) energy group boundaries. These are set using the \( \text{mgdBounds}### \) runtime parameters, where \( ### \) is a number between 1 and \( N_g + 1 \). By default, enough runtime parameters exist for specifying 101 energy group boundaries. In the case that more groups are needed, the \( \text{mgd_maxgroups} \) setup variable sets the number of group boundary runtime parameters. For example, if 200 energy groups are needed, then \( \text{mgd_maxgroups} = 200 \) should be specified on the setup line. If \( N_g \leq 100 \), no action is needed.

When MGD is in use, the Opacity unit provides opacities for each cell. Thus, the \( \text{useOpacity} \) runtime parameter must be set to true and the appropriate opacity model must be specified. Please see section 21.4 for a detailed description on how to use the Opacity unit in FLASH.

Finally, the radiation boundary conditions must be specified using runtime parameters. The parameters:

- \( \text{rt_mgdXlBoundaryType} \)
- \( \text{rt_mgdXrBoundaryType} \)
- \( \text{rt_mgdYlBoundaryType} \)
- \( \text{rt_mgdYrBoundaryType} \)
- \( \text{rt_mgdZlBoundaryType} \)
- \( \text{rt_mgdZrBoundaryType} \)

control the boundary condition on each logical face of the domain. The allowed values are:

- \( \text{reflecting, neumann} \): This is a zero-flux, or Neumann boundary condition and should be used to model reflecting boundaries
- \( \text{dirichlet} \): This (poorly named) value represents a fixed radiation temperature boundary condition
- \( \text{vacuum} \): This represents a vacuum boundary condition

Please see section 17.1.2.1 for more information on boundary conditions for the diffusion solver.

As was mentioned earlier, FLASH uses the generalized implicit diffusion solver, described in section 17.1.2, to solve (22.6) for each energy group. Thus, users must make sure to include an appropriate diffusion solver in their simulation.

### 22.1.2 Using Mesh Replication with MGD

When many energy groups are needed, domain decomposition alone is not an effective method for speeding up calculations. The computation time scales roughly linearly with the number of energy groups. As the number of groups increases, users can divide the mesh among larger number of processors to compensate. This is an effective strategy until the strong scaling limit for the simulation is reached. At this point, mesh replication becomes the only way to speed up the simulation further.

Mesh replication allows FLASH to perform the diffusion solves for each energy group in parallel. By default, mesh replication is deactivated and each diffusion equation is solved serially. When mesh replication is in use the total pool of \( N_p \) processes is divided among \( N_m \) identical copies of the computational mesh. The radiation solver uses these copies of the mesh to perform \( N_m \) parallel radiation solves. The runtime parameter \( \text{meshCopyCount} \) is used to set \( N_m \) and has a value of 1 by default.

This is best illustrated with an example. Suppose \( N_g \), the total number of energy groups, is set to 100, and there are \( N_p = 6 \) processes available to run the simulation. If \( \text{meshCopyCount} = 1 \), then \( N_m = 1 \) and the computational mesh will not be replicated. This is FLASH's normal mode of operation: the mesh will be divided into six pieces and each process will be responsible for one of these pieces. Each process will also be involved in the solution of the diffusion equation for all 100 energy groups. If \( \text{meshCopyCount} = 2 \), then the
computational domain will be divided into three pieces. Two processes will be assigned to each piece. In other words two groups of three processes will be created. The first set of three processes will be responsible for solving the diffusion equations for the odd numbered energy groups. The second set of processes will solve the diffusion equations for the even numbered energy groups. Each process only knows about 50 of the 100 groups. In some cases, this may be substantially faster than the meshCopyCount=1 approach.

Another benefit of mesh replication is that it reduces the amount of memory needed per process. In the example above, when meshCopyCount=1, each process must store the energy density for all 100 energy groups. Thus, storage must exist for 100 cell centered variables on each process. When meshCopyCount=2, storage is only needed for 50 cell centered variables. The setup variable mgd_meshgroups controls the number of cell centered variables created on process. Thus, when meshCopyCount=2, we can set mgd_meshgroups to 50 instead of 100. As a result, far less memory is used per block.

Whether or not mesh replication is helpful for a particular simulation can only be determined through extensive testing. It depends on the particular simulation and the hardware.

22.1.3 Specifying Initial Conditions

The initial conditions must be specified throughout the domain for each radiation energy group. This involves setting the specific energy, \( e_g = u_g / \rho \), in units of erg/g for each group. Because of the mesh replication capability, the user is discouraged from manually setting the value of the underlying radiation mass scalars. Instead, several routines have been added to the RadTrans unit to simplify this process dramatically. Specifying the initial conditions involves two steps. First, the value of \( e_g \) must be set for each group in each cell. Second, either the total specific radiation energy, \( e_r = \sum_g e_g \), or the radiation temperature, \( T_r = (u_r/a)^{1/4} \), must be set - depending on the value of eosModeInit. Below, two use cases are described.

22.1.3.1 Initializing using a Radiation Temperature

The easiest way to set the initial condition is to use a radiation temperature in each cell and to assume that the radiation field is described by a Planck spectrum. The subroutine:

\[ \text{RadTrans\_mgdEFromT(blockId, axis, trad, tradActual)} \]

will automatically set the value of \( e_g \) for each group using a given radiation temperature and assuming a black body spectrum. The arguments are:

1. blockId: The block ID
2. axis: A 3 dimensional array of integers indicating the cell within the block
3. trad: An input, the desired radiation temperature
4. tradActual: An output, The actual radiation temperature (described further below)

When the simulation is initialized, it is important that \( T_r = (\sum_g u_g / a)^{1/4} \). This will not always be the case. The reason is that a black body spectrum has a non zero energy density for all frequencies. But FLASH simulations require finite energy group boundaries. For example, if \( \text{trad} \) is set to 11604 K (this corresponds to 1 eV), but the upper energy group boundary is only set to 10 eV, then we are effectively cutting off the part of the spectrum above 10 eV. The actual radiation temperature, which is consistent with the sum of the energies in each group, will be slightly less than 1 eV. The argument \( \text{tradActual} \) contains this new radiation temperature.

Once \( \text{RadTrans\_mgdEFromT} \) has been used to initialize \( e_g \), either the total radiation specific energy, \( e_r \), must be set or the radiation temperature for the cell must be specified. When the runtime parameter eosModeInit is set to “dens_temp_gather”, the initial radiation temperature must be specified in the variable TRAD_VAR. When eosModeInit is set to “dens_ie_gather”, \( e_r \) must be specified in the variable ERAD_VAR.

The example below shows a segment from a Simulation\_initBlock routine from a typical simulation which uses MGD:
do k = blkLimits(LOW,KAXIS),blkLimits(HIGH,KAXIS)
  do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
    do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)
      axis(IAXIS) = i
      axis(JAXIS) = j
      axis(KAXIS) = k

      ! Set the specific energy in each radiation group using a radiation temperature of 1 eV (11604.55 K):
      call RadTrans_mgdEFromT(blockId, axis, 11604.55, tradActual)

      ! Set the radiation temperature:
      call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, tradActual)

      ! Alternatively, we could have set ERAD_VAR using a*(tradActual)**4

    enddo
  enddo
enddo

Note that tradActual is used to specify the value of TRAD_VAR.

22.1.3.2 Manually setting the radiation spectrum

Specifying the radiation temperature is sufficient for many cases. Alternatively, users can manually specify the radiation spectrum. The subroutine

RadTrans_mgdSetEnergy(blockId, axis, group, eg)

has been created for this purpose. The arguments are:

- **blockId**: The block ID
- **axis**: A 3 dimensional array of integers indicating the cell within the block
- **group**: The group number, between 1 and \( N_g \)
- **eg**: The specific radiation energy (erg/g) for the group

This subroutine sets the value of \( e_g \) for a particular cell and should be called from the **Simulation.initBlock** subroutine.

The example below assumes that the user would like to initialize the radiation field so that \( e_1 = aT^4 \rho \) and all other groups are initialized to zero energy. Note, that for this example, the user should obtain the value of \( a \), the radiation constant, from the **PhysicalConstants** unit. This simulations used \( N_g = 4 \).

do k = blkLimits(LOW,KAXIS),blkLimits(HIGH,KAXIS)
  do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
    do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)
      axis(IAXIS) = i
      axis(JAXIS) = j
      axis(KAXIS) = k

      ! Set the specific energy in each radiation group using a radiation temperature of 1 eV (11604.55 K):
      call RadTrans_mgdEFromT(blockId, axis, 11604.55, tradActual)

      ! Set the radiation temperature:
      call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, tradActual)

      ! Alternatively, we could have set ERAD_VAR using a*(tradActual)**4

    enddo
  enddo
enddo
 CHAPTER 22. RADIATIVE TRANSFER UNIT

! Set the specific energy in each radiation group:
call RadTrans_mgdSetEnergy(blockId, axis, 1, a*sim_trad**4/sim_rho)
call RadTrans_mgdSetEnergy(blockId, axis, 2, 0.0)
call RadTrans_mgdSetEnergy(blockId, axis, 3, 0.0)
call RadTrans_mgdSetEnergy(blockId, axis, 4, 0.0)

! Set the radiation temperature:
call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, sim_trad)

! Alternatively, we could have set ERAD_VAR using a*sim_trad**4/sim_rho

enddo
enddo
enddo

22.1.4 Altering the Radiation Spectrum

In some cases, it is necessary to alter the radiation spectrum manually, for example, to add a radiation energy source. In these cases, it is useful to manually alter the mesh variables which store the specific radiation energy within each group: \( e_g = \frac{u_g}{\rho} \). However, because of mesh replication, altering these variables requires a different procedure than modifying other mesh variables (such as the density) in FLASH. The reason is that each process does not, in general, have access to each \( e_g \). The procedure for modifying \( e_g \) directly is as follows:

1. Loop over the energy groups that each process sees
2. Modify \( e_g \)
3. Modify the total specific radiation energy, \( e_{rad} \), so that it stores the sum over all \( e_g \)

Note, that in step 1, looping over the energy groups does not simply involve looping from 1 to \( N_g \). The reason is that each process must loop over only those groups that it is responsible for. Again, because of mesh replication, this may be less than \( N_g \). Below, an example is shown where the specific energy in each group is manually set to \( 10^{12} \text{ erg/g} \). An ideal place for this type of code to exist is in the `Simulation.adjustEvolution` subroutine which gets called every time step.

! Loop over all cells, set the specific radiation energy to zero:
do lb = 1, nblk
   call Grid_getBlkIndexLimits(blklst(lb), blkLimits, blkLimitsGC)
call Grid_getBlkPtr(blklst(lb), blkPtr)

   do k = blkLimits(LOW,KAXIS), blkLimits(HIGH,KAXIS)
      do j = blkLimits(LOW,JAXIS), blkLimits(HIGH,JAXIS)
         do i = blkLimits(LOW,IAXIS), blkLimits(HIGH,IAXIS)
            blkPtr(ERAD_VAR,i,j,k) = 0.0
         end do
      end do
   end do
   call Grid_releaseBlkPtr(blklst(lb), blkPtr)
end do

! Loop over radiation energy groups:
do gloc = 1, NONREP_NLOCS(rt_acrossMe, rt_meshCopyCount, rt_mgdNumGroups)
   ! g represents the global energy group number:
22.1. MULTIGROUP DIFFUSION

\[ g = \text{NONREP\_LOC2GLOB}(\text{gloc, rt\_acrossMe, rt\_meshCopyCount}) \]

! gvar represents the index in the block pointer for group g:
\[ gvar = \text{MGDR\_NONREP\_LOC2UNK}(\text{gloc}) \]

! Loop over blocks and set the energy density in each group:
do lb = 1, nblk
  call \text{Grid\_getBlkIndexLimits}(\text{blklst}(lb), \text{blkLimits}, \text{blkLimitsGC})
  call \text{Grid\_getBlkPtr}(\text{blklst}(lb), \text{blkPtr})
  do k = \text{blkLimits}(LOW,KAXIS), \text{blkLimits}(HIGH,KAXIS)
    do j = \text{blkLimits}(LOW,JAXIS), \text{blkLimits}(HIGH,JAXIS)
      do i = \text{blkLimits}(LOW,IAXIS), \text{blkLimits}(HIGH,IAXIS)
        ! Set the specific energy in group g:
        \text{blkPtr}(gvar,i,j,k) = 1.0e+12
        ! Make sure to track the total radiation energy in the cell:
        \text{blkPtr}(\text{ERAD\_VAR},i,j,k) = \text{blkPtr}(\text{ERAD\_VAR},i,j,k) + \\
        \text{blkPtr}(gvar,i,j,k)
      end do
    end do
  end do
  call \text{Grid\_releaseBlkPtr}(\text{blklst}(lb), \text{blkPtr})
end do

! Finally, every process needs to know the total specific radiation
! energy. \text{RadTrans\_sumEnergy} adds up ERAD\_VAR across all of the
! meshes:
call \text{RadTrans\_sumEnergy}(\text{ERAD\_VAR, nblk, blklst})

! Call EOS to get a consistent state:
do lb = 1, nblk
  call \text{Grid\_getBlkIndexLimits}(\text{blklst}(lb), \text{blkLimits}, \text{blkLimitsGC})
  call \text{Eos\_wrapped}(\text{MODE\_DENS\_EI\_GATHER}, \text{blkLimits}, \text{blklst}(lb))
end do

In the first part of the example, the value of \text{ERAD\_VAR} is set to zero. \text{ERAD\_VAR} simply stores the total specific radiation energy in each cell. After we are done modifying the energy for each group, it is important that \text{ERAD\_VAR} be updated to contain the total specific radiation energy.

The second part of the example loops over the groups. Notice the strange loop ending index. The \text{NONREP\_NLOCS} macro computes the number of groups represented on a given process. To find the actual group number, \( g \), a call is made to \text{NONREP\_LOC2GLOB}. Next, \text{MGDR\_NONREP\_LOC2UNK} is called to get the actual index into the block pointer which corresponds to group \( g \). Following this is a fairly standard loop over blocks and cells where the block pointer is used to modify the specific energy in each group. Notice that we are keeping a running total of the total specific energy in the cell which is stored in \text{ERAD\_VAR}.

Finally, we must call \text{RadTrans\_sumEnergy} to add up \text{ERAD\_VAR} across all of the meshes. After this call, every process will contain the same total specific energy in each cell. This is followed by a call to the equation of state. This is needed to update the radiation temperature and pressure, \text{TRAD\_VAR} and \text{ERAD\_VAR}, respectively.
Part VI

Monitor Units
Chapter 23

Logfile Unit

Figure 23.1: The Logfile unit directory structure.

FLASH supplies the Logfile unit to manage an output log during a FLASH simulation. The logfile contains various types of useful information, warnings, and error messages produced by a FLASH run. Other units can add information to the logfile through the Logfile unit interface. The Logfile routines enable a program to open and close a log file, write time or date stamps to the file, and write arbitrary messages to the file. The file is kept closed and is only opened for appending when information is to be written, thus avoiding problems with unflushed buffers. For this reason, Logfile routines should not be called within time-sensitive loops, as system calls are generated. Even when starting from scratch, the logfile is opened in append mode to avoid deleting important logfiles. Two kinds of Logfiles are supported. The first kind is similar to that in FLASH2 and early releases of FLASH3, where the master processor has exclusive access to the logfile and writes global information to it. The newer kind gives all processors access to their own private logfiles if they need to have one. Similar to the traditional logfile, the private logfiles are opened in append mode, and they are created the first time a processor writes to one. The private logfiles are extremely useful to gather information about failures caused by a small fraction of processors; something that cannot be done in the traditional logfile.

The Logfile unit is included by default in all the provided FLASH simulations because it is required by the Driver/DriverMain Config. As with all the other units in FLASH, the data specific to the Logfile unit is stored in the module Logfile_data.F90. Logfile unit scope data variables begin with the prefix log_variableName and they are initialized in the routine Logfile_init.

By default, the logfile is named flash.log and found in the output directory. The user may change the name of the logfile by altering the runtime parameter log_file in the flash.par.

# names of files
basenm = "cellular_"
log_file = "cellular.log"

### 23.1 Meta Data

The logfile stores meta data about a given run including the time and date of the run, the number of MPI tasks, dimensionality, compiler flags and other information about the run. The snippet below is an example from a logfile showing the basic setup and compilation information:

```plaintext
================================================================================
Number of MPI tasks: 2
MPI version: 1
MPI subversion: 2
Dimensionality: 2
Max Number of Blocks/Proc: 1000
Number x zones: 8
Number y zones: 8
Number z zones: 1
Build stamp: Wed Apr 19 16:35:57 2006
System info:
Linux zingiber.uchicago.edu 2.6.12-1.1376_FC3smp #1 SMP Fri Aug 26 23:50:33 EDT
Version: FLASH 3.0.
Build directory: /home/kantypas/FLASH3/trunk/Sod
Setup syntax:
/home/kantypas/FLASH3/trunk/bin/setup.py Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
-objdir=Sod
f compiler flags:
/usr/local/pgi6/bin/pgf90 -I/usr/local/mpich-pg/include -c -r8 -i4 -fast -g
  -DMAXBLOCKS=1000 -DXNB=8 -DNYB=8 -DNZB=1 -DN_DIM=2
c compiler flags:
/usr/local/pgi6/bin/pgcc -I/usr/local/hdf5-pg/include -I/usr/local/mpich-pg/include
  -c -O2 -DMAXBLOCKS=1000 -DXNB=8 -DNYB=8 -DNZB=1 -DN_DIM=2
================================================================================
```

### 23.2 Runtime Parameters, Physical Constants, and Multispecies Data

The logfile also records which units were included in a simulation, the runtime parameters, physical constants, and any species and their properties from the Multispecies unit. The FLASH3 logfile keeps track of whether a runtime parameter is a default value or whether its value has been redefined in the flash.par file. The [CHANGED] symbol will occur next to a runtime parameter if its value has been redefined in the flash.par file. Note that the runtime parameters are output in alphabetical order within the Fortran datatype – so integer parameters are shown first, then real, then string, then Boolean. The snippet below shows the this portion of the logfile; omitted sections are indicated with “...”.

```plaintext
FLASH Units used:
  Driver
  Driver/DriverMain
  Driver/DriverMain/TimeDep
  Grid
  Grid/GridMain
```
Grid/GridMain/paramesh
Grid/GridMain/paramesh/paramesh4
...
Multispecies
Particles
PhysicalConstants
PhysicalConstants/PhysicalConstantsMain
RuntimeParameters
RuntimeParameters/RuntimeParametersMain
...
physics/utilities/solvers/LinearAlgebra
==============================================================================
RuntimeParameters:
==============================================================================
algebra = 2 [CHANGED]
bndpriorityone = 1
bndprioritythree = 3
... 
cfl = 0.800E+00
checkpointfileintervaltime = 0.100E-08 [CHANGED]
cvisc = 0.100E+00
derefine_cutoff_1 = 0.200E+00
derefine_cutoff_2 = 0.200E+00
... 
zmax = 0.128E+02 [CHANGED]
zmin = 0.000E+00
basenm = cellular_ [CHANGED]
eosmode = dens_ie
eosmodeinit = dens_ie
geometry = cartesian
log_file = cellular.log [CHANGED]
output_directory =
pc_unitsbase = CGS
plot_grid_var_1 = none
plot_grid_var_10 = none
plot_grid_var_11 = none
plot_grid_var_12 = none
plot_grid_var_2 = none
... 
yr_boundary_type = periodic
zl_boundary_type = periodic
zr_boundary_type = periodic
bytepack = F
chkguardcells = F
converttoconsvdformeshcalls = F
converttoconsvdinmeshinterp = F
... 
useburn = T [CHANGED]
useburntable = F
==============================================================================
Known units of measurement:
CHAPTER 23. LOGFILE UNIT

Unit | CGS Value | Base Unit
--- | --- | ---
1 | cm | 1.0000 | cm
2 | s | 1.0000 | s
3 | K | 1.0000 | K
4 | g | 1.0000 | g
5 | esu | 1.0000 | esu
6 | m | 100.00 | cm
7 | km | 0.10000E+06 | cm
8 | pc | 0.30857E+19 | cm

Known physical constants:

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Constant Value</th>
<th>cm</th>
<th>s</th>
<th>g</th>
<th>K</th>
<th>esu</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Newton</td>
<td>0.66726E-07</td>
<td>3.00</td>
<td>-2.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2 speed of light</td>
<td>0.29979E+11</td>
<td>1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15 Euler</td>
<td>0.57722</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

==============================================================================

Multifluid database contents:

Initially defined values of species:

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
<th>Total</th>
<th>Positive</th>
<th>Neutral</th>
<th>Negative</th>
<th>bind</th>
<th>Ener</th>
<th>Gamma</th>
</tr>
</thead>
</table>
ar36  | 12    | 3.60E+01 | 1.80E+01 | -9.99E+02 | -9.99E+02 | 3.07E+02 | -9.99E+02 |
c12   | 13    | 1.20E+01 | 6.00E+00 | -9.99E+02 | -9.99E+02 | 9.22E+01 | -9.99E+02 |
ca40  | 14    | 4.00E+01 | 2.00E+01 | -9.99E+02 | -9.99E+02 | 3.42E+02 | -9.99E+02 |
| ...  | ...   | ...    | ...      | ...      | ...      | ...   | ...   | ...    |
ti44  | 24    | 4.40E+01 | 2.20E+01 | -9.99E+02 | -9.99E+02 | 3.75E+02 | -9.99E+02 |

==============================================================================

23.3 Accessor Functions and Timestep Data

Other units within FLASH may make calls to write information, or stamp, the logfile. For example, the Driver unit calls the API routine Logfile stamp after each timestep. The Grid unit calls Logfile stamp whenever refinement occurs in an adaptive grid simulation. If there is an error that is caught in the code the API routine Driver abortFlash stamps the logfile before aborting the code. Any unit can stamp the logfile with one of two routines Logfile stamp which includes a data and time stamp along with a logfile message, or Logfile stampMessage which simply writes a string to the logfile.

The routine Logfile stamp is overloaded so the user must use the interface file Logfile interface.F90 in the calling routine. The next snippit shows logfile output during the evolution loop of a FLASH run.

==============================================================================

[ 04-19-2006 16:40.43 ] [Simulation_init]: initializing Sod problem
[GRID amr_refine_derefine] initiating refinement
[GRID amr_refine_derefine] min blks 0 max blks 1 tot blks 1
[GRID amr_refine_derefine] min leaf blks 0 max leaf blks 1 tot leaf blks 1
[GRID amr_refine_derefine] refinement complete
[ 04-19-2006 16:40.43 ] [GRID gr_expandDomain]: create level=2
...
23.4 Performance Data

Finally, the logfile records performance data for the simulation. The Timers unit (see Section 24.1) is responsible for storing, collecting and interpreting the performance data. The Timers unit calls the API routine Logfile_writeSummary to format the performance data and write it to the logfile. The snippet below shows the performance data section of a logfile.

```
perf_summary: code performance summary
    beginning : 04-19-2006 16:40.43
    ending: 04-19-2006 16:41.06
    seconds in monitoring period : 23.188
    number of subintervals : 21
    number of evolved zones : 16064
    zones per second : 692.758

-----------------------------------------------------
<table>
<thead>
<tr>
<th>accounting unit</th>
<th>time sec</th>
<th>num calls</th>
<th>secs avg</th>
<th>time pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>1.012</td>
<td>1</td>
<td>1.012</td>
<td>4.366</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.155</td>
<td>17</td>
<td>0.009</td>
<td>0.669</td>
</tr>
<tr>
<td>writeCheckpoint</td>
<td>0.085</td>
<td>1</td>
<td>0.085</td>
<td>0.365</td>
</tr>
<tr>
<td>writePlotfile</td>
<td>0.061</td>
<td>1</td>
<td>0.061</td>
<td>0.264</td>
</tr>
<tr>
<td>evolution</td>
<td>22.176</td>
<td>1</td>
<td>22.176</td>
<td>95.633</td>
</tr>
<tr>
<td>hydro</td>
<td>18.214</td>
<td>40</td>
<td>0.455</td>
<td>78.549</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>2.603</td>
<td>80</td>
<td>0.033</td>
<td>11.227</td>
</tr>
<tr>
<td>sourceTerms</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.002</td>
</tr>
<tr>
<td>particles</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.001</td>
</tr>
<tr>
<td>Grid_updateRefinement</td>
<td>1.238</td>
<td>20</td>
<td>0.062</td>
<td>5.340</td>
</tr>
<tr>
<td>tree</td>
<td>1.126</td>
<td>10</td>
<td>0.113</td>
<td>4.856</td>
</tr>
<tr>
<td>guardcell tree</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.459</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.458</td>
</tr>
<tr>
<td>markRefineDerefine</td>
<td>0.339</td>
<td>10</td>
<td>0.034</td>
<td>1.460</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.053</td>
<td>10</td>
<td>0.005</td>
<td>0.230</td>
</tr>
<tr>
<td>amr_refine_derefine</td>
<td>0.003</td>
<td>10</td>
<td>0.000</td>
<td>0.011</td>
</tr>
<tr>
<td>updateData</td>
<td>0.002</td>
<td>10</td>
<td>0.000</td>
<td>0.009</td>
</tr>
<tr>
<td>guardcell</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.453</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.452</td>
</tr>
</tbody>
</table>
```
23.5 Example Usage

An example program using the Logfile unit might appear as follows:

```fortran
program testLogfile
    use Logfile_interface, ONLY: Logfile_init, Logfile_stamp, Logfile_open, Logfile_close
    use Driver_interface, ONLY: Driver_initParallel
    use RuntimeParameters_interface, ONLY: RuntimeParameters_init
    use PhysicalConstants_interface, ONLY: PhysicalConstants_init
    implicit none

    integer :: i
    integer :: log_lun
    integer :: myPE, numProcs
    logical :: restart, localWrite

    call Driver_initParallel(myPE, numProcs) !will initialize MPI
    call RuntimeParameters_init(myPE, restart) ! Logfile_init needs runtime parameters
    call PhysicalConstants_init(myPE) ! PhysicalConstants information adds to logfile
    call Logfile_init(myPE, numProcs) ! will end with Logfile_create(myPE, numProcs)

    call Logfile_stamp (myPE, "beginning log file test...", ">[program testLogfile]"")
    localWrite=.true.
    call Logfile_open(log_lun,localWrite) !! open the local logfile
    do i = 1, 10
        write (log_lun,*), 'i = ', i
    enddo
    call Logfile_stamp (myPE, "finished logfile test", ">[program testLogfile]"")
    call Logfile_close(myPE, log_lun)

end program testLogfile
```
Chapter 24

Timer and Profiler Units

24.1 Timers

Figure 24.1: The Timers unit directory tree.

24.1.1 MPINative

FLASH includes an interface to a set of stopwatch-like timing routines for monitoring performance. The interface is defined in the monitors/Timers unit, and an implementation that uses the timing functionality provided by MPI is provided in monitors/Timers/TimersMain/MPINative. Future implementations might use the PAPI framework to track hardware counter details.

The performance routines start or stop a timer at the beginning or end of a section of code to be monitored, and accumulate performance information in dynamically assigned accounting segments. The code also has an interface to write the timing summary to the FLASH logfile. These routines are not recommended for timing very short segments of code due to the overhead in accounting.

There are two ways of using the Timers routines in your code. One mode is to simply pass timer names as strings to the start and stop routines. In this first way, a timer with the given name will be created if it doesn’t exist, or otherwise reference the one already in existence. The second mode of using the timers references them not by name but by an integer key. This technique offers potentially faster access if a timer
is to be started and stopped many times (although still not recommended because of the overhead). The integer key is obtained by calling with a string name `Timers_create` which will only create the timer if it doesn’t exist and will return the integer key. This key can then be passed to the start and stop routines.

The typical usage pattern for the timers is implemented in the default `Driver` implementation. This pattern is: call `Timers_init` once at the beginning of a run, call `Timers_start` and `Timers_stop` around sections of code, and call `Timers_getSummary` at the end of the run to report the timing summary at the end of the logfile. However, it is possible to call `Timers_reset` in the middle of a run to reset all timing information. This could be done along with writing the summary once per-timestep to report code times on a per-timestep basis, which might be relevant, for instance, for certain non-fixed operation count solvers. Since `Timers_reset` does not reset the integer key mappings, it is safe to obtain a key through `Timers_create` once in a saved variable, and continue to use it after calling `Timers_reset`.

Two runtime parameters control the `Timer` unit and are described below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eachProcWritesSummary</td>
<td>LOGICAL</td>
<td>TRUE</td>
<td>Should each process write its summary to its own file? If true, each process will write its summary to a file named <code>timer_summary_&lt;process id&gt;</code></td>
</tr>
<tr>
<td>writeStatSummary</td>
<td>LOGICAL</td>
<td>TRUE</td>
<td>Should timers write the max/min/avg values for timers to the logfile?</td>
</tr>
</tbody>
</table>

`monitors/Timers/TimersMain/MPINative` writes two summaries to the logfile: the first gives the timer execution of the master processor, and the second gives the statistics of max, min, and avg times for timers on all processors. The secondary max, min, and avg times will not be written if some process executed timers differently than another. For example, this anomaly happens if not all processors contain at least one block. In this case, the Hydro timers only execute on the processors that possess blocks. See Section 23.4 for an example of this type of output. The max, min, and avg summary can be disabled by setting the runtime parameter `writeStatSummary` to false. In addition, each process can write its summary to its own file named `timer_summary_<process id>`. To prohibit each process from writing its summary to its own file, set the runtime parameter `eachProcWritesSummary` to false.

### 24.1.2 Tau

In `FLASH3.1` we add an alternative `Timers` implementation which is designed to be used with the `Tau` framework (http://acts.nersc.gov/tau/). Here, we use `Tau` API calls to time the `FLASH` labeled code sections (marked by `Timers_start` and `Timers_stop`). After running the simulation, the `Tau` profile contains timing information for both `FLASH` labeled code sections and all individual subroutines / functions. This is useful because fine grained subroutine / function level data can be overwhelming in a huge code like `FLASH`. Also, the callpaths are preserved, meaning we can see how long is spent in individual subroutines / functions when they are called from within a particular `FLASH` labeled code section. Another reason to use the `Tau` version is that the `MPINative` version (See Section 24.1.1) is implemented using recursion, and so incurs significant overhead for fine grain measurements.

To use this implementation we must compile the `FLASH` source code with the `Tau` compiler wrapper scripts. These are set as the default compilers automatically whenever we specify the `-tau` option (see Section 5.2) to the setup script. In addition to the `-tau` option we must specify `--with-unit=monitors/Timers/Timers-Main/Tau` as this `Timers` implementation is not the default.
24.2 Profiler

In addition to an interface for simple timers, FLASH includes a generic interface for third-party profiling or tracing libraries. This interface is defined in the monitors/Profiler unit.

In FLASH4 we created an interface to the IBM profiling libraries libmpihpm.a and libmpihpm_smp.a and also to HPCToolkit http://hpctoolkit.org/ (Rice University). We make use of this interface to profile FLASH evolution only, i.e. not initialization. To use this style of profiling add \texttt{-unit=monitors/Profiler/ProfilerMain/mpihpm} or \texttt{-unit=monitors/Profiler/ProfilerMain/hpctoolkit} to your setup line and also set the FLASH runtime parameter profileEvolutionOnly = .true.

For the IBM profiling library (mpihpm) you need to add \texttt{LIB(MPIHPM} and \texttt{LIB(MPIHPM\_SMP} macros to your Makefile.h to link FLASH to the profiling libraries. The actual macro used in the link line depends on whether you setup FLASH with multithreading support (\texttt{LIB(MPIHPM} for MPI-only FLASH and \texttt{LIB(MPIHPM\_SMP} for multithreaded FLASH). Example values from sites/miralac1/Makefile.h follow

\begin{verbatim}
LIB\_MPI =
HPM\_COUNTERS = /bgsys/drivers/ppcfloor/bgpm/lib/libbgpm.a
LIB\_MPIHPM = -L/soft/perftools/hpctw -lmpihpm $(HPM\_COUNTERS) $(LIB\_MPI)
LIB\_MPIHPM\_SMP = -L/soft/perftools/hpctw -lmpihpm_smp $(HPM\_COUNTERS) $(LIB\_MPI)
\end{verbatim}

For HPCToolkit you need to set the environmental variable HPRUN\_DELAY\_SAMPLING=1 at job launch to enable selective profiling (see the HPCToolkit user guide).
Part VII

Simulation Units
Chapter 25

The Supplied Test Problems

Figure 25.1: The Simulation unit directory tree. Only some of the provided simulation implementations are shown. Users are expected to add their own simulations to the tree.

To verify that FLASH works as expected and to debug changes in the code, we have created a suite of standard test problems. Many of these problems have analytical solutions that can be used to test the accuracy of the code. Most of the problems that do not have analytical solutions produce well-defined flow features that have been verified by experiments and are stringent tests of the code. For the remaining problems, converged solutions, which can be used to test the accuracy of lower resolution simulations, are easy to obtain. The test suite configuration code is included with the FLASH source tree (in the Simulation/ directory), so it is easy to configure and run FLASH with any of these problems ‘out of the box.’ Sample runtime parameter files are also included. All the test problems reside in the Simulations unit. The unit provides some general interfaces most of which do not have general implementations. Each application provides its own implementation for these interfaces, for example, Simulation_initBlock. The exception is Simulation_initSpecies, which provides general implementations for different classes of problems.

25.1 Hydrodynamics Test Problems

These problems are primarily designed to test the functioning of the hydrodynamics solvers within FLASH4.
25.1.1 Sod Shock-Tube

The Sod problem (Sod 1978) is a one-dimensional flow discontinuity problem that provides a good test of a compressible code’s ability to capture shocks and contact discontinuities with a small number of cells and to produce the correct profile in a rarefaction. It also tests a code’s ability to correctly satisfy the Rankine-Hugoniot shock jump conditions. When implemented at an angle to a multidimensional grid, it can be used to detect irregularities in planar discontinuities produced by grid geometry or operator splitting effects.

We construct the initial conditions for the Sod problem by establishing a planar interface at some angle to the \(x\)- and \(y\)-axes. The fluid is initially at rest on either side of the interface, and the density and pressure jumps are chosen so that all three types of nonlinear, hydrodynamic waves (shock, contact, and rarefaction) develop. To the “left” and “right” of the interface we have

\[
\rho_L = 1.0 \quad \rho_R = 0.125 \\
p_L = 1.0 \quad p_R = 0.1
\]  

(25.1)

The ratio of specific heats \(\gamma\) is chosen to be 1.4 on both sides of the interface.

In FLASH, the Sod problem \((\text{Sod})\) uses the runtime parameters listed in Table 25.1 in addition to those supplied by default with the code. For this problem we use the Gamma equation of state alternative implementation and set \(\text{gamma}\) to 1.4. The default values listed in Table 25.1 are appropriate to a shock with normal parallel to the \(x\)-axis that initially intersects that axis at \(x = 0.5\) (halfway across a box with unit dimensions).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{sim_rhoLeft}</td>
<td>real</td>
<td>1</td>
<td>Initial density to the left of the interface ((\rho_L))</td>
</tr>
<tr>
<td>\text{sim_rhoRight}</td>
<td>real</td>
<td>0.125</td>
<td>Initial density to the right ((\rho_R))</td>
</tr>
<tr>
<td>\text{sim_pLeft}</td>
<td>real</td>
<td>1</td>
<td>Initial pressure to the left ((p_L))</td>
</tr>
<tr>
<td>\text{sim_pRight}</td>
<td>real</td>
<td>0.1</td>
<td>Initial pressure to the right ((p_R))</td>
</tr>
<tr>
<td>\text{sim_uLeft}</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the left ((u_L))</td>
</tr>
<tr>
<td>\text{sim_uRight}</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the right ((u_R))</td>
</tr>
<tr>
<td>\text{sim_xangle}</td>
<td>real</td>
<td>90</td>
<td>Angle made by interface normal with the (x)-axis ((\text{degrees}))</td>
</tr>
<tr>
<td>\text{sim_yangle}</td>
<td>real</td>
<td>90</td>
<td>Angle made by interface normal with the (y)-axis ((\text{degrees}))</td>
</tr>
<tr>
<td>\text{sim_posn}</td>
<td>real</td>
<td>0.5</td>
<td>Point of intersection between the interface plane and the (x)-axis</td>
</tr>
</tbody>
</table>

Figure 25.2 shows the result of running the Sod problem with FLASH on a two-dimensional grid with the analytical solution shown for comparison. The hydrodynamical algorithm used here is the directionally split piecewise-parabolic method (PPM) included with FLASH. In this run the shock normal is chosen to be parallel to the \(x\)-axis. With six levels of refinement, the effective grid size at the finest level is \(256^2\), so the finest cells have width 0.00390625. At \(t = 0.2\), three different nonlinear waves are present: a rarefaction between \(x = 0.263\) and \(x = 0.486\), a contact discontinuity at \(x = 0.685\), and a shock at \(x = 0.850\). The two discontinuities are resolved with approximately two to three cells each at the highest level of refinement, demonstrating the ability of PPM to handle sharp flow features well. Near the contact discontinuity and in the rarefaction, we find small errors of about \(1 - 2\%\) in the density and specific internal energy, with similar errors in the velocity inside the rarefaction. Elsewhere, the numerical solution is close to exact; no oscillations are present.

Figure 25.3 shows the result of running the Sod problem on the same two-dimensional grid with different shock normals: parallel to the \(x\)-axis \((\theta = 0^\circ)\) and along the box diagonal \((\theta = 45^\circ)\). For the diagonal solution, we have interpolated values of density, specific internal energy, and velocity to a set of 256 points.
Figure 25.2: Comparison of numerical and analytical solutions to the Sod problem. A 2D grid with six levels of refinement is used. The shock normal is parallel to the $x$-axis.
Figure 25.3: Comparison of numerical solutions to the Sod problem for two different angles ($\theta$) of the shock normal relative to the $x$-axis. A 2D grid with six levels of refinement is used.
spaced exactly as in the $x$-axis solution. This comparison shows the effects of the second-order directional splitting used with FLASH on the resolution of shocks. At the right side of the rarefaction and at the contact discontinuity, the diagonal solution undergoes slightly larger oscillations (on the order of a few percent) than the $x$-axis solution. Also, the value of each variable inside the discontinuity regions differs between the two solutions by up to 10%. However, the location and thickness of the discontinuities is the same for the two solutions. In general, shocks at an angle to the grid are resolved with approximately the same number of cells as shocks parallel to a coordinate axis.

Figure 25.4 presents a colormap plot of the density at $t = 0.2$ in the diagonal solution together with the block structure of the AMR grid. Note that regions surrounding the discontinuities are maximally refined, while behind the shock and contact discontinuity, the grid has de-refined, because the second derivative of the density has decreased in magnitude. Because zero-gradient outflow boundaries were used for this test, some reflections are present at the upper left and lower right corners, but at $t = 0.2$ these have not yet propagated to the center of the grid.

Figure 25.4: Density in the diagonal 2D Sod problem with six levels of refinement at $t = 0.2$. The outlines of AMR blocks are shown (each block contains $8 \times 8$ cells).

25.1.2 Variants of the Sod Problem in Curvilinear Geometries

Variants of the Sod problems can be set up in various geometries to test the handling of non-Cartesian geometries.

- An axisymmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with .\setup Sod -auto -2d -geometry=cylindrical and using runtime parameters that include geometry = "cylindrical". Use sim_xangle = 0 to configure an initial shock front that is shaped like a cylinder. Results as in those discussed in Toro 1999 can be obtained.
• A spherically symmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with ./setup Sod -auto -1d -geometry=spherical and using runtime parameters that include geometry = "spherical". Again results as in those discussed in Toro 1999 can be obtained.

• To test the behavior of FLASH solutions when the physical symmetry of the problem does not match the geometry of the simulation, a separate simulation is provided under the name SodSpherical. To use this, configure with ./setup SodSpherical -auto -2d -geometry=spherical and using runtime parameters that include geometry = "spherical". As a 2D setup, SodSpherical represents physically axisymmetric initial conditions in spherical coordinates. The physical problem can be chosen to be the same as in the previous case with cylindrical Sod. Again results as in those discussed in Toro 1999 can be obtained.

• The SodSpherical setup can also configured in 1D and will act like the 1D Sod setup in that case.

25.1.3 Interacting Blast-Wave Blast2

This Blast2 problem was originally used by Woodward and Colella (1984) to compare the performance of several different hydrodynamical methods on problems involving strong shocks and narrow features. It has no analytical solution (except at very early times), but since it is one-dimensional, it is easy to produce a converged solution by running the code with a very large number of cells, permitting an estimate of the self-convergence rate. For FLASH, it also provides a good test of the adaptive mesh refinement scheme.

The initial conditions consist of two parallel, planar flow discontinuities. Reflecting boundary conditions are used. The density is unity and the velocity is zero everywhere. The pressure is large at the left and right and small in the center

\[ p_L = 1000, \quad p_M = 0.01, \quad p_R = 100. \]  

(25.2)

The equation of state is that of a perfect gas with \( \gamma = 1.4 \).

Figure 25.5 shows the density and velocity profiles at several different times in the converged solution, demonstrating the complexity inherent in this problem. The initial pressure discontinuities drive shocks into the middle part of the grid; behind them, rarefactions form and propagate toward the outer boundaries, where they are reflected back into the grid. By the time the shocks collide at \( t = 0.028 \), the reflected rarefactions have caught up to them, weakening them and making their post-shock structure more complex. Because the right-hand shock is initially weaker, the rarefaction on that side reflects from the wall later, so the resulting shock structures going into the collision from the left and right are quite different. Behind each shock is a contact discontinuity left over from the initial conditions (at \( x \approx 0.50 \) and 0.73). The shock collision produces an extremely high and narrow density peak. The peak density should be slightly less than 30. However, the peak density shown in Figure 25.5 is only about 18, since the maximum value of the density does not occur at exactly \( t = 0.028 \). Reflected shocks travel back into the colliding material, leaving a complex series of contact discontinuities and rarefactions between them. A new contact discontinuity has formed at the point of the collision (\( x \approx 0.69 \)). By \( t = 0.032 \), the right-hand reflected shock has met the original right-hand contact discontinuity, producing a strong rarefaction, which meets the central contact discontinuity at \( t = 0.034 \). Between \( t = 0.034 \) and \( t = 0.038 \), the slope of the density behind the left-hand shock changes as the shock moves into a region of constant entropy near the left-hand contact discontinuity.

Figure 25.6 shows the self-convergence of density and pressure when FLASH is run on this problem. We compare the density, pressure, and total specific energy at \( t = 0.038 \) obtained using FLASH with ten levels of refinement to solutions using several different maximum refinement levels. This figure plots the L1 error norm for each variable \( u \), defined using

\[ E(N_{ref}; u) = \frac{1}{N(N_{ref})} \sum_{i=1}^{N(N_{ref})} \left| \frac{u_i(N_{ref}) - Au_i(10)}{u_i(10)} \right|, \]  

(25.3)

against the effective number of cells \( (N(N_{ref})) \). In computing this norm, both the ‘converged’ solution \( u(10) \) and the test solution \( u(N_{ref}) \) are interpolated onto a uniform mesh having \( N(N_{ref}) \) cells. Values of \( N_{ref} \) between 2 (corresponding to cell size \( \Delta x = 1/16 \)) and 9 (\( \Delta x = 1/2048 \)) are shown.
Figure 25.5: Density and velocity profiles in the Woodward-Colella interacting blast-wave problem Blast2 as computed by FLASH using ten levels of refinement.
Although PPM is formally a second-order method, the convergence rate is only linear. This is not surprising, since the order of accuracy of a method applies only to smooth flow and not to flows containing discontinuities. In fact, all shock capturing schemes are only first-order accurate in the vicinity of discontinuities. Indeed, in their comparison of the performance of seven nominally second-order hydrodynamic methods on this problem, Woodward and Colella found that only PPM achieved even linear convergence; the other methods were worse. The error norm is very sensitive to the correct position and shape of the strong, narrow shocks generated in this problem.

The additional runtime parameters supplied with the 2blast problem are listed in Table 25.2. This problem is configured to use the perfect-gas equation of state (\texttt{gamma}) with \texttt{gamma} set to 1.4 and is run in a two-dimensional unit box. Boundary conditions in the y-direction (transverse to the shock normals) are taken to be periodic.
25.1. HYDRODYNAMICS TEST PROBLEMS

Figure 25.6: Self-convergence of the density, pressure, and total specific energy in the Blast2 test problem.

Table 25.2: Runtime parameters used with the 2blast test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rho_left</td>
<td>real</td>
<td>1</td>
<td>Initial density to the left of the left interface ($\rho_L$)</td>
</tr>
<tr>
<td>rho_mid</td>
<td>real</td>
<td>1</td>
<td>Initial density between the two interfaces ($\rho_M$)</td>
</tr>
<tr>
<td>rho_right</td>
<td>real</td>
<td>1</td>
<td>Initial density to the right of the right interface ($\rho_R$)</td>
</tr>
<tr>
<td>p_left</td>
<td>real</td>
<td>1000</td>
<td>Initial pressure to the left ($p_L$)</td>
</tr>
<tr>
<td>p_mid</td>
<td>real</td>
<td>0.01</td>
<td>Initial pressure in the middle ($p_M$)</td>
</tr>
<tr>
<td>p_right</td>
<td>real</td>
<td>100</td>
<td>Initial pressure to the right ($p_R$)</td>
</tr>
<tr>
<td>u_left</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the left ($u_L$)</td>
</tr>
<tr>
<td>u_mid</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) in the middle ($u_M$)</td>
</tr>
<tr>
<td>u_right</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the right ($u_R$)</td>
</tr>
<tr>
<td>xangle</td>
<td>real</td>
<td>0</td>
<td>Angle made by interface normal with the x-axis (degrees)</td>
</tr>
<tr>
<td>yangle</td>
<td>real</td>
<td>90</td>
<td>Angle made by interface normal with the y-axis (degrees)</td>
</tr>
<tr>
<td>posnL</td>
<td>real</td>
<td>0.1</td>
<td>Point of intersection between the left interface plane and the x-axis</td>
</tr>
<tr>
<td>posnR</td>
<td>real</td>
<td>0.9</td>
<td>Point of intersection between the right interface plane and the x-axis</td>
</tr>
</tbody>
</table>
25.1.4 Sedov Explosion

The Sedov explosion problem (Sedov 1959) is another purely hydrodynamical test in which we check the code’s ability to deal with strong shocks and non-planar symmetry. The problem involves the self-similar evolution of a cylindrical or spherical blast wave from a delta-function initial pressure perturbation in an otherwise homogeneous medium. To initialize the code, we deposit a quantity of energy \( E = 1 \) into a small region of radius \( \delta r \) at the center of the grid. The pressure inside this volume \( p'_{0} \) is given by

\[
p'_{0} = \frac{3(\gamma - 1)E}{(\nu + 1)^{3/2}} \delta r^{\nu},
\]

where \( \nu = 2 \) for cylindrical geometry and \( \nu = 3 \) for spherical geometry. We set the ratio of specific heats \( \gamma = 1.4 \). In running this problem we choose \( \delta r \) to be 3.5 times as large as the finest adaptive mesh resolution in order to minimize effects due to the Cartesian geometry of our grid. The density is set equal to \( \rho_{0} = 1 \) everywhere, and the pressure is set to a small value \( p_{0} = 10^{-5} \) everywhere but in the center of the grid. The fluid is initially at rest. In the self-similar blast wave that develops for \( t > 0 \), the density, pressure, and radial velocity are all functions of \( \xi \equiv r/R(t) \), where

\[
R(t) = C_{\nu}(\gamma) \left( \frac{Et^{2}}{\rho_{0}} \right)^{1/(\nu+2)}.
\]

Here \( C_{\nu} \) is a dimensionless constant depending only on \( \nu \) and \( \gamma \); for \( \gamma = 1.4 \), \( C_{2} \approx C_{3} \approx 1 \) to within a few percent. Just behind the shock front at \( \xi = 1 \) the analytical solution is

\[
\begin{align*}
\rho &= \rho_{1} = \frac{\gamma + 1}{\gamma - 1} \rho_{0} \\
\rho &= \rho_{1} = \frac{2}{\gamma + 1} \rho_{0} u^{2} \\
v &= v_{1} = \frac{2}{\gamma + 1} u,
\end{align*}
\]

where \( u \equiv dR/dt \) is the speed of the shock wave. Near the center of the grid,

\[
\begin{align*}
\rho(\xi)/\rho_{1} &\propto \xi^{\nu/(\gamma-1)} \\
p(\xi)/p_{1} &= \text{constant} \\
v(\xi)/v_{1} &\propto \xi.
\end{align*}
\]

Figure 25.7 shows density, pressure, and velocity profiles in the two-dimensional, cylindrical Sedov problem at \( t = 0.05 \). Solutions obtained with FLASH on grids with 2, 4, 6, and 8 levels of refinement are shown in comparison with the analytical solution. In this figure we have computed average radial profiles in the following way. We interpolated solution values from the adaptively gridded mesh used by FLASH onto a uniform mesh having the same resolution as the finest AMR blocks in each run. Then, using radial bins with the same width as the cells in the uniform mesh, we binned the interpolated solution values, computing the average value in each bin. At low resolutions, errors show up as density and velocity overestimates behind the shock, underestimates of each variable within the shock, and a very broad shock structure. However, the central pressure is accurately determined, even for two levels of refinement. Because the density goes to a finite value rather than to its correct limit of zero, this corresponds to a finite truncation of the temperature (which should go to infinity as \( r \to 0 \)). This error results from depositing the initial energy into a finite-width region rather than starting from a delta function. As the resolution improves and the value of \( \delta r \) decreases, the artificial finite density limit also decreases; by \( N_{\text{ref}} = 6 \) it is less than 0.2% of the peak density. Except for the \( N_{\text{ref}} = 2 \) case, which does not show a well-defined peak in any variable, the shock itself is always captured with about two cells. The region behind the shock containing 90% of the swept-up material is represented by four cells in the \( N_{\text{ref}} = 4 \) case, 17 cells in the \( N_{\text{ref}} = 6 \) case, and 69 cells for \( N_{\text{ref}} = 8 \). However, because the solution is self-similar, for any given maximum refinement level, the structure will be four cells wide at a sufficiently early time. The behavior when the shock is under-resolved is to underestimate the peak value of each variable, particularly the density and pressure.
Figure 25.7: Comparison of numerical and analytical solutions to the Sedov problem in two dimensions. Numerical solution values are averages in radial bins at the finest AMR grid resolution \( N_{\text{ref}} \) in each run.
Figure 25.8 shows the pressure field in the 8-level calculation at \( t = 0.05 \) together with the block refinement pattern. Note that a relatively small fraction of the grid is maximally refined in this problem. Although the pressure gradient at the center of the grid is small, this region is refined because of the large temperature gradient there. This illustrates the ability of PARAMESH to refine grids using several different variables at once.

![Figure 25.8](image.png)

Figure 25.8: Pressure field in the 2D Sedov explosion problem with 8 levels of refinement at \( t = 0.05 \). The outlines of the AMR blocks are overlaid on the pressure colormap.

We have also run FLASH on the spherically symmetric Sedov problem in order to verify the code’s performance in three dimensions. The results at \( t = 0.05 \) using five levels of grid refinement are shown in Figure 25.9. In this figure we have plotted the average values as well as the root-mean-square (RMS) deviations from the averages. As in the two-dimensional runs, the shock is spread over about two cells at the finest AMR resolution in this run. The width of the pressure peak in the analytical solution is about 1 1/2 cells at this time, so the maximum pressure is not captured in the numerical solution. Behind the shock the numerical solution average tracks the analytical solution quite well, although the Cartesian grid geometry produces RMS deviations of up to 40% in the density and velocity in the de-refined region well behind the shock. This behavior is similar to that exhibited in the two-dimensional problem at comparable resolution.

The additional runtime parameters supplied with the Sedov problem are listed in Table 25.3. This problem is configured to use the perfect-gas equation of state (\( \Gamma \)) with \( \gamma \) set to 1.4. It is simulated in a unit-sized box.

### 25.1.4.1 Sedov Self-Gravity

Another variant of the Sedov problem is included in the release which runs with spherical coordinates in one dimension. The Sedov Self-Gravity problem also allows the effects of gravitational acceleration where the gravitational potential is calculated using the multipole solver. Figure 25.10 and 25.11 show the snapshots of the density profile and the gravitational potential at two different times during the evolution. The first snapshot is at \( t = 0.5 \ \text{sec} \), when evolution is halfway through, while the second snapshot is at the end of the evolution, where \( t = 1.0 \ \text{sec} \).
Figure 25.9: Comparison of numerical and analytical solutions versus radius $r$ to the spherically symmetric Sedov problem. A 3D grid with five levels of refinement is used.
Table 25.3: Runtime parameters used with the Sedov test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_pAmbient</td>
<td>real</td>
<td>10^{-3}</td>
<td>Initial ambient pressure ($p_0$)</td>
</tr>
<tr>
<td>sim_rhoAmbient</td>
<td>real</td>
<td>1</td>
<td>Initial ambient density ($\rho_0$)</td>
</tr>
<tr>
<td>sim_expEnergy</td>
<td>real</td>
<td>1</td>
<td>Explosion energy ($E$)</td>
</tr>
<tr>
<td>sim_rInit</td>
<td>real</td>
<td>0.05</td>
<td>Radius of initial pressure perturbation ($\delta r$)</td>
</tr>
<tr>
<td>sim_xctr</td>
<td>real</td>
<td>0.5</td>
<td>$x$-coordinate of explosion center</td>
</tr>
<tr>
<td>sim_yctr</td>
<td>real</td>
<td>0.5</td>
<td>$y$-coordinate of explosion center</td>
</tr>
<tr>
<td>sim_zctr</td>
<td>real</td>
<td>0.5</td>
<td>$z$-coordinate of explosion center</td>
</tr>
<tr>
<td>sim_nSubZones</td>
<td>integer</td>
<td>7</td>
<td>Number of sub-cells in cells for applying the 1D profile</td>
</tr>
</tbody>
</table>

Figure 25.10: Snapshots of Sedov Self-gravity density profile and gravitational potential at time $t=0.5$ sec.

Figure 25.11: Snapshots of Sedov Self-gravity density profile and gravitational potential at time $t=1.0$ sec.
25.1.5 Isentropic Vortex

The two-dimensional isentropic vortex problem is often used as a benchmark for comparing numerical methods for fluid dynamics. The flow-field is smooth (there are no shocks or contact discontinuities) and contains no steep gradients, and the exact solution is known. It was studied by Yee, Vinokur, and Djomehri (2000) and by Shu (1998). In this subsection the problem is described, the FLASH control parameters are explained, and some results demonstrating how the problem can be used are presented.

The simulation domain is a square, and the center of the vortex is located at \((x_{ctr}, y_{ctr})\). The flow-field is defined in coordinates centered on the vortex center \((x' = x - x_{ctr}, y' = y - y_{ctr})\) with \(r^2 = x'^2 + y'^2\). The domain is periodic, but it is assumed that off-domain vortexes do not interact with the primary; practically, this assumption can be satisfied by ensuring that the simulation domain is large enough for a particular vortex strength. We find that a domain size of \(10 \times 10\) (specified through the `Grid` runtime parameters \(x_{min}, x_{max}, y_{min}, \) and \(y_{max}\)) is sufficiently large for a vortex strength (defined below) of 5.0. In the initialization below, \(x'\) and \(y'\) are the coordinates with respect to the nearest vortex in the periodic sense.

The ambient conditions are given by \(\rho_\infty, u_\infty, v_\infty, \) and \(p_\infty,\) and the non-dimensional ambient temperature is \(T^*_\infty = 1.0.\) Using the equation of state, the (dimensional) \(T_\infty\) is computed from \(p_\infty\) and \(\rho_\infty.\) Perturbations are added to the velocity and nondimensionalized temperature, \(u = u_\infty + \delta u, v = v_\infty + \delta v,\) and \(T^* = T^*_\infty + \delta T^*\) according to

\[
\begin{align*}
\delta u &= -y' \frac{\beta}{2\pi} \exp \left( \frac{1 - r^2}{2} \right) \tag{25.8} \\
\delta v &= x' \frac{\beta}{2\pi} \exp \left( \frac{1 - r^2}{2} \right) \tag{25.9} \\
\delta T^* &= -\frac{(\gamma - 1)\beta}{8\gamma \pi^2} \exp \left( 1 - r^2 \right) \tag{25.10}
\end{align*}
\]

where \(\gamma = 1.4\) is the ratio of specific heats and \(\beta = 5.0\) is a measure of the vortex strength. The temperature and density are then given by

\[
\begin{align*}
T &= \frac{T_\infty}{T^*_\infty} \tag{25.11} \\
\rho &= \rho_\infty \left( \frac{T}{T_\infty} \right)^{\frac{1}{\gamma - 1}}. \tag{25.12}
\end{align*}
\]

At any location in space, the conserved variables (density, \(x\)- and \(y\)-momentum, and total energy) can be computed from the above quantities. The flow-field is initialized by computing cell averages of the conserved variables; in each cell, the average is approximated by averaging over \(nx_{subint} \times ny_{subint}\) subintervals. The runtime parameters for the isentropic vortex problem are listed in Table 25.4.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient pressure ((p_\infty))</td>
</tr>
<tr>
<td>rho_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient density ((\rho_\infty))</td>
</tr>
<tr>
<td>u_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient (x)-velocity ((u_\infty))</td>
</tr>
<tr>
<td>v_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient (y)-velocity ((v_\infty))</td>
</tr>
<tr>
<td>vortex_strength</td>
<td>real</td>
<td>5.0</td>
<td>Non-dimensional vortex strength</td>
</tr>
<tr>
<td>xctr</td>
<td>real</td>
<td>0.0</td>
<td>(x)-coordinate of vortex center</td>
</tr>
<tr>
<td>yctr</td>
<td>real</td>
<td>0.0</td>
<td>(y)-coordinate of vortex center</td>
</tr>
<tr>
<td>nx_subint</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (x)-direction</td>
</tr>
<tr>
<td>ny_subint</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (y)-direction</td>
</tr>
</tbody>
</table>
Chapter 25: The Supplied Test Problems

Figure 25.12 shows the exact density distribution represented on a $40 \times 40$ uniform grid with $-5.0 \leq x, y \leq 5.0$. The borders of each grid block ($8 \times 8$ cells) are superimposed. In addition to the shaded representation, contour lines are shown for $\rho = 0.95$, $0.85$, $0.75$, and $0.65$. The density distribution is radially symmetric, and the minimum density is $\rho_{\text{min}} = 0.510287$. Because the exact solution of the isentropic vortex problem is the initial solution shifted by $(u_\infty t, v_\infty t)$, numerical phase (dispersion) and amplitude (dissipation) errors are easy to identify. Dispersive errors distort the shape of the vortex, breaking its symmetry. Dissipative errors smooth the solution and flatten extrema; for the vortex, the minimum in density at the vortex core will increase.

Figure 25.12: Density at $t = 0.0$ for the isentropic vortex problem. Shown are the initial condition and the exact solution at $t = 10.0, 20.0, \ldots$.

A numerical simulation using the PPM scheme was run to illustrate such errors. The simulation used the same grid shown in Figure 25.12 with the same contour levels and color values. The grid is intentionally coarse and the evolution time long to make numerical errors visible. The vortex is represented by approximately $8$ grid points in each coordinate direction and is advected diagonally with respect to the grid. At solution times of $t = 10.0, 20.0, \ldots$, etc., the vortex should be back at its initial location.

Figure 25.13 shows the solution at $t = 50.0$; only slight differences are observed. The density distribution is almost radially symmetric, although the minimum density has risen to $0.0537360$. Accumulating dispersion error is clearly visible at $t = 100.0$ (Figure 25.14), and the minimum density is now $0.601786$.

Figure 25.15 shows the density near $y = 0.0$ at three simulation times. The black line shows the initial condition. The red line corresponds to $t = 50.0$ and the blue line to $t = 100.0$. For the later two times, the density is not radially symmetric. The lines plotted are just representative profiles for those times, which give an idea of the magnitude and character of the errors.

25.1.6 The double Mach reflection problem

This numerical planar shock test problem introduced by Woodward and Colella (1984) simulates an evolution of an unsteady planar shock that is incident on an oblique surface. Initially, the incident planar shock begins to propagate to the bottom surface at a $30^\circ$ angle with the shock Mach number of $10$. Later, the solution to this problem produces a self-similar wave pattern that corresponds to the double Mach reflection. Following
Figure 25.13: Density at $t = 50.0$ for the isentropic vortex problem.

Figure 25.14: Density at $t = 100.0$ for the isentropic vortex problem.
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

Figure 25.15: Representative density profiles for the isentropic vortex near \( y = 0 \) at \( t = 0 \) (black), \( t = 50 \) (red), and \( t = 100 \) (blue).

many other numerical setups of this problem, we tilt the incident shock rather than the reflecting wall so as to avoid numerical complications in modeling an oblique physical boundary.

The initial setup involves a Mach 10 shock in air, \( \gamma = 1.4 \), on a rectangular 2D domain of size \([0, 4] \times [0, 1]\). The reflecting wall is represented as the bottom surface of the domain, beginning at \( x = 1/6 \). The velocity normal to the incident shock in the post-shock region is 8.25, and the flow is at rest in the pre-shock region. The undisturbed air ahead of the shock has a density of 1.4 and a pressure of 1. See the initial density profile in Figure 25.16 resolved on 6 refinement AMR levels using \( 16^3 \)-cell block size.

The boundary condition on \( 0 \leq x \leq 1/6 \) at \( y = 0 \) is fixed in time with the initial values so that the reflected shock is attached to the bottom surface. We impose reflecting boundary condition (i.e., negating the \( y \) velocity field \( v \)) on the rest of the bottom surface. On the top surface of \( y = 1 \), we allow the initial Mach 10 shock proceeds exactly as a function of time in order that the numerical evolution follows the oblique shock propagation without any planar distortion. At \( x = 0 \), we impose a supersonic inflow boundary condition, and the outflow condition at \( x = 4 \).

In later time, the solution develops to form two Mach stems and two contact discontinuities, as shown in Figure 25.17 the density at \( t = 2.5 \) sec. Also shown in Figure 25.18 are 30 levels of contour lines of pressure, illustrating the evolved flow discontinuities at \( t = 2.5 \) sec. We can also see that the numerical solution is smooth and non-oscillatory in the region bounded by the curved reflected shock, the bottom surface and the second Mach stem (see more discussion in Woodward and Colella, 1984).

The 5th order WENO method in the unsplit hydrodynamics solver is used with a choice of hybrid Riemann solver which selectively adopts HLL only at strong shocks and HLLC otherwise.
25.1. HYDRODYNAMICS TEST PROBLEMS

Figure 25.16: The initial density at $t = 0$ visualized with 6 levels of AMR block structures.

Figure 25.17: Density profile at $t = 2.5$. Two contact discontinuities are denoted as "CD", along with two Mach stems, a curved reflected shock, and a jet formulation of the denser fluid along the bottom surface.

Figure 25.18: The 30 levels of contour lines of pressure at $t = 2.5$. 
Table 25.5: Runtime parameters used with the double Mach reflection test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_rholLeft</td>
<td>real</td>
<td>8</td>
<td>Initial density to the left of the shock ($\rho_L$)</td>
</tr>
<tr>
<td>sim_rholRight</td>
<td>real</td>
<td>1.4</td>
<td>Initial density to the right ($\rho_R$)</td>
</tr>
<tr>
<td>sim_pLleft</td>
<td>real</td>
<td>116.5</td>
<td>Initial pressure to the left ($p_L$)</td>
</tr>
<tr>
<td>sim_pRleft</td>
<td>real</td>
<td>1</td>
<td>Initial pressure to the right ($p_R$)</td>
</tr>
<tr>
<td>sim_uLleft</td>
<td>real</td>
<td>7.1447096</td>
<td>Initial $x$-velocity to the left ($u_L$)</td>
</tr>
<tr>
<td>sim_uRleft</td>
<td>real</td>
<td>0</td>
<td>Initial $x$-velocity to the right ($u_R$)</td>
</tr>
<tr>
<td>sim_uLleft</td>
<td>real</td>
<td>-4.125</td>
<td>Initial $y$-velocity to the left ($v_L$)</td>
</tr>
<tr>
<td>sim_uRleft</td>
<td>real</td>
<td>0</td>
<td>Initial $y$-velocity to the right ($v_R$)</td>
</tr>
<tr>
<td>sim_xangle</td>
<td>real</td>
<td>-30</td>
<td>Angle made by shock normal with the $x$-axis (degrees)</td>
</tr>
<tr>
<td>sim_yangle</td>
<td>real</td>
<td>90</td>
<td>Angle made by shock normal with the $y$-axis (degrees)</td>
</tr>
<tr>
<td>sim_posn</td>
<td>real</td>
<td>1/6</td>
<td>Point of intersection between the shock plane and the $x$-axis</td>
</tr>
</tbody>
</table>

25.1.7 Wind Tunnel With a Step

The problem of a wind tunnel containing a step, WindTunnel, was first described by Emery (1968), who used it to compare several hydrodynamical methods. Woodward and Colella (1984) later used it to compare several more advanced methods, including PPM. Although it has no analytical solution, this problem is useful because it exercises a code’s ability to handle unsteady shock interactions in multiple dimensions. It also provides an example of the use of FLASH to solve problems with irregular boundaries.

The problem uses a two-dimensional rectangular domain three units wide and one unit high. Between $x = 0.6$ and $x = 3$ along the $x$-axis is a step 0.2 units high. The step is treated as a reflecting boundary, as are the lower and upper boundaries in the $y$-direction. For the right-hand $x$-boundary, we use an outflow (zero gradient) boundary condition, while on the left-hand side we use an inflow boundary. In the inflow boundary cells, we set the density to $\rho_0$, the pressure to $p_0$, and the velocity to $u_0$, with the latter directed parallel to the $x$-axis. The domain itself is also initialized with these values. We use

$$\rho_0 = 1.4, \quad p_0 = 1, \quad u_0 = 3, \quad \gamma = 1.4,$$

(25.13)

which corresponds to a Mach 3 flow. Because the outflow is supersonic throughout the calculation, we do not expect reflections from the right-hand boundary.

The additional runtime parameters supplied with the WindTunnel problem are listed in Table 25.6. This problem is configured to use the perfect-gas equation of state (Gamma) with $\gamma$ set to 1.4. We also set $x_{\max} = 3$, $y_{\max} = 1$, $N_{\text{blockx}} = 15$, and $N_{\text{blocky}} = 5$ in order to create a grid with the correct dimensions. The version of Simulation_defineDomain supplied with this problem removes all but the first three top-level blocks along the lower edge of the grid to generate the step, and gives REFLECTING boundaries to the obstacle blocks. Finally, we use $x_l_{\text{boundary type}} = \text{"user"}$ (USER_DEFINED condition) and $x_r_{\text{boundary type}} = \text{"outflow"}$ (OUTFLOW boundary) to instruct FLASH to use the correct boundary conditions in the $x$-direction. Boundaries in the $y$-direction are reflecting (REFLECTING).

Until $t = 12$, the flow is unsteady, exhibiting multiple shock reflections and interactions between different types of discontinuities. Figure 25.19 shows the evolution of density and velocity between $t = 0$ and $t = 4$ (the period considered by Woodward and Colella). Immediately, a shock forms directly in front of the step and begins to move slowly away from it. Simultaneously, the shock curves around the corner of the step, extending farther downstream and growing in size until it strikes the upper boundary just after $t = 0.5$. The corner of the step becomes a singular point, with a rarefaction fan connecting the still gas just above the step to the shocked gas in front of it. Entropy errors generated in the vicinity of this singular point produce a numerical boundary layer about one cell thick along the surface of the step. Woodward and Colella reduce this effect by resetting the cells immediately behind the corner to conserve entropy and the sum of enthalpy.
Figure 25.19: Density and velocity in the Emery wind tunnel test problem, as computed with FLASH. A 2D grid with five levels of refinement is used.
Figure 25.19: Density and velocity in the Emery wind tunnel test problem (continued).
Table 25.6: Runtime parameters used with the WindTunnel test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim.pAmbient</td>
<td>real</td>
<td>1</td>
<td>Ambient pressure ($p_0$)</td>
</tr>
<tr>
<td>sim.rhoAmbient</td>
<td>real</td>
<td>1.4</td>
<td>Ambient density ($\rho_0$)</td>
</tr>
<tr>
<td>sim.windVel</td>
<td>real</td>
<td>3</td>
<td>Inflow velocity ($u_0$)</td>
</tr>
</tbody>
</table>

and specific kinetic energy through the rarefaction. However, we are less interested here in reproducing the exact solution than in verifying the code and examining the behavior of such numerical effects as resolution is increased, so we do not apply this additional boundary condition. The errors near the corner result in a slight over-expansion of the gas there and a weak oblique shock where this gas flows back toward the step. At all resolutions we also see interactions between the numerical boundary layer and the reflected shocks that appear later in the calculation.

The shock reaches the top wall at $t \approx 0.65$. The point of reflection begins at $x \approx 1.45$ and then moves to the left, reaching $x \approx 0.95$ at $t = 1$. As it moves, the angle between the incident shock and the wall increases until $t = 1.5$, at which point it exceeds the maximum angle for regular reflection ($40^\circ$ for $\gamma = 1.4$) and begins to form a Mach stem. Meanwhile the reflected shock has itself reflected from the top of the step, and here too the point of intersection moves leftward, reaching $x \approx 1.65$ by $t = 2$. The second reflection propagates back toward the top of the grid, reaching it at $t = 2.5$ and forming a third reflection. By this time in low-resolution runs, we see a second Mach stem forming at the shock reflection from the top of the step; this results from the interaction of the shock with the numerical boundary layer, which causes the angle of incidence to increase faster than in the converged solution. Figure 25.20 compares the density field at $t = 4$ as computed by FLASH using several different maximum levels of refinement. Note that the size of the artificial Mach reflection diminishes as resolution improves.

The shear cell behind the first ("real") Mach stem produces another interesting numerical effect, visible at $t \geq 3$ — Kelvin-Helmholtz amplification of numerical errors generated at the shock intersection. The waves thus generated propagate downstream and are refracted by the second and third reflected shocks. This effect is also seen in the calculations of Woodward and Colella, although their resolution was too low to capture the detailed eddy structure we see. Figure 25.21 shows the detail of this structure at $t = 4$ on grids with several different levels of refinement. The effect does not disappear with increasing resolution, for three reasons. First, the instability amplifies numerical errors generated at the shock intersection, no matter how small. Second, PPM captures the slowly moving, nearly vertical Mach stem with only 1–2 cells on any grid, so as it moves from one column of cells to the next, artificial kinks form near the intersection, providing the seed perturbation for the instability. Third, the effect of numerical viscosity, which can diffuse away instabilities on course grids, is greatly reduced at high resolution. This effect can be reduced by using a small amount of extra dissipation to smear out the shock, as discussed by Colella and Woodward (1984). This tendency of physical instabilities to amplify numerical noise vividly demonstrates the need to exercise caution when interpreting features in supposedly converged calculations.

Finally, we note that in high-resolution runs with FLASH, we also see some Kelvin-Helmholtz roll up at the numerical boundary layer along the top of the step. This is not present in Woodward and Colella's calculation, presumably because their grid resolution was lower (corresponding to two levels of refinement for us) and because of their special treatment of the singular point.

### 25.1.8 The Shu-Osher problem

The Shu-Osher problem (Shu and Osher, 1989) tests a shock-capturing scheme’s ability to resolve small-scale flow features. It gives a good indication of the numerical (artificial) viscosity of a method. Since it is designed to test shock-capturing schemes, the equations of interest are the one-dimensional Euler equations for a single-species perfect gas.

In this problem, a (nominally) Mach 3 shock wave propagates into a sinusoidal density field. As the shock advances, two sets of density features appear behind the shock. One set has the same spatial frequency as the unshocked perturbations, but for the second set, the frequency is doubled. Furthermore, the second set
Figure 25.20: Density at $t = 4$ in the Emery wind tunnel test problem, as computed with FLASH using several different levels of refinement.
follows more closely behind the shock. None of these features is spurious. The test of the numerical method is to accurately resolve the dynamics and strengths of the oscillations behind the shock.

The shu_osher problem is initialized as follows. On the domain \(-4.5 \leq x \leq 4.5\), the shock is at \(x = x_s\) at \(t = 0.0\). On either side of the shock,

\[
\begin{align*}
\rho(x) & = \rho_L & \rho_R (1.0 + a_{\rho} \sin(f_{\rho}x)) \\
p(x) & = p_L & p_R \\
u(x) & = u_L & u_R
\end{align*}
\]  \tag{25.14}

where \(a_{\rho}\) is the amplitude and \(f_{\rho}\) is the frequency of the density perturbations. The \texttt{gamma} equation of state alternative implementation is used with \texttt{gamma} set to 1.4. The runtime parameters and their default values are listed in Table 25.7. The initial density, \(x\)-velocity, and pressure distributions are shown in Figure 25.22.

Table 25.7: Runtime parameters used with the \texttt{shu_osher} test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>posn</td>
<td>real</td>
<td>-4.0</td>
<td>Initial shock location ((x_s))</td>
</tr>
<tr>
<td>rho_left</td>
<td>real</td>
<td>3.857143</td>
<td>Initial density to the left of the shock ((\rho_L))</td>
</tr>
<tr>
<td>rho_right</td>
<td>real</td>
<td>1.0</td>
<td>Nominal initial density to the right ((\rho_R))</td>
</tr>
<tr>
<td>p_left</td>
<td>real</td>
<td>10.33333</td>
<td>Initial pressure to the left ((p_L))</td>
</tr>
<tr>
<td>p_right</td>
<td>real</td>
<td>1.0</td>
<td>Initial pressure to the right ((p_R))</td>
</tr>
<tr>
<td>u_left</td>
<td>real</td>
<td>2.629369</td>
<td>Initial velocity to the left ((u_L))</td>
</tr>
<tr>
<td>u_right</td>
<td>real</td>
<td>0.0</td>
<td>Initial velocity to the right ((u_R))</td>
</tr>
<tr>
<td>a_rho</td>
<td>real</td>
<td>0.2</td>
<td>Amplitude of the density perturbations</td>
</tr>
</tbody>
</table>
The problem is strictly one-dimensional; building 2-d or 3-d executables should give the same results along each x-direction grid line. For this problem, special boundary conditions are applied. The initial conditions should not change at the boundaries; if they do, errors at the boundaries can contaminate the results. To avoid this possibility, a boundary condition subroutine was written to set the boundary values to their initial values.

The purpose of the tests is to determine how much resolution, in terms of mesh cells per feature, a particular method requires to accurately represent small scale flow features. Therefore, all computations are carried out on equispaced meshes without adaptive refinement. Solutions are obtained at $t = 1.8$. The reference solution, using 3200 mesh cells, is shown in Figure 25.23. This solution was computed using PPM at a CFL number of 0.8. Note the shock located at $x \simeq 2.4$, and the high frequency density oscillations just to the left of the shock. When the grid resolution is insufficient, shock-capturing schemes underpredict the amplitude of these oscillations and may distort their shape.

The density field for the same scheme at 400 mesh cells and at 200 mesh cells. With 400 cells, the amplitudes are only slightly reduced compared to the reference solution; however, the shapes of the oscillations have been distorted. The slopes are steeper and the peaks and troughs are broader, which is the result of overcompression from the contact-steepening part of the PPM algorithm. For the solution on 200 mesh cells, the amplitudes of the high-frequency oscillations are significantly underpredicted.

25.1.9 Driven Turbulence StirTurb

The driven turbulence problem StirTurb simulates homogeneous, isotropic and weakly-compressible turbulence. Because theories of turbulence generally assume a steady state, and because turbulence is inherently a dissipative phenomenon, the fluid must be driven to sustain a steady-state. This driving must be done carefully in order to avoid introducing artifacts into the turbulent flow. We use a relatively sophisticated stochastic driving method originally introduced by Eswaran & Pope (1988). The initial conditions setup up a homogeneous background. The resolution used for this test run was $32^3$, and the boundary conditions were periodic. The Table 25.8 shows values the runtime parameters values to control the amount of driving, and the Figures Figure 25.25 and Figure 25.26 show the density and x velocity profile of an xy plane in the center of the domain.

25.1.10 Relativistic Sod Shock-Tube

The relativistic version of the shock tube problem (RHD_Sod) is a simple one-dimensional setup that involves the decay of an initially discontinuous two fluids into three elementary wave structures: a shock, a contact, and a rarefaction wave. As in Newtonian hydrodynamics case, this type of problem is useful in addressing the ability of the Riemann solver to check the code correctness in evolving such simple elementary waves.

We construct the initial conditions for the relativistic shock tube problem as found in Martí & Müller (2003). We use an ideal equation of state with $\Gamma = 5/3$ for this problem and the left and right states are

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_stirmax</td>
<td>real</td>
<td>25.1327</td>
<td>maximum stirring wavenumber</td>
</tr>
<tr>
<td>st_stirmin</td>
<td>real</td>
<td>6.2832</td>
<td>minimum stirring wavenumber</td>
</tr>
<tr>
<td>st_energy</td>
<td>real</td>
<td>5E-6</td>
<td>energy input per mode</td>
</tr>
<tr>
<td>st_decay</td>
<td>real</td>
<td>0.5</td>
<td>correlation time for driving</td>
</tr>
<tr>
<td>st_freq</td>
<td>integer</td>
<td>1</td>
<td>frequency of stirring</td>
</tr>
</tbody>
</table>
25.1. HYDRODYNAMICS TEST PROBLEMS

Figure 25.22: Initial density, $x$-velocity, and pressure for the Shu-Osher problem.
Figure 25.23: Density, $x$-velocity, and pressure for the reference solution at $t = 1.8$. 
Figure 25.24: Density fields on 400 and 200 mesh cells from the PPM scheme.

Figure 25.25: Density profile for the StirTurb driven turbulence problem.
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

Figure 25.26: velocity along X dimension for the StirTurb driven turbulence problem.

given:

\[
\rho_L = 10 \quad \rho_R = 1.0 \\
p_L = \frac{40}{3} \quad p_R = \frac{2}{3} \times 10^{-6}
\]  

(25.15)

and the computational domain is \(0 \leq x \leq 1\). The initial shock location is at \(x = 0.5\) (halfway across a box with unit dimensions).

In FLASH, the RHD Sod problem (RHD_Sod) uses the runtime parameters listed in Table 25.9:

Table 25.9: Runtime parameters used with the RHD_Sod test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_rhLeft</td>
<td>real</td>
<td>10</td>
<td>Initial density to the left of the interface (\rho_L)</td>
</tr>
<tr>
<td>sim_rhRight</td>
<td>real</td>
<td>1.0</td>
<td>Initial density to the right (\rho_R)</td>
</tr>
<tr>
<td>sim_pLeft</td>
<td>real</td>
<td>40/3</td>
<td>Initial pressure to the left (p_L)</td>
</tr>
<tr>
<td>sim_pRight</td>
<td>real</td>
<td>2/3 \times 10^{-6}</td>
<td>Initial pressure to the right (p_R)</td>
</tr>
<tr>
<td>sim_uLeft</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the left (u_L)</td>
</tr>
<tr>
<td>sim_uRight</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the right (u_R)</td>
</tr>
<tr>
<td>sim_xangle</td>
<td>real</td>
<td>0</td>
<td>Angle made by interface normal with the x-axis (degrees)</td>
</tr>
<tr>
<td>sim_yangle</td>
<td>real</td>
<td>90</td>
<td>Angle made by interface normal with the y-axis (degrees)</td>
</tr>
<tr>
<td>sim_posn</td>
<td>real</td>
<td>0.5</td>
<td>Point of intersection between the interface plane and the x-axis</td>
</tr>
</tbody>
</table>

Figure 25.27, Figure 25.28, and Figure 25.29 show the results of running the RHD Sod problem on a one-dimensional uniform grid of size 400 at simulation time \(t = 0.36\). In this run the left-going wave is the rarefaction wave, while two right-going waves are the contact discontinuity and the shock wave. This configuration results in mildly relativistic effects that are mainly thermodynamical in nature.

The differences in the relativistic regime, as compared to Newtonian hydrodynamics, can be seen in a curved velocity profile for the rarefaction wave and the narrow constant state (density shell) in between...
25.1. HYDRODYNAMICS TEST PROBLEMS

Figure 25.27: Density of numerical solution to the relativistic Sod problem at time $t = 0.36$. The shock wave and contact discontinuity. Numerically, it is particularly challenging to resolve the thin narrow density plateau, which is bounded by a leading shock front and a trailing contact discontinuity (see Figure 25.27).
Figure 25.28: Pressure of numerical solution to the relativistic Sod problem at time \( t = 0.36 \).

Figure 25.29: Normal velocity of numerical solution to the relativistic Sod problem at time \( t = 0.36 \).
25.1. HYDRODYNAMICS TEST PROBLEMS

Figure 25.30: Log of density of numerical solution to the relativistic 2D Riemann problem at time $t = 0.8$. The solution was resolved on AMR grid with 6 levels of refinements.

25.1.11 Relativistic Two-dimensional Riemann

The two-dimensional Riemann problem (RHD_Riemann2D), originally proposed by Schulz et al. (1993), involves studying interactions of four basic waves that consist of shocks, rarefactions, and contact discontinuities. The initial condition provided here is based on Migone et al. (2005) producing these elementary waves at every interface. The setup of the problem is given on a rectangular domain of size $[-1, 1] \times [-1, 1]$, which is divided into four constant state subregions as:

$$
\begin{align*}
(rho, p, v_x, v_y) &= \begin{cases}
(0.5, 1.0, 0.0, 0.0) & \quad -1.0 \leq x < 0.0, \quad -1.0 \leq y < 0.0 \\
(0.1, 1.0, 0.0, 0.99) & \quad 0.0 \leq x \leq 1.0, \quad -1.0 \leq y < 0.0 \\
(0.1, 1.0, 0.99, 0.0) & \quad -1.0 \leq x < 0.0, \quad 0.0 \leq y \leq 1.0 \\
(rho_1, p_1, 0.0, 0.0) & \quad 0.0 \leq x \leq 1.0, \quad 0.0 \leq y \leq 1.0
\end{cases} 
\end{align*}
$$

(25.16)

where $rho_1 = 5.477875 \times 10^{-3}$ and $p_1 = 2.762987 \times 10^{-3}$. An ideal EOS is used with the specific heat ratio $\Gamma = 5/3$.

The solution obtained at $t = 0.8$ in Figure 25.30 shows that the symmetry of the problem is well maintained. The two shocks are propagated from the upper left and the lower right regions to the upper right region, yielding continuous collisions of shocks at the upper right corner. The curved shock fronts are transmitted and formed in the diagonal region of the domain. The lower left region is bounded by contact discontinuities. By the time $t = 0.8$ most of regions are filled with shocked gas, whereas there are still two unperturbed regions in the lower left and upper right regions.
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

25.1.12 Flow Interactions with Stationary Rigid Body

The stationary rigid body is only implemented and tested in the unsplit hydro solver Section 14.1.3. It is possible that the unsplit staggered mesh MHD solver Section 14.3.3 can support the rigid body but we have not tested yet.

25.1.12.1 NACA Airfoil

Flow simulations over a series of NACA airfoils (or a simple flat plate) can be obtained using the implementation of a stationary rigid body in the unsplit hydro solver described in Section 14.1.3.1. In this example, the cambered NACA2412 airfoil is initialized with positive unity values indicating the part of the domain that belongs to a stationary rigid body. The rest of the domain is assigned negative unity values to indicate it as a flow region for the unsplit hydro solver. At the surface of the rigid body, a reflecting boundary condition is applied in order to represent the fact that there is no flow penetrating the rigid object.

Plots in 25.31(a) – 25.33(b) illustrate Mach number and pressure plots over the airfoil with the three different initial Mach numbers, 0.65, 0.95 and 1.2 at \( t = 1.8 \). By this time, the flow conditions have reached their steady states. For Mach number \( = 0.65 \), the critical Mach number has not yet been obtained and the flow over the airfoil is all subsonic as shown in 25.31(a). Since the airfoil is asymmetric and cambered, we see there are pressure gradients across the top and bottom surfaces even at zero angle of attack in 25.31(b). These pressure gradients (higher pressure at the bottom than the top) generate a lift force by which an airplane can fly defying gravity.

At Mach number reaching 0.95 as shown in 25.32(a) there are local points that are supersonic. This indicates that the critical Mach number for the airfoil is between 0.65 and 0.95. In fact, one can show that the critical Mach number is around 0.7 for the NACA2412 airfoil. We see that there is a development of a bow shock formation in front of the airfoil. A formation of a subsonic region between the bow shock and the nose of the airfoil is visible in the Mach number plot. Inside the bow shock, a sonic line at which the local flow speed becomes the sound speed makes an oval shape together with the bow shock. In both the Mach number and pressure plots, a strong wake forms starting from the top and bottom of the surfaces near the trailing edge. The wake is hardly visible for Mach number 0.65 in 25.31(a) and 25.31(b). Normal shock waves have formed stemming from the trailing edge as seen in 25.32(a) and 25.32(b).

At Mach number 1.2 the flow becomes supersonic everywhere. In this case, the shape of the bow shock becomes narrower and there are much larger supersonic pockets developed on the top and bottom surfaces with a smaller subsonic region between the bow shock and the nose region.

![Figure 25.31: NACA2412 in Mach number 0.65 flow at 0 degree angle of attack problem at t = 1.8 (a) Mach number (b) Pressure](image)

Figure 25.31: NACA2412 in Mach number 0.65 flow at 0 degree angle of attack problem at \( t = 1.8 \) (a) Mach number (b) Pressure
25.1. HYDRODYNAMICS TEST PROBLEMS

25.1.1 Hydrodynamics Test Problems

Another problem for testing a stationary rigid body in a simulation is to consider the Sedov-like explosion in a chamber surrounded by a solid wall with holes. The wall is shown as red blocks with white boundary in 25.34(a) and 25.34(b). The simulation was done on a uniform Cartesian grid with 300 cells on each direction. Three holes in the wall subdivide it into four different stationary solid bodies in a square computational domain $[0, 1.5] \times [0, 1.5]$. The explosion goes off at the origin and generates shock waves inside the chamber. In later time, when the shock waves pass through the three holes in the wall, turbulence effects are triggered from the interaction between the fluid and the wall and enhance vortical fluid motions.

One important thing in this problem is to keep the given symmetry throughout the simulation. The flow symmetry across the diagonal direction is well preserved in 25.34(a) and 25.34(b).

Figure 25.32: NACA2412 in Mach number 0.95 flow at 0 degree angle of attack problem at $t = 1.8$ (a) Mach number (b) Pressure

25.1.12.2 Solid Objects in Sedov Explosion

Another problem for testing a stationary rigid body in a simulation is to consider the Sedov-like explosion in a chamber surrounded by a solid wall with holes. The wall is shown as red blocks with white boundary in 25.34(a) and 25.34(b). The simulation was done on a uniform Cartesian grid with 300 cells on each direction. Three holes in the wall subdivide it into four different stationary solid bodies in a square computational domain $[0, 1.5] \times [0, 1.5]$. The explosion goes off at the origin and generates shock waves inside the chamber. In later time, when the shock waves pass through the three holes in the wall, turbulence effects are triggered from the interaction between the fluid and the wall and enhance vortical fluid motions.

One important thing in this problem is to keep the given symmetry throughout the simulation. The flow symmetry across the diagonal direction is well preserved in 25.34(a) and 25.34(b).
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

(a) Mach number (b) Pressure

Figure 25.33: NACA2412 in Mach number 1.2 flow at 0 degree angle of attack problem at $t = 1.8$ (a) Mach number (b) Pressure

25.2 Magnetohydrodynamics Test Problems

The magnetohydrodynamics (MHD) test problems provided in this release can be found in source/\Simulation/SimulationMain/magnetoHD/. In order to setup an MHD problem users need to specify the magnetoHD path in a setup script. For instance, BrioWu problem can be setup by typing ./setup magnetoHD/BrioWu -auto -1d.

25.2.1 Brio-Wu MHD Shock Tube

The Brio-Wu MHD shock tube problem (Brio and Wu, 1988), magnetoHD/BrioWu, is a coplanar magnetohydrodynamic counterpart of the hydrodynamic Sod problem (Section 25.1.1). The initial left and right states are given by $\rho_l = 1, u_l = v_l = 0, p_l = 1, (B_y)_l = 1$; and $\rho_r = 0.125, u_r = v_r = 0, p_r = 0.1, (B_y)_r = -1$. In addition, $B_x = 0.75$ and $\gamma = 2$. This is a good problem to test wave properties of a particular MHD solver, because it involves two fast rarefaction waves, a slow compound wave, a contact discontinuity and a slow shock wave.

The conventional 800 point solution to this problem computed with FLASH is presented in Figure 25.35, Figure 25.36, Figure 25.37, Figure 25.38, and Figure 25.39. The figures show the distribution of density, normal and tangential velocity components, tangential magnetic field component and pressure at $t = 0.1$ (in non-dimensional units). As can be seen, the code accurately and sharply resolves all waves present in the solution. There is a small undershoot in the solution at $x \approx 0.44$, which results from a discontinuity-enhancing monotonicized centered gradient limiting function (LeVeque 1997). This undershoot can be easily removed if a less aggressive limiter, e.g. a minmod or a van Leer limiter, is used instead. This, however, will degrade the sharp resolution of other discontinuities.

The directionally splitting $\text{S}\text{W}\text{a}\text{v}\text{e}$ MHD solver with a second-order MUSCL-Hancock scheme was used for the results shown in this simulation. The $\text{StaggeredMesh}$ MHD solver can also be used for this Brio-Wu problem in one- and two-dimensions. However, in the latter case, the $\text{StaggeredMesh}$ solver only supports non-rotated setups for which a shock normal is parallel to the $x$-axis that initially intersects that axis at $x = 0.5$ (halfway across a box with unit dimensions). This limitation occurs in the $\text{StaggeredMesh}$ scheme because the currently released version of the FLASH code does not truly support physically correct boundary conditions for this rotated shock geometry.
25.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

Figure 25.34: The Sedov explosion in a chamber surrounded by a wall with holes. (a) Density plot at $t = 0.0$ sec. (b) Denstiy plot at $t = 0.1$ sec.

Figure 25.35: Density profile for the Brio-Wu shock tube problem.

25.2.1.1 Slowly moving shocks (SMS) issues in the Brio-Wu problem

Figure 25.40 clearly demonstrates that the conventional PPM reconstruction method fails to preserve monotonicities, shedding oscillations especially in the plateau near strong discontinuities such as the contact and right going slow MHD shock. In Fig. 25.41, Mach numbers are plotted with varying strengths of the transverse field $B_y$. The oscillations increase with an increase of $B_y$; the reason being that stronger $B_y$ introduces more transverse effect that resists shock propagation in the $x$ direction causing the shock to move slowly. This effect is clearly seen in the locations of the shock fronts (right going slow MHD shocks in this case), which remain closer to the initial location $x = 0.5$ when $B_y$ is stronger.

Results from using the upwind biased slope limiter for PPM are illustrated in Figures 25.42 – 25.45. The oscillation shedding found in the conventional PPM (e.g., Fig. 25.40) are significantly reduced in both density, and Mach number profiles. The overall qualitative solution behavior of using the upwind PPM.
approach, shown in Fig. 25.42, is very much similar to those from the 5th order WENO method as illustrated in Fig. 25.43. As shown in the closeup views in Fig. 25.44 and 25.45, the solutions with the upwind PPM slope limiter (blue curves) outperforms the conventional PPM method (red curves), and compares well with the WENO scheme (green curves). In fact, the upwind approach shows the most flat density plateau of all methods. Note that the SMS issue can be observed regardless of the choice of Riemann solvers and the dissipation mechanism in PPM (e.g., even with flattening on). 400 grid points were used.

25.2.2 Orszag-Tang MHD Vortex

The Orszag-Tang MHD vortex problem (Orszag and Tang, 1979), magetoHD/OrszagTang, is a simple two-dimensional problem that has become a classic test for MHD codes. In this problem a simple, non-random initial condition is imposed at time \( t = 0 \)

\[
\mathbf{V} = V_0 (-\sin(2\pi y), \sin(2\pi x), 0), \quad \mathbf{B} = B_0 (-\sin(2\pi y), \sin(4\pi x), 0), \quad (x, y) \in [0, 1]^2, \quad (25.17)
\]

where \( B_0 \) is chosen so that the ratio of the gas pressure to the RMS magnetic pressure is equal to \( 2\gamma \). In this setup the initial density, the speed of sound and \( V_0 \) are set to unity; therefore, the initial pressure \( p_0 = 1/\gamma \)
25.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

Figure 25.38: Normal velocity profile for the Brio-Wu shock tube problem.

Figure 25.39: Tangential velocity profile for the Brio-Wu shock tube problem.

and $B_0 = 1/\gamma$.

As the evolution time increases, the vortex flow pattern becomes increasingly complicated due to the nonlinear interactions of waves. A highly resolved simulation of this problem should produce two-dimensional MHD turbulence. Figure 25.46 and Figure 25.47 shows density and magnetic field contours at $t = 0.5$. As one can observe, the flow pattern at this time is already quite complicated. A number of strong waves have formed and passed through each other, creating turbulent flow features at all spatial scales.

The results were obtained using the directionally splitting $8$Wave MHD solver for this Orszag-Tang problem.

The 3D version are also shown in the below solved using the unsplit staggered mesh MHD solver.
Figure 25.40: Density (thick red) and Mach number (thick blue) at $t = 0.1$ of the Brio-Wu test with the conventional PPM's MC limiter. Thin black curves represent reference solutions using the PLM of MUSCL-Hancock scheme. Severe numerical oscillations are evident in the solution using the conventional PPM reconstruction on 400 grid points.

Figure 25.41: Mach numbers at $t = 0.1$ with varying $B_y$ from 0 to 1 with the conventional PPM's MC limiter. Curves in red, blue, green, purple, black, and cyan represent $B_y = 0, 0.2, 0.4, 0.6, 0.8$, and 1, respectively. Severe numerical oscillations are evident in the solution using the conventional PPM reconstruction on 400 grid points.
Figure 25.42: Density (red) and Mach number (blue) at $t = 0.1$ of the Brio-Wu test using the upwind biased PPM limiter.

Figure 25.43: Density (red) and Mach number (blue) at $t = 0.1$ of the Brio-Wu test using the 5th order WENO scheme.
Figure 25.44: Density closeup at $t = 0.1$ of the Brio-Wu test. The conventional PPM in red curve, the upwind PPM in blue curve, and the WENO scheme in green curve.

Figure 25.45: Mach number closeup at $t = 0.1$ of the Brio-Wu test. The conventional PPM in red curve, the upwind PPM in blue curve, and the WENO scheme in green curve.
Figure 25.46: Density contours in the Oszag-Tang MHD vortex problem at $t = 0.5$.

Figure 25.47: Magnetic field contours in the Oszag-Tang MHD vortex problem at $t = 0.5$. 
Figure 25.48: Density plots of a 3D version of the Orszag-Tang problem on a $128^3$ uniform grid resolution. (a) Density at $t = 0.2$  (b) Density at $t = 0.5$  (c) Density at $t = 0.7$  (d) Density at $t = 1.0$. 
25.2.3 Magnetized Accretion Torus

The magnetized accretion torus problem is based on the global magneto-rotational instability (MRI) simulations of Hawley (2000). It can be found under `magnetoHD/Torus`. We consider a magnetized torus of constant angular momentum (\( \Omega \propto r^{-2} \)), inside a normalized Paczyński-Wiita pseudo-Newtonian gravitational potential (Paczyński & Wiita, 1980) of the form

\[
\Phi = -\frac{1}{(R - 1)}, \quad R = \sqrt{r^2 + z^2}.
\]

The cylindrical computational domain, with lengths normalized at \( r_0 \), extends from 1.5 \( r_0 \) to 15.5 \( r_0 \) in the radial direction and from \(-7 r_0\) to \(7 r_0\) in the z direction. For this specific simulation we use seven levels of refinement, linear reconstruction with vanLeer limiter and the HLLC Riemann solver. The boundary conditions are set to outflow, except for the leftmost side where a diode-like condition is applied.

Assuming an adiabatic equation of state, the initial density profile of the torus is given by

\[
\frac{\Gamma P}{(\Gamma - 1) \rho} = C - \Phi - \frac{l_K^2}{2r^2},
\]

where the specific heats ratio is \( \Gamma = 5/3 \), \( l_K \) is the Keplerian angular momentum at the pressure maximum \( (r_{P\text{max}} = 4.7r_0) \) and \( C \) is an integration constant that specifies the outer surface of the torus, given its inner radius \( (r_{in} = 3r_0) \). The initial poloidal magnetic field is computed using the \( \phi \) component of the vector potential, \( A_\phi \propto \max(\rho - 5, 0) \), and normalized so as the initial minimum value of the plasma \( \beta = 2P/B^2 \) is equal to \( 10^2 \). The resulting field follows the contours of density, i.e. torus-like nested loops, and is embedded well within the torus.

We allow the system to evolve for \( t = 150 t_0 \). The torus is MRI unstable and after approximately one revolution accretion sets in. The strong shear generates an azimuthal field component and the angular momentum is redistributed. Due to the instability, filamentary structures form at the torus surface that account for its rich morphology (Figure 25.49). These results can be promptly compared to those in Hawley (2000) and Mignone et al. (2007).

Figure 25.49: Left: 3D rendering of the axisymmetric torus evolution. Right: Density logarithm of the magnetized accretion torus after 150 \( t_0 \). Superposed are the AMR levels and the mesh.
25.2.4 Magnetized Noh Z-pinch

In this test we consider the magnetized version of the classical Noh problem (Noh, 1987). It consists of a cylindrically symmetric implosion of a pressure-less gas: the gas stagnates at the symmetry axis and creates an outward moving accretion shock which propagates at a constant velocity. The self-similar analytic solution of this problem has been widely used as benchmark test for hydrodynamic codes, especially those targeting implosion.

Recently, Velikovich et al. (2012) have extended the original test problem to include an embedded azimuthal magnetic field, in accordance with the Z-pinch physics. In their study, they present a family of exact analytic solutions for which the values of primitive variables are finite everywhere, providing an excellent benchmark for Z-pinch and general MHD codes.

We perform this test in both cartesian and cylindrical geometries. The tests can be found respectively in \textit{magnetoHD/Noh} and \textit{magnetoHD/NohCylindrical}. For brevity we describe only the cylindrical initialization.

The cartesian can be easily recovered by projecting the solution onto the X-Y plane. A 3T MHD version of this test can also be found under \textit{magnetoHD/unitTest/NohCylindricalRagelike}.

The simulation is initialized in a computational box that spans \([0, 3]\) cm in the \(r\) and \(z\) directions, in cylindrical \((r - z)\) geometry. The leftmost boundary condition is set to axisymmetry, whereas the remaining boundaries are set to outflow (zero gradient). The initial condition in primitive variables is defined as

\[
\begin{align*}
\rho &= 3.1831 \times 10^{-5} r^2 \text{g/cm}^3, \\
v &= (-3.24101 \times 10^7, 0, 0) \text{cm/s}, \\
B &= (0, 6.35584 \times 10^5 r, 0) \text{gauss}, \\
P &= C B^2. \\
\end{align*}
\]

Since Godunov-type codes cannot run with zero pressure, we initialize \(P\) by choosing \(C = P/B^2 = 10^{-6}\), ensuring a magnetically dominated plasma.

The simulations are evolved for 30 ns, utilizing the unsplit solver and a Courant number of 0.8. We use 6 levels of refinement, corresponding to an equivalent resolution of 256 \(\times\) 256 zones. The reconstruction is piecewise-linear (second order) with characteristic limiting, whereas the Riemann solvers employed are the HLLD and Roe.

The resulting density profile is shown in Figure 25.50. The refinement closely follows the propagation of the discontinuity which reaches the analytically predicted location \((r = 0.3 \text{ cm})\) at \(t = 30\) ns. The lineouts for HLLD (green dots) and Roe (blue dots) shown on the right, display good agreement with the analytic solution (red line) and the discontinuity is sharply captured (resolved on two points). Our Figure 25.50 can be directly compared to Figures 2a and 3a of Velikovich et al. (2012).

25.2.5 MHD Rotor

The two-dimensional MHD rotor problem (Balsara and Spicer, 1999), \textit{magnetoHD/Rotor}, is designed to study the onset and propagation of strong torsional Alfvén waves, which is thereby relevant for star formation. The computational domain is a unit square \([0, 1] \times [0, 1]\) with non-reflecting boundary conditions on all four sides. The initial conditions are given by

\[
\begin{align*}
\rho(x, y) &= \begin{cases} 
10 & r \leq r_0 \\
1 + 9f(r) & r_0 < r < r_1 \\
1 & r \geq r_1 
\end{cases} \\
u(x, y) &= \begin{cases} 
-f(r)u_0(y - 0.5)/r_0 & r \leq r_0 \\
f(r)u_0(y - 0.5)/r & r_0 < r < r_1 \\
0 & r \geq r_1 
\end{cases} \\
v(x, y) &= \begin{cases} 
-f(r)u_0(x - 0.5)/r_0 & r \leq r_0 \\
f(r)u_0(x - 0.5)/r & r_0 < r < r_1 \\
0 & r \geq r_1 
\end{cases}
\end{align*}
\]
25.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

Figure 25.50: Magnetized Noh problem in cylindrical coordinates. Left: density snapshot along with AMR levels after 30 ns for the HLLD Riemann solver. Right: Lineouts of density close to the origin, \( r = [0, 0.6] \) cm, for HLLD and Roe, superposed on the analytic solution.

\[
p(x,y) = 1 \quad (25.23)
\]
\[
B_x(x,y) = \frac{5}{\sqrt{4\pi}} \quad (25.24)
\]
\[
B_y(x,y) = 0, \quad (25.25)
\]

where \( r_0 = 0.1, r_1 = 0.115, r = \sqrt{(x-0.5)^2 + (y-0.5)^2}, w = B_z = 0 \) and a taper function \( f(r) = (r_1 - r)/(r - r_0) \). The value \( \gamma = 1.4 \) is used. The initial set-up is occupied by a dense rotating disk at the center of the domain, surrounded by the ambient flow at rest with uniform density and pressure. The rapidly spinning rotor is not in an equilibrium state due to the centrifugal forces. As the rotor spins with the given initial rotating velocity, the initially uniform magnetic field in \( x \)-direction will wind up the rotor. The rotor will be wrapped around by the magnetic field, and hence start launching torsional Alfvén waves into the ambient fluid. The angular momentum of the rotor will be diminished in later times as the evolution time increases. The circular rotor will be progressively compressed into an oval shape by the build-up of the magnetic pressure around the rotor. The results shown in Figure 25.51 were obtained using the StaggeredMesh MHD solver using 6 refinement levels. The divergence free evolution of the magnetic fields are well preserved as illustrated in Figure 25.52.

25.2.6 MHD Current Sheet

The two-dimensional current sheet problem, magnetoHD/CurrentSheet, has recently been studied by Gardiner and Stone (2005) in ideal MHD regime. The two current sheets are initialized and therefore magnetic reconnections are inevitably driven. In the regions the magnetic reconnection takes place the magnetic flux approaches vanishingly small values, and the loss in the magnetic energy is converted into heat (thermal energy). This phenomenon changes the overall topology of the magnetic fields and hence affects the global magnetic configuration.

The square computational domain is given as \([0, 2] \times [0, 2]\) with periodic boundary conditions on all four
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

Figure 25.51: The Rotor problem at $t = 0.15$ (a) Density (b) Pressure (c) Mach number (d) Magnetic pressure.

sides. We initialize two current sheets in the following:

$$
B_y = \begin{cases} 
B_0 & 0 \leq x < 0.5 \\
-B_0 & 0.5 \leq x < 1.5 \\
B_0 & 1.5 \leq x \leq 2.0 
\end{cases},
$$

where $B_0 = 1$. The other magnetic field components $B_x, B_z$ are set to be zeros. The $x$ component of the velocity is given by $u = u_0 \sin 2\pi y$ with $u_0 = 0.1$, and all the other velocity components are initialized with zeros. The density is unity and the gas pressure $p = 0.1$.

The changes of the magnetic fields seed the magnetic reconnection and develop formations of magnetic islands along the two current sheets. The small islands are then merged into the bigger islands by continuously shifting up and down along the current sheets until there is one big island left in each current sheet.

The temporal evolution of the magnetic field lines from $t = 0.0$ to $t = 5.0$ are shown in 25.53(a) – 25.53(f) on a $256 \times 256$ uniform grid. In Figure 25.54 the same problem is resolved on an AMR grid with 6 refinement levels, showing the current density $j_z$ along with the AMR block structures at $t = 4.0$. The StaggeredMesh solver was used for this problem.
Figure 25.52: Divergence of magnetic fields using the StaggeredMesh solver at $t = 0.15$ for the Rotor problem.
Figure 25.53: The temporal evolutions of field lines for the MHD CurrentSheet problem. Equally spaced 60 contour lines are shown at time (a) $t = 0.0$ (b) $t = 1.0$ (c) $t = 2.0$ (d) $t = 3.0$ (e) $t = 4.0$ (f) $t = 5.0$. 
Figure 25.54:  Current density at \( t = 4.0 \) using the StaggeredMesh solver for the MHD CurrentSheet problem.
25.2.7 Field Loop

The 2D and 3D field loop advection problems (magnetohd/FieldLoop) are known to be stringent test cases in multidimensional MHD. In this test problem we consider a 2D advection of a weakly magnetized field loop traversing the computational domain diagonally. Details of the problem has been described in Gardiner and Stone (2005).

The computational domain is \([-1,1] \times [-0.5,0.5]\), with a grid resolution 256 \times 148, and doubly-periodic boundary conditions. With this rectangular grid cell, the flow is not symmetric in \(x\) and \(y\) directions because the field loop does not advect across each grid cell diagonally and hence the resulting fluxes are different in \(x\) and \(y\) directions. The density and pressure are unity everywhere and \(\gamma = 5/3\). The velocity fields are defined as,

\[
\mathbf{U} = u_0(\cos \theta, \sin \theta, 1)
\]

with the advection angle \(\theta\), given by \(\theta = \tan^{-1}(0.5) \approx 26.57^\circ\). For the choice of the initial advection velocity we set \(u_0 = \sqrt{5}\). The size of domain and other parameters were chosen such that the weakly magnetized field loop makes one complete cycle by \(t = 1\). It is important to initialize the magnetic fields to satisfy \(\nabla \cdot \mathbf{B} = 0\) numerically in order to avoid any initial nonzero error in \(\nabla \cdot \mathbf{B}\). As suggested in Gardiner and Stone (2005), the magnetic field components are initialized by taking the numerical curl of the \(z\)-component of the magnetic vector potential \(A_z\),

\[
B_x = \frac{\partial A_z}{\partial y}, \quad B_y = -\frac{\partial A_z}{\partial x},
\]

where

\[
A_z = \begin{cases} A_0(R - r) & r \leq R \\ 0 & \text{otherwise}. \end{cases}
\]

By using this initialization process, divergence-free magnetic fields are constructed with a maximum value of \(\nabla \cdot \mathbf{B}\) in the order of \(10^{-16}\) at the chosen resolution. The parameters in (25.29) are \(A_0 = 10^{-3}\) and a field loop radius \(R = 0.3\). This initial condition results in a very high plasma beta \(\beta = p/B_t = 2 \times 10^6\) for the inner region of the field loop. Inside the loop the magnetic field strength is very weak and the flow dynamics is dominated by the gas pressure.

The field loop advection is integrated to a final time \(t = 2\). The advection test is found to truly require the full multidimensional MHD approach (Gardiner and Stone, 2005, 2008; Lee and Deane, 2008). Since the field loop is advected at an oblique angle to the \(x\)-axis of the computational domain, the values of \(\partial B_x/\partial x\) and \(\partial B_y/\partial y\) are non-zero in general and their roles are crucial in multidimensional MHD flows. These terms, together with the multidimensional MHD terms \(\mathbf{A}_{B_x}\) and \(\mathbf{A}_{B_y}\), are explicitly included in the data reconstruction-evolution algorithm in the USM scheme (see Lee and Deane, 2008). During the advection a good numerical scheme should maintain: (a) the circular symmetry of the loop at all time: a numerical scheme that lacks proper numerical dissipation results in spurious oscillations at the loop, breaking the circular symmetry; (b) \(B_z = 0\) during the simulation: \(B_z\) will grow proportional to \(w\nabla \cdot \mathbf{B} \Delta t\) if a numerical scheme does not properly include multidimensional MHD terms.

From the results in Figure 25.55, the USM scheme maintains the circular shape of the loop extremely well to the final time step. The scheme successfully retains the initial circular symmetry and does not develop severe oscillations.

A variant 3D version of this problem (Gardiner and Stone, 2008) is also available, and is illustrated in Fig. 25.56. This problem is considered to be a particularly challenging test because the correct solution requires inclusion of the multidimensional MHD terms to preserve the in-plane dynamics. Otherwise, the failure in preserving the in-plane dynamics (i.e., growth in the out-of-plane component) results in erroneous behavior of the field loop. As shown in Fig. 25.56, the USM solver successfully maintains the circular shape of the field loop and maintains the out-of-plane component of the magnetic field to very small values over the domain. This figure compares very well with Fig. 2 in Gardiner and Stone, 2008.

25.2.8 3D MHD Blast

A 2D version of the MHD blast problem was studied by Zachary et al. (Zachary, Malagoli, and Colella, 1994) and we consider a variant 3D version of the MHD spherical blast wave problem here. This problem
25.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

Figure 25.55: The field loop advection problem using the StaggeredMesh solver at time $t = 2$ with the Roe Riemann solver. (a) $B_p$ with the MEC at $t = 2$. The color scheme between $2.32 \times 10^{-25}$ and $7.16 \times 10^{-7}$ was used. (b) Magnetic field lines with the MEC at $t = 2$. 20 contour lines of $A_z$ between $-2.16 \times 10^{-6}$ and $2.7 \times 10^{-4}$ are shown.

Figure 25.56: Thresholded images of the field loop advection problem at times $t = 0.0$, $1.5$, and $2.0$ using a uniform grid size $128 \times 128 \times 256$. 
leads to the formation and propagation of strong MHD discontinuities, relevant to astrophysical phenomena where the magnetic field energy has strong dynamical effects. With a numerical scheme that fails to preserve the divergence-free constraint, unphysical states could be obtained involving negative gas pressure because the background magnetic pressure increases the strength of magnetic monopoles.

This problem can be computed in various magnetized flow regimes by considering different magnetic field strengths. The computational domain is a square \([-0.5, 0.5] \times [-0.5, 0.5] \times [-0.5, 0.5]\) with a maximum refinement level 4. The explosion is driven by an over-pressurized circular region at the center of the domain with a radius \(r = 0.1\). The initial density is unity everywhere, and the pressure of the ambient gas is 0.1, whereas the pressure of the inner region is 1000. The strength of a uniform magnetic field in the \(x\)-direction are \(0, 50/\sqrt{4\pi}\) and \(100/\sqrt{4\pi}\). This initial condition results in a very low-\(\beta\) ambient plasma state, \(\beta = 2.513 \times 10^{-4}\). Through this low-\(\beta\) ambient state, the explosion emits almost spherical fast magneto-sonic shocks that propagate with the fastest wave speed. The flow has \(\gamma = 1.4\).

With this strong magnetic field strength, \(B_x = 100/\sqrt{4\pi}\), shown in Figure 25.57, the explosion now becomes highly anisotropic as shown in the pressure plot in Figure 25.57. The Figure shows that the displacement of gas in the transverse \(y\)-direction is increasingly inhibited and hydrodynamical shocks propagate in both positive and negative \(x\)-directions parallel to \(B_x\). This process continues until total pressure equilibrium is reached in the central region. This problem is also available in 2D setup.

25.3 Gravity Test Problems

25.3.1 Jeans Instability

The linear instability of self-gravitating fluids was first explored by Jeans (1902) in connection with the problem of star formation. The nonlinear phase of the instability is currently of great astrophysical interest, but the linear instability still provides a very useful test of the coupling of gravity to hydrodynamics in FLASH.

The Jeans problem allows one to examine the behavior of sinusoidal, adiabatic density perturbations in both the pressure-dominated and gravity-dominated limits. This problem uses periodic boundary conditions. The equation of state is that of a perfect gas. The initial conditions at \(t = 0\) are

\[
\begin{align*}
\rho(x) &= \rho_0 [1 + \delta \cos(k \cdot x)] \\
p(x) &= p_0 [1 + \gamma \delta \cos(k \cdot x)] \\
v(x) &= 0,
\end{align*}
\]

where the perturbation amplitude \(\delta \ll 1\). The stability of the perturbation is determined by the relationship between the wavenumber \(k \equiv |k|\) and the Jeans wavenumber \(k_J\), where \(k_J\) is given by

\[
k_J \equiv \frac{\sqrt{4\pi G \rho_0}}{c_0}, \tag{25.31}
\]

and where \(c_0\) is the unperturbed adiabatic sound speed

\[
c_0 = \sqrt{\frac{\gamma p_0}{\rho_0}} \tag{25.32}
\]

(Chandrasekhar 1961). If \(k > k_J\), the perturbation is stable and oscillates with frequency

\[
\omega = \sqrt{c_0^2 k^2 - 4\pi G \rho_0}; \tag{25.33}
\]

otherwise, it grows exponentially, with a characteristic timescale given by \(\tau = (i\omega)^{-1}\).

We checked the dispersion relation (25.33) for stable perturbations with \(\gamma = 5/3\) by fixing \(\rho_0\) and \(p_0\) and performing several runs with different \(k\). We followed each case for roughly five oscillation periods using a uniform grid in the box \([0, L]^2\). We used \(\rho_0 = 1.5 \times 10^7\) g cm\(^{-3}\) and \(p_0 = 1.5 \times 10^7\) dyn cm\(^{-2}\), yielding
25.3. GRAVITY TEST PROBLEMS

Figure 25.57: The MHD blast test at time $t = 0.01$ using the unsplit staggered mesh MHD solver. Density (top half) and magnetic pressure (bottom half) plots for three different strengths of $B_x$. (a) $B_x = 0$ (b) $B_x = 50/\sqrt{4\pi}$ (c) $B_x = 100/\sqrt{4\pi}$.

$k_J = 2.747$ cm$^{-1}$. The perturbation amplitude $\delta$ was fixed at $10^{-3}$. The box size $L$ is chosen so that $k_J$ is smaller than the smallest nonzero wavenumber that can be resolved on the grid

$$L = \frac{1}{2} \sqrt{\frac{\pi \gamma p_0}{G \rho_0^2}}. \quad (25.34)$$

This prevents roundoff errors at wavenumbers less than $k_J$ from being amplified by the physical Jeans instability. We used wavevectors $\mathbf{k}$ parallel to and at 45 degrees to the $x$-axis. Each test calculation used the multigrid Poisson solver together with its default settings.

The resulting kinetic, thermal, and potential energies as functions of time for one choice of $\mathbf{k}$ are shown
in Figure 25.58 together with the analytic solution, which is given in two dimensions by

\[
T(t) = \frac{\rho_0 \delta^2 |\omega|^2 L^2}{8k^2} [1 - \cos(2\omega t)]
\]

\[
U(t) - U(0) = -\frac{1}{8} \rho_0 c_0^2 \delta^2 L^2 [1 - \cos(2\omega t)]
\]  \( (25.35) \)

\[
W(t) = -\frac{\pi G \rho_0^2 \delta^2 L^2}{2k^2} [1 + \cos(2\omega t)].
\]

The figure shows that FLASH obtains the correct amplitude and frequency of oscillation. We computed the average oscillation frequency for each run by measuring the time interval required for the kinetic energy to undergo exactly ten oscillations. Figure 25.59 compares the resulting dispersion relation to (25.33). It can be seen from this plot that FLASH correctly reproduces (25.33). At the highest wave number \( (k = 100) \), each wavelength is resolved using only about 14 cells on a six-level uniform grid, and the average timestep (which depends on \( c_0, \Delta x, \text{ and } \Delta y \), and has nothing to do with \( k \)) turns out to be comparable to the oscillation period. Hence the frequency determined from the numerical solution for this value of \( k \) is somewhat more poorly determined than for the other runs. At lower wavenumbers, however, the frequencies are correct to less than 1%.

![Figure 25.58: Kinetic, internal, and potential energy versus time for a stable Jeans mode with \( k = 10.984 \). Points indicate numerical values found using FLASH 3.0 with a fixed four-level adaptive grid. The analytic solution for each form of energy is shown using a solid line.](image)

The additional runtime parameters supplied with the Jeans problem are listed in Table 25.10. This problem is configured to use the perfect-gas equation of state (gamma) with gamma set to 1.67 and is run in a two-dimensional unit box. The refinement marking routine (Grid_markRefineDerefine.F90) supplied with this problem refines blocks whose mean density exceeds a given threshold. Since the problem is not spherically symmetric, the multigrid Poisson solver should be used.
25.3. **GRAVITY TEST PROBLEMS**

Figure 25.59: Computed versus expected Jeans dispersion relation (for stable modes) found using FLASH 1.62 with a six-level uniform grid.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rho0</td>
<td>real</td>
<td>$1.5 \times 10^7$</td>
<td>Initial unperturbed density ($\rho_0$)</td>
</tr>
<tr>
<td>p0</td>
<td>real</td>
<td>$1.5 \times 10^7$</td>
<td>Initial unperturbed pressure ($p_0$)</td>
</tr>
<tr>
<td>amplitude</td>
<td>real</td>
<td>0.001</td>
<td>Perturbation amplitude ($\delta$)</td>
</tr>
<tr>
<td>lambdax</td>
<td>real</td>
<td>0.572055</td>
<td>Perturbation wavelength in $x$ direction ($\lambda_x = 2\pi/k_x$)</td>
</tr>
<tr>
<td>lambday</td>
<td>real</td>
<td>$1.0 \times 10^{10}$</td>
<td>Perturbation wavelength in $y$ direction ($\lambda_y = 2\pi/k_y$)</td>
</tr>
<tr>
<td>lambdaz</td>
<td>real</td>
<td>$1.0 \times 10^{10}$</td>
<td>Perturbation wavelength in $z$ direction ($\lambda_z = 2\pi/k_z$)</td>
</tr>
<tr>
<td>delta_ref</td>
<td>real</td>
<td>0.01</td>
<td>Refine a block if the maximum density contrast relative to $\rho_{ref}$ is greater than this</td>
</tr>
<tr>
<td>delta_deref</td>
<td>real</td>
<td>-0.01</td>
<td>Derefine a block if the maximum density contrast relative to $\rho_{ref}$ is less than this</td>
</tr>
<tr>
<td>reference_density</td>
<td>real</td>
<td>$1.5 \times 10^7$</td>
<td>Reference density for grid refinement ($\rho_{ref}$). Density contrast is used to determine which blocks to refine; it is defined as</td>
</tr>
</tbody>
</table>

$$\max_{\text{block}} \left\{ \frac{|\rho_{ijk}|}{\rho_{ref}} - 1 \right\}$$
25.3.2 Homologous Dust Collapse

The homologous dust collapse problem DustCollapse is used to test the ability of the code to solve self-gravitating problems in which the flow geometry is spherical and gas pressure is negligible. The problem was first described by Colgate and White (1966) and has been used by Mönchmeyer and Müller (1989) to test hydrodynamical schemes in curvilinear coordinates. We solve this problem using a 3D Cartesian grid.

The initial conditions consist of a uniform sphere of radius $r_0$ and density $\rho_0$ at rest. The pressure $p_0$ is taken to be constant and very small

$$p_0 \ll \frac{4\pi G}{\gamma} \rho_0^2 r_0^2.$$  \hspace{2cm} (25.36)

We refer to such a nearly pressureless fluid as ‘dust’. A perfect-gas equation of state is used, but the value of $\gamma$ is not significant. Outflow boundary conditions are used for the gas, while isolated boundary conditions are used for the gravitational field.

The collapse of the dust sphere is self-similar; the cloud should remain spherical with uniform density as it collapses. The radius of the cloud, $r(t)$, should satisfy

$$\left( \frac{8\pi G}{3} \rho_0 \right)^{1/2} t = \left( 1 - \frac{r(t)}{r_0} \right)^{1/2} \left( \frac{r(t)}{r_0} \right)^{1/2} + \sin^{-1} \left( 1 - \frac{r(t)}{r_0} \right)^{1/2}$$  \hspace{2cm} (25.37)

(Colgate & White 1966). Thus, we expect to test three things with this problem: the ability of the code to maintain spherical symmetry during an implosion (in particular, no block boundary effects should be evident); the ability of the code to keep the density profile constant within the cloud; and the ability of the code to obtain the correct collapse factor. The second of these is particularly difficult, because the edge of the cloud is very sharp and because the Cartesian grid breaks spherical symmetry most dramatically at the center of the grid, which is where all of the matter ultimately ends up.

Results of a DustCollapse run using FLASH 3.0 appear in Figure 25.60, which shows plots of density and the X component of velocity in menacing color scheme. The values are plotted at the end of the run from an X-Y plane in the center of the physical domain; density is in logarithmic scale. This run used a resolution of $128^3$, and the results were compared against a similar run using FLASH 2.5. We have also included figures from an earlier higher resolution run using FLASH2 which used $4^3$ top-level blocks and seven levels of refinement, for an effective resolution of $2048^3$. In both the runs, the multipole Poisson solver was used with a maximum multipole moment $\ell = 0$. The initial conditions used $\rho_0 = 10^9$ g cm$^{-3}$ and $r_0 = 6.5 \times 10^8$ cm. In Figure 25.61a, the density, pressure, and velocity are scaled by $2.43 \times 10^9$ g cm$^{-3}$, $2.08 \times 10^{17}$ dyn cm$^{-2}$, and $7.30 \times 10^9$ cm s$^{-1}$, respectively. In Figure 25.61b they are scaled by $1.96 \times 10^{11}$ g cm$^{-3}$, $2.08 \times 10^{17}$ dyn cm$^{-2}$, and $2.90 \times 10^{10}$ cm s$^{-1}$. Note that within the cloud, the profiles are very isotropic, as indicated by the small dispersion in each profile. Significant anisotropy is only present for low-density material flowing in through the Cartesian boundaries. In particular, it is encouraging that the velocity field remains isotropic all the way into the center of the grid; this shows the usefulness of refining spherically symmetric problems near $r = 0$. However, as material flows inward past refinement boundaries, small ripples develop in the density profile due to interpolation errors. These remain spherically symmetric but increase in amplitude as they are compressed. Nevertheless, they are still only a few percent in relative magnitude by the second frame. The other numerical effect of note is a slight spreading at the edge of the cloud. This does not appear to worsen significantly with time. If one takes the radius at which the density drops to one-half its central value as the radius of the cloud, then the observed collapse factor agrees with our expectation from (25.37). Overall our results, including the numerical effects, agree well with those of Mönchmeyer and Müller (1989).

This problem is configured to use the perfect-gas equation of state (gamma) with gamma set to 1.67 and is run in a three-dimensional box. The problem uses the specialized refinement marking routine supplied under the Grid interface of Grid_markRefineSpecialized which refines blocks containing the center of the cloud.
25.3. GRAVITY TEST PROBLEMS

Figure 25.60: XY plane of Density (a) and X component of Velocity (b) are shown at the center of the domain for the DustCollapse problem. The velocity is in normal scale, while density is logscale.

Figure 25.61: Density (black), pressure (red), and velocity (blue) profiles in the homologous dust collapse problem at (a) $t = 0.0368$ sec and (b) $t = 0.0637$ sec. The density, pressure, and velocity are scaled as discussed in the text.
25.3.3 Huang-Greengard Poisson Test

The PoisTest problem tests the convergence properties of the multigrid Poisson solver on a multidimensional, highly (locally) refined grid. This problem is described by Huang and Greengard (2000). The source function consists of a sum of thirteen two-dimensional Gaussians

\[ \rho(x, y) = \sum_{i=1}^{13} e^{-\sigma_i [(x-x_i)^2 + (y-y_i)^2]} , \]  

(25.38)

where the constants \( \sigma_i, x_i, \) and \( y_i \) are given in Table 25.11. The very large range of widths and ellipticities of these peaks forces the mesh structure to be highly refined in some places. The density field and block structure are shown for a 14-level mesh in Figure 25.62.

![Figure 25.62: Density field and block structure for a 14-level mesh applied to the Huang-Greengard PoisTest problem. The effective resolution of the mesh is 65,356².](image)

Table 25.11: Constants used in the poistest problem.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0.28125</td>
<td>0.5</td>
<td>0.3046875</td>
<td>0.3046875</td>
</tr>
<tr>
<td>( y_i )</td>
<td>0</td>
<td>0.09375</td>
<td>1</td>
<td>0.53125</td>
<td>0.53125</td>
<td>0.1875</td>
<td>0.125</td>
</tr>
<tr>
<td>( \sigma_i )</td>
<td>0.01</td>
<td>4000</td>
<td>20000</td>
<td>80000</td>
<td>16</td>
<td>360000</td>
<td>400000</td>
</tr>
</tbody>
</table>

Table 25.11: Constants used in the poistest problem.

<table>
<thead>
<tr>
<th>i</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>0.375</td>
<td>0.5625</td>
<td>-0.5</td>
<td>-0.125</td>
<td>0.296875</td>
<td>0.5234375</td>
</tr>
<tr>
<td>( y_i )</td>
<td>0.15625</td>
<td>-0.125</td>
<td>-0.703125</td>
<td>-0.703125</td>
<td>-0.609375</td>
<td>-0.78125</td>
</tr>
<tr>
<td>( \sigma_i )</td>
<td>2000</td>
<td>18200</td>
<td>128</td>
<td>49000</td>
<td>37000</td>
<td>18900</td>
</tr>
</tbody>
</table>

The PoisTest problem uses one additional runtime parameter sim.smlRho, the smallest allowed value of density. Runtime parameters from the Multigrid unit (both Gravity and GridSolvers) are relevant; see Section 8.10.2.6.
25.3.4 MacLaurin

The gravitational potential at the surface of, and inside a homogeneous spheroid called a “MacLaurin spheroid” is expressible in terms of analytical functions. This handy result was first determined by MacLaurin (1801), and later summarized by, amongst others, Chandrasekhar (1989). These properties allow validation of the FLASH4 gravitational solvers against the analytical solutions.

As a test case, an oblate \((a_1 = a_2 > a_3)\) Maclaurin spheroid, of a constant density \(\rho = 1\) in the interior, and \(\rho = \epsilon \to 0\) outside (in FLASH4 smlrho is used). The spheroid is motionless and in hydrostatic equilibrium. The gravitational potential of such object is analytically calculable, and is:

\[
\phi(x) = \pi G \rho \left[ 2A_1 a_1^2 - A_1 (x^2 + y^2) + A_3 (a_3^2 - z^2) \right],
\]

for a point inside the spheroid. Here

\[
A_1 = \frac{\sqrt{1 - e^2}}{e^3} \sin^{-1} e - \frac{1 - e^2}{e^2},
\]

\[
A_3 = \frac{2}{e^2} - \frac{2\sqrt{1 - e^2}}{e^3} \sin^{-1} e,
\]

where \(e\) is the ellipticity of a spheroid:

\[
e = \sqrt{1 - \left( \frac{a_3}{a_1} \right)^2}.
\]

For a point outside the spheroid, potential is:

\[
\phi(x) = \frac{2n a_3}{e^2} \pi G \rho \left[ a_1 \epsilon \tan^{-1} h - \frac{1}{2} \left( (x^2 + y^2) \left( \tan^{-1} h - \frac{h}{1 + h^2} \right) + 2z^2 (h - \tan^{-1} h) \right) \right],
\]

where

\[
h = \frac{a_1 \epsilon}{\sqrt{a_3^2 + \lambda}},
\]

and \(\lambda\) is the positive root of the equation

\[
\frac{x^2}{a_1^2 + \lambda} + \frac{y^2}{a_2^2 + \lambda} + \frac{z^2}{a_3^2 + \lambda} = 1.
\]

This test is also useful because the spheroid has spherical symmetry in the X–Y plane, but also lack of such symmetry in X–Z and Y–Z planes. The density distribution of the spheroid is shown in Equation 25.3.4. Spherical symmetry is simple to reproduce with a solution using multipole expansion. However, the non-symmetric solution requires an infinite number of multipole moments, while the code calculates solution up to a certain \(l_{\text{max}}\), specified by the user as runtime parameter \(\text{mpole}_l_{\text{max}}\). The error is thus expected to be dominated by the first non-zero term in the remainder of expansion. Also, the solution for any point inside the spheroid is the sum of monopole and dipole moments.

The simulation is calculated on a Maclaurin spheroid with eccentricity \(e = 0.9\); several other values for eccentricity were tried with results qualitatively the same. All tests used 3D Cartesian coordinates. The gravitational potential is calculated on an adaptive mesh, and the relative error is investigated:

\[
\epsilon = \left| \frac{\phi_{\text{analytical}} - \phi_{\text{FLASH}}}{\phi_{\text{analytical}}} \right|
\]

from zone to zone.

As expected, increasing spatial resolution improves the solution quality, but here we focus on how the solution depends on the choice of \(l_{\text{max}}\), the cutoff \(l\) in (8.15). In Figure 25.64–25.64 the gravitational potential for the Maclaurin spheroid, the FLASH4 solution, and relative errors for several \(l_{\text{max}}\)’s are shown. A similar figure produced for \(l_{\text{max}} = 1\) shows no difference from Figure 25.64, indicating that the last dipole term in the multipole expansion does not contribute to the accuracy of the solution but does increase computational...
cost. Because gravity sources are all of the same sign, and the symmetry of the problem, all odd-\( l \) moments are zero: reasonable, physically motivated values for setting \( m_{\text{pole}} l_{\text{max}} \) should be an even number.

In the X–Y plane, where the solution is radially symmetric, the first monopole term is enough to qualitatively capture the correct potential. As expected, the error is the biggest on the spheroid boundary, and decreases both outwards and inwards. Increasing the maximum included moment reduces errors. However, in other non-symmetric planes, truncating the potential to certain \( l_{\text{max}} \) leads to an error whose leading term will be the spherical harmonic of order \( l_{\text{max}} + 2 \), as can be nicely seen in the lower right sections of Figure 25.64 – 25.66. Increasing \( l_{\text{max}} \) reduces the error, but also increases the required time for computation. This computational increase is not linear because of the double sum in (8.17). Luckily, convergence is rather fast, and already for \( l_{\text{max}} = 4 \), there are only few zones with relative error bigger than 1%, while for the most of the computational domain the error is several orders of magnitude less.

Figure 25.63: Density of the MacLaurin spheroid (left X–Y plane, right Y–Z plane) with ellipticity \( e = 0.9 \). The FLASH4 block structure is shown on top.

Table 25.12: Minimal and maximal relative error in all zones of the simulation, calculated using (25.46). Last row is approximate time for one full timestep (gravity only).

<table>
<thead>
<tr>
<th>( l_{\text{max}} )</th>
<th>( \min(\epsilon) )</th>
<th>( \max(\epsilon) )</th>
<th>relative ( L_{\text{in}}^{2} ) norm</th>
<th>relative ( L_{\text{out}}^{2} ) norm</th>
<th>approx. time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 4.5 \times 10^{-6} )</td>
<td>0.182</td>
<td>( 7.1 \times 10^{-2} )</td>
<td>( 6.8 \times 10^{-2} )</td>
<td>9.8</td>
</tr>
<tr>
<td>1</td>
<td>( 4.5 \times 10^{-6} )</td>
<td>0.182</td>
<td>( 7.1 \times 10^{-2} )</td>
<td>( 6.8 \times 10^{-2} )</td>
<td>14.5</td>
</tr>
<tr>
<td>2</td>
<td>( 9.8 \times 10^{-6} )</td>
<td>0.062</td>
<td>( 1.4 \times 10^{-2} )</td>
<td>( 1.7 \times 10^{-2} )</td>
<td>34.7</td>
</tr>
<tr>
<td>4</td>
<td>( 1.0 \times 10^{-8} )</td>
<td>0.026</td>
<td>( 4.0 \times 10^{-3} )</td>
<td>( 5.0 \times 10^{-3} )</td>
<td>55.4</td>
</tr>
<tr>
<td>6</td>
<td>( 6.1 \times 10^{-9} )</td>
<td>0.013</td>
<td>( 1.6 \times 10^{-3} )</td>
<td>( 2.5 \times 10^{-3} )</td>
<td>134.9</td>
</tr>
<tr>
<td>8</td>
<td>( 7.8 \times 10^{-9} )</td>
<td>0.007</td>
<td>( 8.7 \times 10^{-4} )</td>
<td>( 1.2 \times 10^{-3} )</td>
<td>210.2</td>
</tr>
<tr>
<td>10</td>
<td>( 6.7 \times 10^{-9} )</td>
<td>0.004</td>
<td>( 5.5 \times 10^{-4} )</td>
<td>( 7.0 \times 10^{-4} )</td>
<td>609.7</td>
</tr>
</tbody>
</table>
Figure 25.64: Maclaurin spheroid: $l_{max} = 0$, 6 refinement levels. Left column is X–Y plane, cut through $z=0.5$, right column is Y–Z plane cut through $x=0.5$. From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.
Figure 25.65: Maclaurin spheroid: \( l_{max} = 2 \), 6 refinement levels. Left column is X–Y plane, cut through \( z=0.5 \), right column is Y–Z plane cut through \( x=0.5 \). From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.
Figure 25.66: Maclaurin spheroid: $l_{max} = 10$, 6 refinement levels. Left column is X–Y plane, cut through $z=0.5$, right column is Y–Z plane cut through $x=0.5$. From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.


25.4 Particles Test Problems

These problems are primarily designed to test the functioning of the particle tracking routines within FLASH4.

25.4.1 Two-particle Orbit

The Orbit problem tests the mapping of particle positions to gridded density fields, the mapping of gridded potentials onto particle positions to obtain particle forces, and the time integration of particle motion. The initial conditions consist of two particles of unit mass and separation $r_0$ located at positions $(x, y, z) = (0.5(L_x \pm r_0), 0.5L_y, 0.5L_z)$, where $(L_x, L_y, L_z)$ are the dimensions of the computational volume. The initial particle velocity vectors are parallel to the $y$-axis and have magnitude

$$|v| = \sqrt{\frac{2GM}{r_0}},$$

(25.47)

if a constant gravitational field due to a point mass $M$ at $(0.5L_x, 0.5L_y, 0.5L_z)$ is employed, or

$$|v| = \frac{1}{2} \sqrt{\frac{2G}{r_0}},$$

(25.48)

if the particles are self-gravitating. The correct behavior is for the particles to orbit the center of the grid in a circle with constant velocity. Figure 25.67 shows a typical pair of particle trajectories for this problem.

![Particle trajectories in the Orbit test problem for a 3D grid at a fixed refinement level of 2. There is no motion along the z-axis.](image)

Figure 25.67: Particle trajectories in the Orbit test problem for a 3D grid at a fixed refinement level of 2. There is no motion along the z-axis.

No specific gravity unit is required by the problem configuration file, because the problem is intended to be run with either a fixed external field or the particles’ own field. If the particles are to orbit in an external field (\texttt{ext\_field} = \texttt{true}.), the field is assumed to be a central point-mass field (\texttt{physics/Gravity/GravityMain/PointMass}), and the parameters for that unit should be assigned appropriate values. If the particles are self-gravitating (\texttt{ext\_field} = \texttt{false}.), the \texttt{physics/Gravity/GravityMain/Poisson}
unit should be included in the code, and a Poisson solver that supports isolated boundary conditions should be used (grav_boundary_type = "isolated").

In either case, long-range forces for the particles must be turned on, or else they will not experience any accelerations at all. This can be done using the particle-mesh method by including the unit

```
Particles/ParticlesMain/active/longRange/gravity/ParticleMesh.
```

### FLASH Transition

Although the Multipole solver can work with the Orbit problem, the solutions are very poor. We strongly recommend the use of Multigrid solver with this problem.

As of FLASH 2.1 both the multigrid and multipole solvers support isolated boundary conditions. This problem should be run in three dimensions.

### Grid interpolation

The FLASH2 user guide recommends that this problem be run with conservative, quadratic interpolants (such as mesh/amr/paramesh2.0/quadratic_cartesian) and monotonicity enforcement turned off (monotone = .false.). In FLASH4, you should use the default 2nd order monotonic interpolation scheme (see Section 8.6.2) in PARAMESH 4.

The two-particle orbit problem uses the runtime parameters listed in Table 25.13 in addition to the regular ones supplied with the code.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>separation</td>
<td>real</td>
<td>0.5</td>
<td>Initial particle separation ($r_0$)</td>
</tr>
<tr>
<td>ext_field</td>
<td>logical</td>
<td>.false.</td>
<td>Whether to make the particles self-gravitating or to have them orbit in an external potential. In the former case GravityMain/Poisson should be used; in the latter, GravityMain/PointMass.</td>
</tr>
</tbody>
</table>

### 25.4.2 Zel’dovich Pancake

The cosmological pancake problem (Zel’dovich 1970), Pancake, provides a good simultaneous test of the hydrodynamics, particle dynamics, Poisson solver, and cosmological expansion modules. Analytic solutions well into the nonlinear regime are available for both N-body and hydrodynamical codes (Anninos & Norman 1994), permitting an assessment of the code’s accuracy. After caustic formation the problem provides a stringent test of the code’s ability to track thin, poorly resolved features and strong shocks using most of the basic physics needed for cosmological problems. Also, as pancakes represent single-mode density perturbations, coding this test problem is useful as a basis for creating more complex cosmological initial conditions.

We set the initial conditions for the pancake problem in the linear regime using the analytic solution given by Anninos and Norman (1994). In a universe with $\Omega_0 = 1$ at redshift $z$, a perturbation of wavenumber...
\( k \) which collapses to a caustic at redshift \( z_c < z \) has comoving density and velocity given by

\[
\rho(x_c; z) = \tilde{\rho} \left[ 1 + \frac{1 + z_c}{1 + z} \cos(kx) \right]^{-1}
\]

\[
v(x_c; z) = -H_0 (1 + z)^{1/2} \frac{1 + z_c}{k} \sin(kx) \frac{1 + z}{k},
\]

where \( \tilde{\rho} \) is the comoving mean density. Here \( x_c \) is the distance of a point from the pancake midplane, and \( x \) is the corresponding Lagrangian coordinate, found by iteratively solving

\[
x_c = x - \frac{1 + z_c}{1 + z} \frac{\sin(kx)}{k}.
\]

The temperature solution is determined from the density under the assumption that the gas is adiabatic with ratio of specific heats \( \gamma \):

\[
T(x_c; z) = (1 + z)^2 T_{\text{fid}} \left[ \left( \frac{1 + z_{\text{fid}}}{1 + z} \right)^3 \frac{\rho(x_c; z_{\text{fid}})}{\rho(x_c; z)} \right]^{\gamma^{-1}}.
\]

The mean temperature \( T_{\text{fid}} \) is specified at a redshift \( z_{\text{fid}} \).

Dark matter particles are initialized using the same solution as the gas. The Lagrangian coordinates \( x \) are assigned to lie on a uniform grid. The corresponding perturbed coordinates \( x_c \) are computed using (25.51). Particle velocities are assigned using (25.50).

At caustic formation \((z = z_c)\), planar shock waves form in the gas on either side of the pancake midplane and begin to propagate outward. A small region at the midplane is left unshocked. Immediately behind the shocks, the comoving density and temperature vary approximately as

\[
\rho(x_c; z) \approx \tilde{\rho} \frac{18}{(kx)^2} \frac{(1 + z_c)^3}{(1 + z)^3}
\]

\[
T(x_c; z) \approx \frac{\mu H_0^2}{6 k_B k^2} (1 + z_c) (1 + z)^2 (kx)^2.
\]

At the midplane, which undergoes adiabatic compression, the comoving density and temperature are approximately

\[
\rho_{\text{center}} \approx \tilde{\rho} \left[ \frac{1 + z_{\text{fid}}}{1 + z} \right]^3 \left[ \frac{3 H_0^2 \mu}{k_B T_{\text{fid}} k^2} \frac{(1 + z_c)^4 (1 + z)^5}{1 + z_{\text{fid}}} \right]^{1/\gamma}
\]

\[
T_{\text{center}} \approx \frac{3 H_0^2 \mu}{k_B k^2} (1 + z)^2 (1 + z_c)^4 \frac{\tilde{\rho}}{\rho_{\text{center}}}
\]

An example FLASH calculation of the post-caustic gas solution appears in Figure 25.68.

Because they are collisionless, the particles behave very differently than the gas. As particles accelerate toward the midplane, their phase profile develops a backwards “S” shape. At caustic formation the velocity becomes multivalued at the midplane. The region containing multiple streams grows in size as particles pass through the midplane. At the edges of this region (the caustics, or the inflection points of the “S”), the particle density is formally infinite, although the finite force resolution of the particles keeps the height of these peaks finite. Some of the particles that have passed through the midplane fall back and form another pair of caustics, twisting the phase profile again. Because each of these secondary caustics contains five streams of particles rather than three, the second pair of density peaks are higher than the first pair. This caustic formation process repeats arbitrarily many times in the analytic solution. In practice, the finite number of particles and the finite force resolution limit the number of caustics that are observed. An example FLASH calculation of the post-caustic particle solution appears in Figure 25.69.

The 2D pancake problem in FLASH4 uses the runtime parameters listed in Table 25.14 in addition to the regular ones supplied with the code.

This problem uses periodic boundary conditions and is intrinsically one-dimensional, but it can be run using Cartesian coordinates in 1D, 2D, or 3D, with the pancake midplane tilted with respect to the coordinate axes if desired.
25.4. PARTICLES TEST PROBLEMS

Figure 25.68: Example solution for the gas in a mixed particle/gas Zel’dovich Pancake problem. A comoving wavelength \( \lambda = 10 \) Mpc, caustic redshift \( z_c = 5 \), fiducial redshift \( z_{fid} = 200 \), and fiducial temperature \( T_{fid} = 550 \) K were used together with a Hubble constant of \( 50 \) km s\(^{-1}\) Mpc\(^{-1}\). The cosmological model was flat with a baryonic fraction of 0.15. Results are shown for redshift \( z = 0 \). An adaptive mesh with an effective resolution of 1024 cells was used. Other parameters for this run were as described in the text. The distance \( x \) is measured from the pancake midplane. (a) Gas density. (b) Gas temperature. (c) Gas velocity.

Figure 25.69: Example solution for the dark matter in a mixed particle/gas Zel’dovich pancake. Perturbation and cosmological parameters were the same as in Figure 25.68. Results are shown for redshift \( z = 0 \). An adaptive mesh with an effective resolution of 1024 cells was used. The number of particles used was 8192. Other parameters for this run were as described in the text. Distance \( x \) is measured from the pancake midplane. (a) Dark matter density. (b) Dark matter phase diagram showing particle positions \( x \) and velocities \( v \).

Table 25.14: Runtime parameters used with the 2D pancake test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda</td>
<td>real</td>
<td>( 3.0857 \times 10^{25} )</td>
<td>Wavelength of the initial perturbation ((2\pi/k))</td>
</tr>
<tr>
<td>zcaustic</td>
<td>real</td>
<td>5.</td>
<td>Redshift at which pancake forms a caustic ((z_c))</td>
</tr>
<tr>
<td>Tfiducial</td>
<td>real</td>
<td>550.</td>
<td>Fiducial gas temperature ((T_{fid}))</td>
</tr>
<tr>
<td>zf fiducial</td>
<td>real</td>
<td>200.</td>
<td>Redshift at which gas temperature is (T_{fid}) ((z_{fid}))</td>
</tr>
<tr>
<td>xangle</td>
<td>real</td>
<td>0.</td>
<td>Angle made by pancake normal with the (x)-axis (degrees)</td>
</tr>
<tr>
<td>yangle</td>
<td>real</td>
<td>90.</td>
<td>Angle made by pancake normal with the (y)-axis (degrees)</td>
</tr>
<tr>
<td>pt numX</td>
<td>integer</td>
<td>128</td>
<td>Number of particles along (x)-side of initial particle “grid”</td>
</tr>
</tbody>
</table>
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

25.4.3 Modified Huang-Greengard Poisson Test

The PoisParticles problem is designed to generate a refined grid from the distribution of particles in the computational domain. In other words, create a grid which is more refined in places where there is a clustering of particles. It is a modified form of the PoisTest simulation described in Section 25.3.3. Recall that the PoisTest problem involves the creation of a highly refined grid, which is used to test grid refinement.

In the PoisParticles problem, the density stored in the grid is used as an indicator of where to create new particles. Here, the number of particles created in each region of the grid is proportional to the grid density, i.e., more particles are created in regions where there is a high density. Each new particle is assigned a mass, which is taken from the density in the grid, so that mass is conserved.

The flash.par parameters shown in Table 25.15 specify that the grid should refine until at most 5 particles exist per block. This creates a refined grid similar to the PoisTest problem.

Table 25.15: Runtime parameters used with the PoisParticles test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refine_on_particle_count</td>
<td>logical</td>
<td>.true.</td>
<td>On/Off flag for refining the grid according to particle count.</td>
</tr>
<tr>
<td>max_particles_per_blk</td>
<td>integer</td>
<td>5</td>
<td>Grid refinement criterion which specifies maximum number of particles per block.</td>
</tr>
</tbody>
</table>

25.5 Burn Test Problem

25.5.1 Cellular Nuclear Burning

The Cellular Nuclear Burning problem is used primarily to test the function of the Burn simulation unit. The problem exhibits regular steady-state behavior and is based on one-dimensional models described by Chappman (1899) and Jouguet (1905) and Zel’ dovich (Ostriker 1992), von Neumann (1942), and Doring (1943). This problem is solved in two dimensions. A complete description of the problem can be found in a recent paper by Timmes, Zingale et al.(2000).

A 13 isotope α-chain plus heavy-ion reaction network is used in the calculations. A definition of what we mean by an α-chain reaction network is prudent. A strict α-chain reaction network is only composed of \((\alpha, \gamma)\) and \((\gamma, \alpha)\) links among the 13 isotopes \(^{4}\text{He}, \text{^{12}}\text{C}, \text{^{16}}\text{O}, \text{^{20}}\text{Ne}, \text{^{24}}\text{Mg}, \text{^{28}}\text{Si}, \text{^{32}}\text{S}, \text{^{36}}\text{Ar}, \text{^{40}}\text{Ca}, \text{^{44}}\text{Ti}, \text{^{48}}\text{Cr}, \text{^{52}}\text{Fe}, \text{^{56}}\text{Ni}\). It is essential, however, to include \((\alpha, p)(p, \gamma)\) and \((\gamma, p)(p, \alpha)\) links in order to obtain reasonably accurate energy generation rates and abundance levels when the temperature exceeds \(\sim 2.5 \times 10^9 \text{ K}\). At these elevated temperatures the flows through the \((\alpha, p)(p, \gamma)\) sequences are faster than the flows through the \((\alpha, \gamma)\) channels. An \((\alpha, p)(p, \gamma)\) sequence is, effectively, an \((\alpha, \gamma)\) reaction through an intermediate isotope. In our α-chain reaction network, we include 8 \((\alpha, p)(p, \gamma)\) sequences plus the corresponding inverse sequences through the intermediate isotopes \(^{27}\text{Al}, \text{^{31}}\text{P}, \text{^{35}}\text{Cl}, \text{^{39}}\text{K}, \text{^{43}}\text{Sc}, \text{^{47}}\text{V}, \text{^{51}}\text{Mn}, \text{ and } \text{^{55}}\text{Co}\) by assuming steady state proton flows.

The two-dimensional calculations are performed in a planar geometry of size 256.0 cm by 25.0 cm. The initial conditions consist of a constant density of \(10^7 \text{ g cm}^{-3}\), temperature of \(2 \times 10^8 \text{ K}\), composition of pure carbon \(X(\text{^{12}}\text{C}) = 1\), and material velocity of \(v_x = v_y = 0 \text{ cm s}^{-1}\). Near the \(x=0\) boundary the initial conditions are perturbed to the values given by the appropriate Chapman-Jouguet solution: a density of \(4.236 \times 10^7 \text{ g}\)
cm$^{-3}$, temperature of $4.423 \times 10^9$ K, and material velocity of $v_x = 2.876 \times 10^8$ cm s$^{-1}$. Choosing different values or different extents of the perturbation simply change how long it takes for the initial conditions to achieve a near ZND state, as well as the block structure of the mesh. Each block contains 8 grid points in the x-direction, and 8 grid points in the y-direction. The default parameters for cellular burning are given in Table 25.16.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xhe4</td>
<td>real</td>
<td>0.0</td>
<td>Initial mass fraction of He4</td>
</tr>
<tr>
<td>xc12</td>
<td>real</td>
<td>1.0</td>
<td>Initial mass fraction of C12</td>
</tr>
<tr>
<td>xo16</td>
<td>real</td>
<td>0.0</td>
<td>Initial mass fraction of O16</td>
</tr>
<tr>
<td>rhoAmbient</td>
<td>real</td>
<td>$1 \times 10^7$</td>
<td>Density of cold upstream material.</td>
</tr>
<tr>
<td>tempAmbient</td>
<td>real</td>
<td>$2 \times 10^8$</td>
<td>Temperature of cold upstream material.</td>
</tr>
<tr>
<td>velxAmbient</td>
<td>real</td>
<td>0.0</td>
<td>X-velocity of cold upstream material.</td>
</tr>
<tr>
<td>rhoPerturb</td>
<td>real</td>
<td>$4.236 \times 10^7$</td>
<td>Density of the post shock material.</td>
</tr>
<tr>
<td>tempPerturb</td>
<td>real</td>
<td>$4.423 \times 10^9$</td>
<td>Temperature of the post shock material.</td>
</tr>
<tr>
<td>velxPerturb</td>
<td>real</td>
<td>$2.876 \times 10^8$</td>
<td>X-velocity of the post shock material.</td>
</tr>
<tr>
<td>radiusPerturb</td>
<td>real</td>
<td>25.6</td>
<td>Distance below which the perturbation is applied.</td>
</tr>
<tr>
<td>xCenterPerturb</td>
<td>real</td>
<td>0.0</td>
<td>X-position of the origin of the perturbation</td>
</tr>
<tr>
<td>yCenterPerturb</td>
<td>real</td>
<td>0.0</td>
<td>Y-position of the origin of the perturbation</td>
</tr>
<tr>
<td>zCenterPerturb</td>
<td>real</td>
<td>0.0</td>
<td>Z-position of the origin of the perturbation</td>
</tr>
<tr>
<td>usePseudo1d</td>
<td>logical</td>
<td>.false.</td>
<td>Defaults to a spherical configuration. Set to .true. if you want to use a 1d configuration, that is copied in the y and z directions.</td>
</tr>
<tr>
<td>noiseAmplitude</td>
<td>real</td>
<td>$1.0 \times 10^{-2}$</td>
<td>Amplitude of the white noise added to the perturbation.</td>
</tr>
<tr>
<td>noiseDistance</td>
<td>real</td>
<td>5.0</td>
<td>The distance above the starting radius to which white noise is added.</td>
</tr>
</tbody>
</table>

The initial conditions and perturbation given above ignite the nuclear fuel, accelerate the material, and produce an over-driven detonation that propagates along the x-axis. The initially over-driven detonation is damped to a near ZND state on short time-scale. After some time, which depends on the spatial resolution and boundary conditions, longitudinal instabilities in the density cause the planar detonation to evolve into a complex, time-dependent structure. Figure 25.70 shows the pressure field of the detonation after $1.26 \times 10^{-7}$ s. The interacting transverse wave structures are particularly vivid, and extend about 25 cm behind the shock front. Figure 25.71 shows a close up of this traverse wave region. Periodic boundary conditions are used at the walls parallel to the y-axis while reflecting boundary conditions were used for the walls parallel to the x-axis.
Figure 25.70: Steady-state conditions of the \textit{Cellular} nuclear burn problem.
Figure 25.71: Close-up of the detonation front in steady-state for the Cellular nuclear burn problem.
25.6 RadTrans Test Problems

There are currently two simulations included in FLASH which involve the RadTrans unit with multigroup diffusion. These simulations are not comparisons to analytic solutions - instead they are simple tests designed to demonstrate how the correct usage of the multigroup diffusion capability in FLASH.

25.6.1 Infinite Medium, Energy Equilibration

The simulation directory for this test is MGDInfinite. This is a fairly simple simulation which has no spatial gradients. Separate ion/electron/radiation temperatures are set throughout the domain initially. Over time, the temperatures should approach one another. The electrons and the radiation field exchange energy through emission and absorption, and the HeatExchange unit controls the rate at which the electrons and ions equilibrate with one another. The radiation field is represented using four radiation energy groups.

A sample setup line is:

```
./setup -auto MGDInfinite -1d +hdf5typeio +mtmmmt +mgd +uhd3t \
 -with-unit=physics/materialProperties/Opacity/OpacityMain/Constant \n species=be,poli,xe mgd_meshgroups=4
```

Figure 25.72 shows the temperatures as a function of time.

![Figure 25.72: Temperatures as a function of time for the Infinite medium, Energy Equilibration simulation](image)

25.6.2 Radiation Step Test

The simulation directory for this test is MGDStep. This simulation involves four group radiation diffusion and electron conduction. A constant opacity is used in this simulation. The transport opacity is set to a very small value to simulate a vacuum. This is 1D test where the initial electron and radiation temperatures are discontinuous. Flux limiters are used for both the electrons and radiation. Over time, energy flows from
the hotter region to the colder region. At the same time, the radiation temperature decreases as energy is absorbed by the electrons. Figure 25.73 shows the final temperature profiles for this simulation. There is a relatively sharp drop off for each curve caused by the use of flux limiters. The drop off can be made sharper by increasing the number cells used in the simulation.

Figure 25.73: Radiation and electron temperatures as a function of position for the MGDStep at 20 ps.
25.7 Other Test Problems

25.7.1 The non-equilibrium ionization test problem

The **neitest** problem tests the ability of FLASH to calculate non-equilibrium ionization (NEI) ion abundances. It simulates a stationary plasma flow through a temperature step profile. The solutions were checked using an independent stationary code based on a fifth order Runge–Kutta method with adaptive stepsize control by step-doubling (see Orlando *et al.* (1999)).

The test assumes a plasma with a mass density of $2 \times 10^{-16}$ gm cm$^{-3}$ flowing with a constant uniform velocity of $3 \times 10^5$ cm s$^{-1}$ through a temperature step between $10^4$ K and $10^6$ K (cf. Figure 25.74). The plasma is in ionization equilibrium before going through the jump in the region at $T = 10^4$ K. The population fractions in equilibrium are obtained from the equations

\[
[n_i^Z]_{eq} S_i^Z = [n_{i+1}^Z]_{eq} \alpha_{i+1}^Z \quad (i = 1, \ldots, l_Z - 1)
\]  

(25.55)

\[
\sum_{i=1}^{l_Z} [n_i^Z]_{eq} = A_Z n_p
\]

(25.56)

The presence of a temperature jump causes a strong pressure difference, which in turn should cause significant plasma motions. Since the purpose is to test the NEI module, it is imposed that the pressure difference does not induce any plasma motion and, to this end, the hydro variables (namely, $T$, $\rho$, $v$) are not updated. In practice, the continuity equations are solved with uniform density and velocity, while the momentum and energy equations are ignored.

Figure 25.75 shows the population fractions for the 12 most abundant elements in astrophysical plasmas derived with the stationary code (Orlando *et al.* (1999)). The out of equilibrium ionization conditions are evident for all the elements just after the flow goes through the temperature jump.
Figure 25.75: Numerical solutions of the stationary code. The figure shows the population fractions vs. space for the 12 elements most abundant in astrophysical plasmas assuming a stationary flow through a temperature jump.
Figure 25.75: ... continued ...
Figure 25.76: As in Figure 25.75 for the solutions of the FLASH code.
Figure 25.76: ... continued ...
The same problem was solved with the NEI module of the FLASH code, assuming that the plasma is initially in ionization equilibrium at $t = t_0$ over all the spatial domain. After a transient lasting approximately 700 s, in which the population fractions evolve due to the plasma flow through the temperature jump, the system reaches the stationary configuration. Outflow boundary conditions (zero-gradient) are assumed at both the left and right boundaries. Figure 25.76 shows the population fraction vs. space after 700 s.

### 25.7.2 The Delta-Function Heat Conduction Problem

The `ConductionDelta` problem tests the basic function of the `Diffuse` unit, in particular `Diffuse_therm`, in connection with `Conductivity`. It can be easily modified to examine the effects of Viscosity, of non-constant conductivity, and of combining these diffusive effects with hydrodynamics.

In its default configuration, `ConductionDelta` models the time development of a temperature peak that at some time $t_{ini}$ has the shape of a delta function in 1D or 3D, subject to heat conduction (with constant coefficient) only. An ideal gas EOS is assumed. The setup includes the `Hydro` code unit, but changes of the solution due to hydrodynamical effects are completely suppressed by zeroing all fluxes each time before `Diffuse_therm` is called.

The theoretical solution of this initial value problem is known: For any $t > t_{ini}$, the temperature profile is a Gaussian that spreads self-similarly over time. In particular in 1D, if the initial condition is defined as

$$T(x, t_{ini}) = Q\delta(x),$$

then

$$T(x, t) = \frac{Q}{(4\pi\chi(t-t_{ini}))^{1/2}}e^{-x^2/4\chi(t-t_{ini})},$$

(with $\chi$ the constant coefficient of diffusivity), see for example Zel’dovich and Raizer Ch. X.

See the end of Section 17.1 for alternative ways of configuring the test problem, using either a flux-based or a standalone `Diffuse` interface.

### 25.7.3 The HydroStatic Test Problem

The `HydroStatic` problem tests the basic function of hydrostatic boundary conditions implemented in the `Grid` unit, in connection with a `Hydro` implementation. It is essentially a 1D problem, but can be configured as 1 1D, 2D, or 3D setup. It can be easily modified to include additional physical effects, by including additional code units in the setup.

In its default configuration, `HydroStatic` is set up with constant `Gravity`. The domain is initialized with density, pressure, etc., fields representing an analytical solution of the hydrostatic problem with the given gravitational acceleration, essentially following the barometric formula.

This initial condition is then evolved in time. Ideally, the solution would remain completely static, and nothing should move. The deviation from this ideal behavior that occurs in practice is a measure of the quality of the discretization of the initial condition, of the hydrodynamics implementation, and of the boundary conditions. The effect of the latter, in particular, can be examined by visualizing artifacts that develop near the boundaries (in particular, velocity artifacts), and studying their dependence on the choice of boundary condition variant.

### 25.7.4 Hybrid-PIC Test Problems

A classic plasma model problem is that of an ion beam into a plasma (Winske 1986). Work by Matthew (1994) describes a two-dimensional simulation of a low density ion beam through a background plasma. The initial condition has a uniform magnetic field, with a beam of ions, number density $n_b$, propagating along the field with velocity $v_b = 10v_A$, where $v_A = B/\sqrt{\mu_0 m_i}$ is the Alfvén velocity, through a background (core) plasma of number density $n_c$. Both the background and beam ions have thermal velocities $v_{th} = v_A$. Here we study what is denoted a resonant beam with $n_0 = 0.015 n_c$ and $v_c = -0.2 v_A$. Electron temperature is assumed to be zero, so the electron pressure term in the electric field equation of state is zero. The weight of the macroparticles are chosen such that there is an equal number of core and beam macroparticles, each beam macroparticle thus represent fewer real ions than the core macroparticles. The number of magnetic field update subcycles is nine.
Figure 25.77: Temperature profile of the Delta-Function Heat Conduction Problem at two times. $t_{ini} = -1 \text{ ms}$, top: $t = 0 \text{ ms}$, bottom: $t = 2.46 \text{ ms}$.
25.7.4.1 One-dimensional ion beam

The spatial extent of the domain is 22016 km along the $x$-axis, divided up in 256 cells with periodic boundary conditions. The core number density is $n_c = 7 \text{ cm}^{-3}$. The magnetic field magnitude is $B = 6 \text{ nT}$ directed along the $x$-axis, which give an Alfvén velocity of 50 km/s and an ion inertial length of $\delta_i = 86 \text{ km}$, where $\delta_i = v_A/\Omega_i$ and $\Omega_i = q_i B/m_i$ is the ion gyrofrequency. The time step is $0.0865 \text{ s} = 0.05 \Omega_i^{-1}$, and the cell size is $\delta_i$. The number of particles per cell is 32. In Fig. 25.78 we show a velocity space plot of the macro-ions at time $t = 34.6 \text{ s} \approx 20 \Omega_i^{-1}$. This can be compared to Fig. 5 in Winske (1984).

![Velocity space plot](image)

Figure 25.78: Velocity space plot for a one-dimensional ion beam. Velocity along the $y$-axis as a function of position at time $t = 34.6 \text{ s} \approx 20 \Omega_i^{-1}$. Each gray dot is a core macro-ion, and each black dot is a beam macro-ion.

25.7.4.2 Two-dimensional ion beam

In the two-dimensional case we have a square grid with sides of length 22016 km, and 128 cells in each direction with periodic boundary conditions. The time step is $0.0216 \text{ s} = 0.05 \Omega_i^{-1}$, and the cell widths are $2\delta_i$. The number of particles per cell is 16. Otherwise the setup is identical to the one-dimensional case. In Fig. 25.79 we show the magnitude of the magnetic field $y$-component at time $t = 77.85 \text{ s} \approx 46 \Omega_i^{-1}$. This can be compared to Fig. 5 in Winske (1986).

25.7.5 Full-physics Laser Driven Simulation

The LaserSlab simulation provides an example of how to run a realistic, laser driven, simulation of an HEDP experiment. It exercises a wide range of HEDP physics including:

- 3T unsplit hydrodynamics in $R - z$ geometry
- Electron thermal conduction with Spitzer conductivities
- Ion/electron equilibration
- Radiation Diffusion with tabulated opacity
- Laser Ray Tracing
- Tabulated Equation of State

This simulation has no analytic solutions associated with it. Rather, it is designed to demonstrate the HEDP capabilities that are available in FLASH.

The simulation models a single laser beam illuminating a solid Aluminum disc in $R - z$ geometry. The laser is focused on the $z$-axis and enters the domain at a 45 degree angle. The simulation contains two
Figure 25.79: The magnetic field y-component for a two-dimensional ion beam at time \( t = 77.85 \text{ s} \approx 46 \Omega_j^{-1} \).

materials (or “species” in FLASH terminology). These are called CHAM (short for chamber) and TARG (short for target). The TARG species models the Aluminum disc. The laser travels through a low density Helium region before reaching the Aluminum. The properties of the CHAM species are set to model the low density Helium. A schematic showing the initial conditions is shown in Figure 25.80. The laser rays enter just above the lower-right corner of the domain and travel until they are absorbed or reflected. The beam intensity is not uniform across the beam cross section. Rather, it has a super-Gaussian intensity profile with a 10 micron spot radius.

The following setup line is used for this simulation:

```
./setup -auto LaserSlab -2d -nxb=16 -nyb=16 +hdf5typeio \
  species=cham,targ +mtmmmt +laser +uhd3t +mgd mgd_meshgroups=6 \ 
  -parfile=example.par
```

where:

- `-2d` and `+cylindrical` tells FLASH that this is a 2D simulation
- `-nxb=16` and `-nyb=16` sets the AMR block size to 16 by 16
- `+hdf5typeio` enables a newer version of IO that writes to HDF5 files. HDF5TypeIO is faster than other IO implementations in FLASH and is also more user friendly when generating figures using external tools, such as VisIt.
- `species=cham,targ` Creates two species named CHAM and TARG using the setup argument `instead of` the SPECIES keyword in the Simulation Config file. This is required for using the Multispecies opacity unit and is more user friendly since it allows many options to be specified in the runtime parameters file. See Section 11.4 for more information.
- `+mtmmmt` Activates the Multi-Temperature, Multi-Material, Multi-Type EOS in FLASH. This EOS lets users select a different EOS model (such as gamma-law or tabulated) for each species in the simulation through the runtime parameters file. (see )
- `+laser` Activates laser ray tracing (see Section 16.4).
• +mgd and mgd\_meshgroups=6 activates multigroup radiation diffusion and tells FLASH that there will be maximum of six groups per mesh (see Chapter 22)

• +uhd3t Activates the 3T unsplit hydro solver (see Section 14.1.4)

• -parfile=example.par Chooses a parameter file that is different than the default

The remainder of this section will describe how each physics unit functions and how the runtime parameters are chosen for this simulation. Images of the results will also be shown at the end of this section.

### 25.7.5.1 Initial Conditions and Material Properties

The section “Initial Conditions” section in the runtime parameter file, “example.par”, sets parameters which define the initial state of the fluid and also set the properties of the materials. The first runtime parameters in this section are:

- \( \text{sim\_targetRadius} = 200.0\text{e-04} \)
- \( \text{sim\_targetHeight} = 20.0\text{e-04} \)
- \( \text{sim\_vacuumHeight} = 60.0\text{e-04} \)

These options define the space occupied by the target material (the \textit{targ} species, Aluminum for this simulation). The Aluminum disc extends from:

\[
0 < R < \text{sim\_targetRadius} \quad \text{and} \quad \text{sim\_vacuumHeight} < z < \text{sim\_vacuumHeight} + \text{sim\_targetHeight},
\]

where all parameters are in centimeters.

The next section sets the initial fluid state in the target region and also sets the properties of the \textit{targ} species to match Aluminum. The relevant options are:

- \# Target material defaults set for Aluminum at room temperature:
- \( \text{sim\_rhoTarg} = 2.7 \) \quad \# Set target material density (solid Al)
CHAPTER 25. THE SUPPLIED TEST PROBLEMS

sim_teleTarg = 290.11375  # Set initial electron temperature
sim_tionTarg = 290.11375  # Set initial ion temperature
sim_tradTarg = 290.11375  # Set initial radiation temperature
ms_targA = 26.9815386    # Set average atomic mass
ms_targZ = 13.0          # Set average atomic number
ms_targZMin = 0.02       # Set minimum allowed ionization level
eos_targEosType = "eos_tab"  # Set EOS model to tabulated EOS
eos_targSubType = "ionmix4"  # Set EOS table file to IONMIX 4 format
eos_targTableFile = "al-imx-003.cn4"  # Set tabulated EOS file name

The initial density is chosen to match solid Aluminum. The initial ion, electron, and radiation temperatures are set to room temperature (290 K, 0.025 eV). The runtime parameters ms_targA and ms_targZ set the average atomic mass and atomic number of the targ species. Note that in the past, this information was specified by modifying the Simulation_initSpecies subroutine. This information can now be specified using runtime parameters since the species=cham,targ setup variable was used (see Section 11.4 for more information). The parameter ms_targZMin sets the minimum ionization level for targ. Without this parameter, the initial ionization would be nearly zero and the slab would be subcritical with respect to the laser light. The laser rays would penetrate deeply into the slab and could possible travel through it with no heating. This can be avoided by putting a lower limit on the ionization level. The final three parameters control the behavior of the MTMMMT EOS with respect to targ. The option eos_targEosType sets the EOS model to use for the Aluminum. In this case, we have chosen to use a tabulated EOS which is in the IONMIX4 format. Finally, we specify the name of the file which contains the EOS data. The file “al-imx-003.cn4” can be found in the LaserSlab simulation directory in the source tree. The IONMIX4 format, described in Section 21.4.3, is not human readable. However, the file “al-imx-003.imx.gz” contains a human readable representation of this EOS and opacity data.

A similar set of runtime parameter options exist for the cham species:

# Chamber material defaults set for Helium at pressure 1.6 mbar:
sim_rhoCham = 1.0e-05
sim_teleCham = 290.11375
sim_tionCham = 290.11375
sim_tradCham = 290.11375
ms_chamA = 4.002602
ms_chamZ = 2.0
eos_chamEosType = "eos_tab"
eos_chamSubType = "ionmix4"
eos_chamTableFile = "he-imx-005.cn4"

The initial Helium density, sim_rhoCham is chosen to be fairly low. This ensures that the Helium remains relatively transparent to the laser light (see Section 16.4.2 for a description of how the laser energy absorption coefficient is computed).

For both the targ and cham species, the initial state of the fluid is defined using a density with three temperatures. The initial EOS mode is set using the eosModeInit runtime parameter. It is set to “dens_temp_gather”. This tells FLASH that the initial state will be specified using the density and the temperature.

25.7.5.2 Multigroup Radiation Diffusion Parameters

This simulation uses multigroup radiation diffusion (MGD) to model the radiation field. This involves setting parameters which control the MGD package itself and setting parameters which control how opacities for the two species are computed. The MGD settings from the parameter file are:

rt_useMGD = .true.
rt_mgdNumGroups = 6
rt_mgdBounds_1 = 1.0e-01
rt_mgdBounds_2 = 1.0e+00
rt_mgdBounds_3 = 1.0e+01
rt_mgdBounds_4 = 1.0e+02
rt_mgdBounds_5 = 1.0e+03
rt_mgdBounds_6 = 1.0e+04
rt_mgdBounds_7 = 1.0e+05
rt_mgdFlMode = "fl_harmonic"
rt_mgdFlCoef = 1.0
rt_mgdXlBoundaryType = "reflecting"
rt_mgdXrBoundaryType = "vacuum"
rt_mgdYlBoundaryType = "vacuum"
rt_mgdYrBoundaryType = "reflecting"
rt_mgdZlBoundaryType = "reflecting"
rt_mgdZrBoundaryType = "reflecting"

The first option turns on MGD. The second parameter sets the number of radiation energy groups in the simulation, six in this case. Note that this number must be less than or equal to the value of the mgd_meshgroups setup variable. This restriction can be relaxed through the use of mesh replication (see Section 22.1.2). The runtime parameters beginning with rt_mgdBounds_ define the group boundaries for each energy group in electron-volts. The parameters rt_mgdFlMode and rt_mgdFlCoef set the flux limiter coefficient (see Section 17.1.3 and Section 22.1 for more information). The final six parameters set the boundary conditions for all groups.

The next set of parameters tell FLASH how to compute opacities for each material. In general, the total cell opacity is a number density weighted average of the opacity of each species in the cell. The parameters which control the behavior of the opacity unit for this simulation are:

useOpacity = .true.

### SET CHAMBER (HELIUM) OPACITY OPTIONS ###
op_chamAbsorb = "op_tabpa"
op_chamEmiss = "op_tabpe"
op_chamTrans = "op_tabro"
op_chamFileType = "ionmix4"
op_chamFileName = "he-imx-005.cn4"

### SET TARGET (ALUMINUM) OPACITY OPTIONS ###
op_targAbsorb = "op_tabpa"
op_targEmiss = "op_tabpe"
op_targTrans = "op_tabro"
op_targFileType = "ionmix4"
op_targFileName = "al-imx-003.cn4"

The first parameter tells FLASH that opacities will be computed. The next five parameters control how opacities are computed for the cham species (Helium) and the final five control the targ species (Aluminum). These options tell FLASH to use the IONMIX4 formatted opacity file “he-imx-005.cn4” for Helium and “al-imx-003.cn4” for Aluminum. Section 21.4.2.1 provides a detailed description of how these parameters are used. Note that the number of groups and group structure in these opacity files must be consistent with the rt_mgdNumGroups and rt_mgdBounds parameters that were described above.

25.7.5.3 Laser Parameters

This section describes the runtime parameters which control the behavior of the laser in FLASH. Modeling laser energy deposition using ray tracing is fairly complicated and requires large numbers of input parameters. It is highly recommended that you read through the section describing the behavior of the laser model in FLASH to learn more about how the ray tracing algorithms actually work (see Section 16.4).
This simulation uses a single laser beam which travels at an angle of 45 degrees from the $z$-axis. The “Laser Parameters” section of the runtime parameters file contains all of the options relevant to defining the laser beam. The first set of parameters are:

```
useEnergyDeposition = .true. # Turn on laser energy deposition
ed_maxRayCount   = 10000 # Set maximum number of rays/cycle/proc to 2000
ed_gradOrder     = 2     # Turn on laser ray refraction
```

The first parameter activates the laser model.

The `ed_maxRayCount` parameter sets the maximum number of rays which can be stored on a given process at once. It tells FLASH how much space to set aside for storing rays. In the case of a single process simulation, `ed_maxRayCount` must be less than the total number of rays launched on each cycle. The total number of rays launched is set by the `ed_numRays` parameters, described below. Finally, the parameter `ed_gradOrder` controls how the electron number density and temperature are interpolated within a cell. A value of two specifies linear interpolation.

The next set of parameters are:

```
ed_laser3Din2D = .true.
ed_laser3Din2DwedgeAngle = 0.1
```

Setting `ed_laser3Din2D` to `.true.` activates the 3D-in-2D ray trace algorithm. This means that in this simulation, the beams are defined in 3D geometry and the laser rays are traced in 3D. The total energy deposited is then projected back on the $R-z$ plane. This can significantly improve the accuracy of the ray trace when compared to 2D-in-2D ray-tracing. The exception to this is the case of a single beam centered on the $z$-axis and traveling along the $z$ axis. In this case, there is no difference between 3D-in-2D and 2D-in-2D ray tracing algorithms. In this example, the rays travel at a 45 degree angle. Thus, the 3D-in-2D ray-trace is needed. When using the 3D-in-2D ray trace, `ed_laser3Din2DwedgeAngle` sets the wedge angle. It should be set to a number between 0.1 and 1.0 degrees. See Section 16.4.7.

The next set of parameters are:

```
ed_useLaserIO   = .true.
ed_laserIOMaxNumPositions = 10000
ed_laserIONumRays   = 128
```

These parameters make use of the LaserIO package in FLASH for visualizing laser rays (see Section 16.4.9.4). These options tell FLASH to write out the trajectory of 128 rays to the plot files. The runtime parameter `plotFileIntervalStep` is set to 100 which tells FLASH to write plot files every 10 cycles. Later in this section, instructions and examples for plotting the laser ray data will be presented.

Next, a single laser pulse is defined. Each laser pulse (in this case, there is only one) represents a specific time history of the power and multiple pulses can be defined. The LaserSlab simulation has only a single laser beam which uses laser pulse number 1. The relevant runtime parameter options are:

```
# Define Pulse 1:
ed_numberOfSections_1 = 4
ed_time_1_1    = 0.0
ed_time_1_2    = 0.1e-09
ed_time_1_3    = 1.0e-09
ed_time_1_4    = 1.1e-09

ed_power_1_1 = 0.0
ed_power_1_2 = 1.0e+09
ed_power_1_3 = 1.0e+09
ed_power_1_4 = 0.0
```

The laser pulse is defined as a piecewise linear function. The parameter `ed_numSections_1` is set to four, meaning the laser power is defined using four different time/power points. The four time and power pairs are defined using the `ed_time_1_#` and `ed_power_1_#` parameters where the times are in seconds and the
powers are specified in Watts. In this case a square pulse is defined with a gradual rise, termination over 0.1 ns. The peak power is $10^9$ W and the pulse lasts for a total of 1.1 ns.

The next set of parameters define the laser beam itself. The trailing .1 in the parameter names indicate that these parameters are associated with the first beam (in this case, the only beam):

```
ed_lensX_1 = 1000.0e-04
ed_lensY_1 = 0.0e-04
ed_lensZ_1 = -1000.0e-04
ed_lensSemiAxisMajor_1 = 10.0e-04
ed_targetX_1 = 0.0e-04
ed_targetY_1 = 0.0e-04
ed_targetZ_1 = 60.0e-04
ed_targetSemiAxisMajor_1 = 10.0e-04
ed_targetSemiAxisMinor_1 = 10.0e-04
ed_pulseNumber_1 = 1
ed_wavelength_1 = 1.053
ed_crossSectionFunctionType_1 = "gaussian2D"
ed_gaussianExponent_1 = 4.0
ed_gaussianRadiusMajor_1 = 7.5e-04
ed_gaussianRadiusMinor_1 = 7.5e-04
ed_numberOfRays_1 = 4096
ed_gridType_1 = "radial2D"
ed_griddRadialTics_1 = 64
ed_semiAxisMajorTorsionAngle_1 = 0.0
ed_semiAxisMajorTorsionAxis_1 = "x"
```

The parameters `ed_targetX_1`, `ed_targetY_1`, and `ed_targetZ_1` define the coordinate of the center of the laser focal spot. Notice that even though this is a 2D simulation, three coordinates are specified. That is because this while most of the FLASH solvers will operate in 2D, $R-z$ geometry, the laser ray trace operates in 3D Cartesian geometry. For specifying the laser focal spot center, “X” refers to the $R$ direction in the FLASH simulation (which is also called the “X” direction for many FLASH runtime parameters, regardless of the geometry). The `ed_targetZ_1` parameter is referring to the $z$ direction in the $R-z$ simulation. This is actually the “Y” direction for other FLASH runtime parameters. Finally, the “Z” direction for the 3D-in-2D ray trace parameters is the direction pointing out of the computer screen. This same naming convention is used for the parameters `ed_lensX_1`, `ed_lensY_1`, and `ed_lensZ_1`. These parameters specify the center of the lens. On each time step, all rays are traced from a location on the lens to the focal spot. The lens must be defined so that it exists entirely outside of the domain. The parameters `ed_targetSemiAxisMajor_1` and `ed_targetSemiAxisMinor_1` set the radius of the laser spot. The laser beam can have an elliptical shape. In this case the two parameters will have different values, but usually, a circular beam is desired. In this case, the beam will have a circular shape and a 10 micron radius. The parameter `ed_lensSemiAxisMajor_1` sets the radius of the focal spot. The laser beam is associated with pulse number 1 (the only pulse in this case) using the parameter `ed_pulseNumber_1`. The parameter `ed_wavelength_1` specifies the wavelength, in microns, of the laser light. The `ed_numberOfRays_1` parameter tells FLASH to launch 4096 rays for the first beam every time the `EnergyDeposition` subroutine is called. When using unsplit hydrodynamics (as in this case), this routine is called a single time per cycle. When using split hydrodynamics, it is called twice.

Taken together, the parameters:

- `ed_crossSectionFunctionType_1`,
- `ed_gaussianExponent_1`,
- `ed_gaussianRadiusMajor_1`, and
- `ed_gaussianRadiusMinor_1`
define the spatial variation of the intensity across the beam. When \( \text{ed\_crossSectionFunctionType\_1} \) is set to “gaussian2D”, the intensity profile is a supergaussian described by:

\[
I(r) = I_0 \exp \left\{ - \left[ \left( \frac{x}{R_x} \right)^2 + \left( \frac{y}{R_y} \right)^2 \right]^{\gamma/2} \right\}
\]

where:

- \( R_x = \text{ed\_gaussianRadiusMajor\_1} \),
- \( R_y = \text{ed\_gaussianRadiusMinor\_1} \), and
- \( \gamma = \text{ed\_gaussianExponent\_1} \).

In our case, we’ve set up a beam with a super-Gaussian profile (\( \gamma = 4 \)) and an e-folding length of 7.5 microns. Since \( R_x = R_y \) we have a circular beam and not an elliptical beam. The user does not directly specify \( I_0 \). Rather, the user specifies the total beam power (as described above). This then sets \( I_0 \) through the integral relation:

\[
P = 2\pi \int_0^R drrI(r),
\]

The parameters \( \text{ed\_semiAxisMajorTorsionAngle\_1} \) and \( \text{ed\_semiAxisMajorTorsionAxis\_1} \) only need to change for non-circular beams.

Finally, the user must specify how rays will be spatially distributed across the beam. The \( \text{ed\_numberOfRays\_1} \) parameter sets the total number of rays to launch on each time step for each beam. Now we have to decide how to distribute those rays across the cross-section of the laser beam. The parameter \( \text{ed\_gridType\_1} \) is set to “radial2D”. This means the rays will be laid out on the circular cross-section of the beam with some radial and angular spacing. The area of the beam is divided into regions of equal radius and angle. The parameter \( \text{ed\_gridnRadialTics\_1} \) is set to 64 meaning that there are 64 radial slices in the beam cross section. Therefore, there are: \( \frac{4096}{64} = 64 \) angular slices.

### 25.7.5.4 Examining LaserSlab Simulation Output

This section describes how the output can be visualized using VisIt and through other means. LaserIO code in FLASH can be used to visualize ray paths (see Section 16.4.9.4 for more information). The LaserIO unit can only be used with parallel HDF5 IO implementations. The \( +\text{hdf5typeio} \) setup option is consistent with this. The runtime parameters:

- \( \text{ed\_useLaserIO} = \text{.true.} \)
- \( \text{ed\_laserIOMaxNumPositions} = 10000 \)
- \( \text{ed\_laserIONumRays} = 128 \)

tell FLASH to use LaserIO and to write out the trajectory of 128 rays to the plot files so they can be visualized in VisIt. Notice that this is a small subset of the total number of rays that are actually launched. Writing out all of the rays is inadvisable since it will have numerous negative effects on performance when large number of rays are used in the simulation. In this case, plot files are written every 0.1 ns and the simulation is run for 2 ns cycles (as indicated by the parameter \( \text{tmax} \)). Thus, the following plot files are generated:

- \( \text{lasslab\_hdf5\_plt\_cnt\_0000} \)
- \( \text{lasslab\_hdf5\_plt\_cnt\_0001} \)
- \( \ldots \)
- \( \text{lasslab\_hdf5\_plt\_cnt\_0199} \)
- \( \text{lasslab\_hdf5\_plt\_cnt\_0200} \)

These HDF5 files contain the ray trajectories for each of the plot time steps (cycles 1, 11, \ldots, 191, and 201 in this case). Unfortunately, at this time, the FLASH VisIt plugin does not natively support ray visualization.
The ray data must be extracted from the plot files in a form that VisIt does understand. The `extract_rays.py` script, found in the `tools/scripts` directory, performs this operation. Note, however, that `extract_rays.py` requires the NumPy and PyTables python packages to operate.

After this example LaserSlab simulation is run, the `extract_rays.py` script can be used as shown:

```bash
>>> extract_rays.py lasslab_hdf5_plt_cnt_*
Processing file: lasslab_hdf5_plt_cnt_0000
Processing file: lasslab_hdf5_plt_cnt_0001
...
Processing file: lasslab_hdf5_plt_cnt_0198
Processing file: lasslab_hdf5_plt_cnt_0199
Processing file: lasslab_hdf5_plt_cnt_0200
No ray data in "lasslab_hdf5_plt_cnt_0200" skipping...
```

Note that a warning message was printed for the last plot file. This occurs because there is no ray data for a plot file written on the last cycle of a simulation. This is a limitation of the LaserIO package. The script generates files in the VTK format named:

```
lasslab_las_0000.vtk
lasslab_las_0001.vtk
...
lasslab_las_0198.vtk
lasslab_las_0199.vtk
```

These files contain meshes which define the ray trajectories in a format understood by VisIt.

The ray data can be plotted on top of the computational mesh. In VisIt, load the `lasslab_las_*.vtk` database. Add the “RayPower Watts” Pseudocolor plot. The “RayPower Watts” plot colors the trajectories based on the power of each ray in units of Watts. Thus, the power of any one ray at any given time should be less than the beam powers specified in the runtime parameters file.

An example showing the ray powers and trajectories is shown in Figure 25.80. The trajectory of 128 laser rays has been drawn. The rays are colored based on their powers.

### Computing the Number Density and Average Ionization

For a simulation using the MTMMMT EOS, the electron number density itself is never directly computed. Rather, for unfortunate legacy reasons, FLASH only keeps track of the variables $Y_E$ and $SUMY$ which are defined as:

$$
Y_E = \frac{\bar{Z}}{\bar{A}}, \quad SUMY = \frac{1}{\bar{A}}
$$

where $\bar{Z}$ is the average ionization level and $\bar{A}$ is the average atomic mass of the cell. Thus, one can compute the average ionization and electron number density from these quantities. For example:

$$
\bar{A} = \frac{1}{SUMY}, \quad \bar{Z} = \frac{Y_E}{SUMY},
$$

$$
n_{ion} = N_A \frac{\rho}{\bar{A}} = SUMY N_A \rho,
$$

$$
n_{ele} = N_A \bar{Z} \frac{\rho}{\bar{A}} = Y_E N_A \rho
$$

Where $N_A$ is Avogadros number, $n_{ion}$ is the ion number density, $n_{ele}$ is the electron number density, and $\rho$ is the mass density (`DENS` in FLASH). These quantities can be plotted in VisIt using expressions.
Another useful tool in visualizing the behavior of the laser is the FLASH variable \texttt{DEPO}. This variable stores the laser energy deposited in a cell on a particular time step per unit mass. The division by mass removes the geometric factor present in the ray power. The amount of laser energy entering the domain and the amount of laser energy exiting the domain can be used to compute the fraction of laser energy that is deposited. The energy entering and exiting the domain is shown for each cycle and integrated over the entire simulation in the file “lasslab/LaserEnergyProfile.dat”. This file is produced whenever the laser ray tracing package is active in FLASH.

25.8 3T Shock Simulations

FLASH now has the ability to simulate plasmas which have separate ion, electron, and radiation temperatures (see Chapter 13. Usually, simulations which multiple temperatures have several physics models active including:

- Electron thermal conduction
- Ion/Electron equilibration
- Radiation emission, absorption, and diffusion

This section contains a series of simulations which verify FLASH through comparisons with analytic solutions of steady shocks where various assumptions are active. Unfortunately, no single analytic solution contains three distinct temperatures with realistic physical coefficients. Thus, each simulation is performed with a different set of assumptions active. Taken together, they adequately exercise the 3T capabilities in FLASH.

25.8.1 Shafranov Shock

The Shafranov problem (Shafranov, 1957) is a one-dimensional problem that provides a good verification for structure of 1D shock waves in a two-temperature plasma with separate ion and electron temperatures. The Shafranov shock solutions takes as input a given upstream condition and shock speed. It then computes the downstream conditions and a shock profile. The solution is fairly sophisticated in that it takes into account electron thermal conduction and ion/electron equilibration. An assumption is made that the electron entropy is continuous across the shock. Thus, immediately downstream of the shock, the ion-temperature is substantially higher than the electron temperature. Far downstream, the temperature equilibrate. Electron conduction creates a preheat region upstream of the shock. An gamma-law EOS is used (typically with $\gamma = 5/3$).

Unfortunately, the Shafranov shock solution can only be simplified to an ODE which must be numerically integrated. The \texttt{ShafranovShock} simulation directory includes analytic solutions for several materials including Hydrogen, Helium, and Xenon in the files \texttt{plasma_shock.out}, \texttt{plasma_shock_Z2.out}, and \texttt{plasma_shock_Z54.out} respectively.

Several solutions are compared here for the fully-ionized Helium case with the following initial/boundary conditions:

- Upstream Ion/Electron Temperature: 5 eV
- Upstream Density: 0.0018 g/cc
- Upstream Velocity: 0.0 cm/s
- Shock Speed: 1.623415272E+07 cm/s

Figure 25.81, Figure 25.82, and Figure 25.83, shows the electron/ion temperature, density, and velocity at 0.15 ns for three cases:

- Analytic Solution: This is the analytic solution for the steady shock. This solution is used to initialize the FLASH simulations. The simulations are correct to the extent that they are able to maintain this shock profile.
Figure 25.81: Electron and ion temperatures from Shafranov shock simulation

- Entropy Advection Solution: This is a 1D FLASH simulation using the split hydrodynamics solver in “entropy advection” mode as described in Section 14.1.4.1.

- RAGE-like Solution: This is a 1D FLASH simulation using the split hydrodynamics solver in “RAGE-like” mode as described in Section 14.1.4.2.

The figures show that the both the entropy advection and RAGE-like FLASH simulation are able to maintain the correct shock speeds. However, the entropy advection approach closely matches the correct analytic ion temperature profile while the RAGE-like simulation a peak ion temperature that is too low. This is the expected behavior since the RAGE-like mode does not attempt to ensure that electrons are adiabatically compressed by the shock.

This simulation can be setup with the following setup command:

```
# For the Entropy Advection Case:
./setup ShafranovShock -auto -1d +pm4dev -parfile=flash_Z2.par

# For the RAGE-like Case:
./setup ShafranovShock -auto -1d +pm4dev -parfile=ragelike_Z2.par
```

25.8.2 Non-Equilibrium Radiative Shock

The non-equilibrium radiative shock solution is an analytical solution to a steady, 1D, radiative shock where $T_e = T_i$ but $T_e \neq T_r$. It is presented in (Lowrie, 2008). A constant opacity is assumed, making this a gray simulation. The “analytic” solution is fairly complex and reduces to an ODE which must be evaluated numerically. This ODE is evaluated for a given set of upstream conditions and a given Mach number (evaluated relative to the upstream sound speed). A gamma-law equation of state is assumed.

The FLASH simulation of this simulation, which resides in the GrayDiffRadShock simulation directory. The simulation can be setup using the following command:

```
setup -auto GrayDiffRadShock -1d +pm4dev mgd_meshgroups=1
```

The simulation is performed using 3T hydrodynamics with the multigroup radiation diffusion (MGD) unit (see Chapter 22). A single radiation energy group is used with opacities set to $\sigma_a = \sigma_e = 423$ cm$^{-1}$ and
Figure 25.82: Mass density from Shafranov shock simulation

Figure 25.83: Velocity from Shafranov shock simulation
25.8.3T SHOCK SIMULATIONS

Figure 25.84: Temperatures from non-equilibrium radiative shock simulation. The FLASH results are compared to analytic solution.

\[ \sigma_t = 788 \text{ cm}^{-1} \text{ (see Section 21.4.1).} \quad \text{A } \gamma = \frac{5}{3} \text{ gamma-law EOS is used with } Z = 1 \text{ and } A = 2. \] The electron and ion temperatures are forced to equilibrium by using the Spitzer Heatexchange implementation (see: Section 16.5.1) where the ion/electron equilibration time has been reduced by a factor of \( 10^6 \) by setting the \texttt{hx.ieTimeCoeff} runtime parameter.

The initial conditions are defined by a step function where the jump occurs at \( x = 0 \). The upstream \((x < 0)\) and downstream \((x > 0)\) conditions are chosen so that the shock remains stationary. The correct jump conditions are to maintain a stationary mach two shock are:

- \( \rho_0 = 1.0 \text{ g/cc} \)
- \( T_0 = 100 \text{ eV} \)
- \( u_0 = 2.536e + 07 \text{ cm/s} \)
- \( \rho_1 = 2.286 \text{ g/cc} \)
- \( T_1 = 2.078 \text{ eV} \)
- \( u_1 = 1.11e + 07 \text{ cm/s} \)

where the subscript 1 represents downstream conditions and the subscript 0 represents upstream conditions. The runtime parameter \texttt{sim.P0} representing the ratio of radiation to matter pressure is set to \( 10^{-4} \).

The verification test is successful to the extent that FLASH is able to maintain a stationary shock with the correct steady state spatial profile as shown in Figure 8 of (Lowrie, 2008). This is an excellent verification test in that no special modifications to the FLASH code are needed to perform this test. The simulation is run for 4.25 ns which is enough time for the initial step function profile to reach a steady state solution. Figure 25.84 compares the temperatures in the FLASH simulation to the analytic solution. Figure 25.85 compares density to the analytic solution. Excellent agreement is obtained.

25.8.3 Blast Wave with Thermal Conduction

Warning: The reason behind the choices of several of the runtime parameters are not completely understood by the FLASH team at this time.
The ReinickeMeyer blast wave solution (Reinicke, 1991) models a blast wave in a single temperature fluid with thermal conduction. The semi-analytic solution reduces to an ODE which must be integrated numerically. Figure 25.86 compares a FLASH simulation to the analytic solution (obtained from cococubed.asu.edu/code_pages/vv.shtml after 0.3242 ns for a particular set of initial conditions. Excellent agreement is obtained with the analytic solution. The image shows that at this time the blast wave is at approximately 0.45 cm and the conduction front is at 0.9 cm. The simulation can be set up using the following command:

setup -auto ReinickeMeyer -1d +pm4dev +spherical -parfile=flash_SPH1D.par
Figure 25.86: Mass density and temperature from ReinickeMeyer blast wave FLASH simulation compared to analytic solution.
Part VIII

Tools
Two tools are included in the release of FLASH to assist users with analysis of data. The first, sfocu, provides a way to compare output files. The second, fidlr3.0, provides visualization and analysis tools by using the proprietary IDL package.
VisIt

The developers of FLASH also highly recommend VisIt, a free parallel interactive visualization package provided by Lawrence Livermore National Laboratory (see https://wci.llnl.gov/codes/visit/). VisIt runs on Unix and PC platforms, and can handle small desktop-size datasets as well as very large parallel datasets in the terascale range. VisIt provides a native reader to import FLASH2.5 and FLASH3. Version 1.10 and higher natively support FLASH3. For VisIt versions 1.8 or less, FLASH3 support can be obtained by installing a tarball patch available at http://flash.uchicago.edu/site/flashcode/user_support/visit/. Full instructions are also available at that site.
Chapter 27

Serial FLASH Output Comparison Utility (sfocu)

Sfocu (Serial Flash Output Comparison Utility) is mainly used as part of an automated testing suite called flashTest and was introduced in FLASH version 2.0 as a replacement for focu.

Sfocu is a serial utility which examines two FLASH checkpoint files and decides whether or not they are “equal” to ensure that any changes made to FLASH do not adversely affect subsequent simulation output. By “equal”, we mean that

- The leaf-block structure matches – each leaf block must have the same position and size in both datasets.
- The data arrays in the leaf blocks (dens, pres...) are identical.
- The number of particles are the same, and all floating point particle attributes are identical.

Thus, sfocu ignores information such as the particular numbering of the blocks and particles, the timestamp, the build information, and so on.

Sfocu can read HDF5 and PnetCDF FLASH checkpoint files. Although sfocu is a serial program, it is able to do comparisons on the output of large parallel simulations. Sfocu has been used on irix, linux, AIX and OSF1.

27.1 Building sfocu

The process is entirely manual, although Makefiles for certain machines have been provided. There are a few compile-time options which you set via the following preprocessor definitions in the Makefile (in the CDEFINES macro):

NO_HDF5 build without HDF5 support
NO_NCDF build without PnetCDF support
NEED_MPI certain parallel versions of HDF5 and all versions of PnetCDF need to be linked with the MPI library. This adds the necessary MPI_Init and MPI_Finalize calls to sfocu. There is no advantage to running sfocu on more than one processor; it will only give you multiple copies of the same report.

27.2 Using sfocu

The basic and most common usage is to run the command sfocu <file1> <file2>. The option -t <dist> allows a distance tolerance in comparing bounding boxes of blocks in two different files to determine which are the same (which have data to compare to one another). You might need to widen your terminal to view the output, since it can be over 80 columns. Sample output follows:
A: 2006-04-25/sod_2d_45deg_4levncmpi_chk_0001
B: 2005-12-14/sod_2d_45deg_4levncmpi_chk_0001

Min Error: $\inf (2|a-b| / \max(|a+b|, 1e-99) )$
Max Error: $\sup (2|a-b| / \max(|a+b|, 1e-99) )$
Abs Error: $\sup |a-b|$
Mag Error: $\sup |a-b| / \max(\sup |a|, \sup |b|, 1e-99)$

Block shapes for both files are: [8,8,1]
Mag-error tolerance: 1e-12
Total leaf blocks compared: 541 (all other blocks are ignored)

<table>
<thead>
<tr>
<th>Var</th>
<th>Bad Blocks</th>
<th>Min Error</th>
<th>Max Error</th>
<th>Abs Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>dens</td>
<td>502</td>
<td>0</td>
<td>1.098e-11</td>
<td>0.424</td>
</tr>
<tr>
<td>eint</td>
<td>502</td>
<td>0</td>
<td>1.1e-11</td>
<td>1.78</td>
</tr>
<tr>
<td>ener</td>
<td>502</td>
<td>0</td>
<td>8.847e-12</td>
<td>2.31</td>
</tr>
<tr>
<td>gamc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>game</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pres</td>
<td>502</td>
<td>0</td>
<td>1.835e-14</td>
<td>0.302</td>
</tr>
<tr>
<td>temp</td>
<td>502</td>
<td>0</td>
<td>1.4e-11</td>
<td>8.56e-09</td>
</tr>
<tr>
<td>velx</td>
<td>516</td>
<td>0</td>
<td>5.62e-17</td>
<td>-1.23e-16</td>
</tr>
<tr>
<td>vely</td>
<td>516</td>
<td>0</td>
<td>2</td>
<td>1e-89</td>
</tr>
<tr>
<td>velz</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mfrc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Var</th>
<th>Bad Blocks</th>
<th>Mag Error</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>dens</td>
<td>502</td>
<td>4.661e-12</td>
<td>1.36e+04</td>
<td>1.36e+04</td>
</tr>
<tr>
<td>eint</td>
<td>502</td>
<td>1.3e+04</td>
<td>1.36e+04</td>
<td>1.36e+04</td>
</tr>
<tr>
<td>ener</td>
<td>502</td>
<td>1.4e+04</td>
<td>1.36e+04</td>
<td>1.36e+04</td>
</tr>
<tr>
<td>gamc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>game</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pres</td>
<td>502</td>
<td>1.221e+14</td>
<td>1.13e+04</td>
<td>1.13e+04</td>
</tr>
<tr>
<td>temp</td>
<td>502</td>
<td>0.00351</td>
<td>0.00351</td>
<td>0.00351</td>
</tr>
<tr>
<td>velx</td>
<td>516</td>
<td>3.4e+04</td>
<td>1.79e+04</td>
<td>1.79e+04</td>
</tr>
<tr>
<td>vely</td>
<td>516</td>
<td>1.06e+14</td>
<td>0.83e+04</td>
<td>0.83e+04</td>
</tr>
<tr>
<td>velz</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mfrc</td>
<td>0</td>
<td>0</td>
<td>3.46e+04</td>
<td>3.46e+04</td>
</tr>
</tbody>
</table>

"Bad Blocks" is the number of leaf blocks where the data was found to differ between datasets. Four different error measures (min/max/abs/mag) are defined in the output above. In addition, the last six columns report the sum, maximum and minimum of the variables in the two files. Note that the sum is physically meaningless, since it is not volume-weighted. Finally, the last line permits other programs to parse the sfocu output easily: when the files are identical, the line will instead read SUCCESS.

It is possible for sfocu to miss machine-precision variations in the data on certain machines because of compiler or library issues, although this has only been observed on one platform, where the compiler produced code that ignored IEEE rules until the right flag was found.
Chapter 28

Drift

28.1 Introduction

Drift is a debugging tool added to FLASH to help catch programming mistakes that occur while refactoring code in a way that should not change numerical behavior. Historically, simulation checkpoints have been used to verify that results obtained after a code modification have not affected the numerics. But if changes are observed, then the best a developer can do to narrow the bug hunting search space is to look at pairs of checkpoint files from the two different code bases sequentially. The first pair to compare unequal will tell you that somewhere between that checkpoint and its immediate predecessor something in the code changed the numerics. Therefore, the search space can only be narrowed to the limit allowed by the checkpointing interval, which in FLASH, without clever calls to IO sprinkled about, is at best once per time cycle.

Drift aims to refine that granularity considerably by allowing comparisons to be made upon every modification to a block’s contents. To achieve this, drift intercepts calls to `Grid::releaseBlkPtr`, and inserts into them a step to checksum each of the variables stored on the block. Any checksums that do not match with respect to the last checksums recorded for that block are logged to a text file along with the source file and line number. The developer can then compare two drift logs generated by the different runs using `diff` to find the first log entry that generates unequal checksums, thus telling the developer which call to `Grid::releaseBlkPtr` first witnessed divergent values.

The following are example excerpts from two drift logs. Notice the checksum value has changed for variable `dens` on block 18. This should clue the developer in that the cause of divergent behavior lies somewhere between `Eos_wrapped.F90:249` and `hy_ppm_sweep.F90:533`.

<table>
<thead>
<tr>
<th>inst=2036</th>
<th>inst=2036</th>
</tr>
</thead>
<tbody>
<tr>
<td>step=1</td>
<td>step=1</td>
</tr>
<tr>
<td>src=Eos_wrapped.F90:249</td>
<td>src=Eos_wrapped.F90:249</td>
</tr>
<tr>
<td>blk=57</td>
<td>blk=57</td>
</tr>
<tr>
<td>dens E8366F68490D1B44</td>
<td>dens E8366F68490D1B44</td>
</tr>
<tr>
<td>eint 89D6855F465E4CE4</td>
<td>eint 89D6855F465E4CE4</td>
</tr>
<tr>
<td>ener 588E54F2E60C9E8F</td>
<td>ener 588E54F2E60C9E8F</td>
</tr>
<tr>
<td>pres 5646263E02E24E1</td>
<td>pres 5646263E02E24E1</td>
</tr>
<tr>
<td>temp DB675D5F0F7D4B8B</td>
<td>temp DB675D5F0F7D4B8B</td>
</tr>
<tr>
<td>velx 42546C82E30F0B3</td>
<td>velx 42546C82E30F0B3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inst=2100</th>
<th>inst=2100</th>
</tr>
</thead>
<tbody>
<tr>
<td>step=1</td>
<td>step=1</td>
</tr>
<tr>
<td>src=hy_ppm_sweep.F90:533</td>
<td>src=hy_ppm_sweep.F90:533</td>
</tr>
<tr>
<td>blk=18</td>
<td>blk=18</td>
</tr>
<tr>
<td>dens 5E52D67C5E93FF</td>
<td>dens 5E52D67C5E93FF</td>
</tr>
<tr>
<td>eint 9CD7982E504C7C7E</td>
<td>eint 9CD7982E504C7C7E</td>
</tr>
<tr>
<td>ener 443E03520C3536B9</td>
<td>ener 443E03520C3536B9</td>
</tr>
<tr>
<td>velx 8193E8C6914072E</td>
<td>velx 8193E8C6914072E</td>
</tr>
<tr>
<td>vely 8C5305C870E570E</td>
<td>vely 8C5305C870E570E</td>
</tr>
</tbody>
</table>
28.2 Enabling drift

In FLASH, drift is disabled by default. Enabling drift is done by hand editing the Flash.h file generated by the setup process. The directive line \#define DRIFT_ENABLE 0 should be changed to \#define DRIFT_ENABLE 1. Once this has been changed, a recompilation will be necessary by executing \texttt{make}.

With drift enabled, the FLASH executable will generate log files in the same directory it is executed in. These files will be named \texttt{drift.<rank>.log}, one for each MPI process.

The following runtime parameters are read by drift to control its behavior:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{drift_trunc_mantissa}</td>
<td>2</td>
<td>The number of least significant mantissa bits to zero out before hashing a floating point value. This can be used to stop numerical noise from altering checksum values.</td>
</tr>
<tr>
<td>\texttt{drift_verbose_inst}</td>
<td>0</td>
<td>The instance index at which drift should start logging checksums per call to \texttt{Grid_releaseBlkPtr}. Before this instance is hit, only user calls to \texttt{Driver_driftUnk} will generate log data. A value of zero means never log checksums per block. Instance counting is described below.</td>
</tr>
<tr>
<td>\texttt{drift_tuples}</td>
<td>.false.</td>
<td>A boolean switch indicating if drift should write logs in the more human readable &quot;non-tuples&quot; format or the machine friendly &quot;tuples&quot; format that can be read in by the \texttt{driftDee} script found in the tools/ directory. Generally the &quot;non-tuples&quot; format is a better choice to use with tools such as \texttt{diff}.</td>
</tr>
</tbody>
</table>

28.3 Typical workflow

Drift has two levels of output verbosity, let us refer to them as verbose and not verbose. When in non-verbose mode, drift will only generate output when directly told to do so through the \texttt{Driver_driftUnk} API call. This call tells drift to generate a checksum for each unk variable over all blocks in the domain and then log those checksums that have changed since the last call to \texttt{Driver_driftUnk}. Verbose mode also generates this information and additionally includes the per-block checksums for every call to \texttt{Grid_releaseBlkPtr}. Verbose mode can generate a lot of log data and so should only be activated when the simulation nears the point at which divergence originates. This is the reason for the \texttt{drift_verbose_inst} runtime parameter.

Drift internally maintains an "instance" counter that is incremented with every intercepted call to \texttt{Grid_releaseBlkPtr}. This is drift’s way of enumerating the program states. When comparing two drift logs, if the first checksum discrepancy occurs at instance number 1349 (arbitrary), then it is clear that somewhere between the 1348’th and 1349’th call to \texttt{Grid_releaseBlkPtr} a divergent event occurred.

The suggested workflow once drift is enabled is to first run both simulations with verbose mode off (\texttt{dirft_verbose_inst=0}). The main \texttt{Driver_evolveFlash} implementations have calls to \texttt{Driver_driftUnk} between all calls to FLASH unit advancement routines. So the default behavior of drift will generate multiple unk-wide checksums for each variable per timestep. These two drift logs should be compared to find the first entry with a mismatched checksum. Each entry generated by \texttt{Driver_driftUnk} will contain an instance range like in the following:

\begin{verbatim}
step=1
from=Driver_evolveFlash.F90:276
unks inst=1234 to 2345
dens 9CF3C169A5BB129C
eint 9573173C3B51CD12
ener 028A5D0DED1BC399
...
\end{verbatim}
The line "unks inst=1349 to 2345" informs us these checksums were generated sometime after the 2345'th call to \texttt{Grid\_releaseBlkPtr}. Assume this entry is the first such entry to not match checksums with its counterpart. Then we know that somewhere between instance 1234 and 2345 divergence began. So we set \texttt{drift\_verbose\_inst = 1234} in the runtime parameters file of each simulation and then run them both again. Now drift will run up to instance 1234 as before, only printing at calls to \texttt{Driver\_driftUnk}, but starting with instance 1234 each call to \texttt{Grid\_releaseBlkPtr} will induce a per block checksum to be logged as well. Now these two drift files can be compared to find the first difference, and hopefully get you on your way to hunting down the cause of the bug.

28.4 Caveats and Annoyances

The machinery drift uses to intercept calls to \texttt{Grid\_releaseBlkPtr} is lacking in sophistication, and as such can put some unwanted constraints on the code base. The technique used is to declare a preprocessor \texttt{#define} in \texttt{Flash\_h} to expand occurrences of \texttt{Grid\_releaseBlkPtr} to something larger that includes \texttt{\_FILE\_} and \texttt{\_LINE\_}. This is how drift is able to correlate calls to \texttt{Grid\_releaseBlkPtr} with the originating line of source code. Unfortunately this technique places a very specific restriction on the code once drift is enabled. The trouble comes from the types of source lines that may refer to a subroutine without calling it. The primary offender being \texttt{use} statements with \texttt{only} clauses listing the module members to import into scope. Because macro expansion is dumb with respect to context, it will expand occurrences of \texttt{Grid\_releaseBlkPtr} in these \texttt{use} statements, turning them into syntactic rubbish. The remedy for this issue is to make sure the line \texttt{#include "Flash\_h"} comes after all statements involving \texttt{Grid\_releaseBlkPtr} but not calling it, and before all statements that are calls to \texttt{Grid\_releaseBlkPtr}. In practice this is quite easy. With only one subroutine per file, there will only be one line like:

\begin{verbatim}
use Grid\_interface, only: ...., Grid\_\_releaseBlkPtr, ...
\end{verbatim}

and it will come before all calls to \texttt{Grid\_releaseBlkPtr}, so just move the \texttt{#include "Flash\_h"} after the \texttt{use} statements. The following is an example:

<table>
<thead>
<tr>
<th>Incorrect</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{#include &quot;Flash_h&quot;} subroutine Flash_subroutine() use Grid_interface, only: Grid_releaseBlkPtr implicit none [...] call Grid_releaseBlkPtr(...) [...] end subroutine Flash_subroutine</td>
<td>subroutine Flash_subroutine() use Grid_interface, only: Grid_releaseBlkPtr implicit none \texttt{#include &quot;Flash_h&quot;} [...] call Grid_releaseBlkPtr(...) [...] end subroutine Flash_subroutine</td>
</tr>
</tbody>
</table>

If such a solution is not possible because no separation between all \texttt{use} and \texttt{call} statements exists, then there are two remaining courses of action to get the source file to compile. One, hide these calls to \texttt{Grid\_releaseBlkPtr} from drift by forcefully disabling the macro expansion. To do so, just add the line \texttt{#undef Grid\_releaseBlkPtr} after \texttt{#include "Flash\_h"}. The second option is to carry out the macro expansion by hand. This also requires disabling the macro with the \texttt{undef} just mentioned, but then also rewriting each call to \texttt{Grid\_releaseBlkPtr} just as the preprocessor would. Please consult \texttt{Flash\_h} to see the text that gets substituted in for \texttt{Grid\_releaseBlkPtr}. 
Chapter 29

FLASH IDL Routines (fidlr3.0)

fidlr3.0 is a set of routines to read and plot data files produced by FLASH. The routines are written in the graphical display language IDL (Interactive Data Language) and require the IDL program from Research Systems Inc. (http://www.rsi.com). These routines include programs which can be run from the IDL command line to read 1D, 2D, or 3D FLASH datasets, interactively analyze datasets, and interpolate them onto uniform grids.

However, some of these routines are not described in this release of the documentation because they have not been thoroughly tested with FLASH4. A graphical user interface (GUI) to these routines (xflash3) is provided, which enables users to read FLASH AMR datasets and make plots of the data. Both plotfiles and checkpoint files, which differ only in the number and numerical precision of the variables stored, are supported.

fidlr3.0 supports Cartesian, cylindrical, polar and spherical geometries. The proper geometry should be detected by xflash3 using the geometry attribute in the file header. System requirements for running fidlr3.0 are: IDL version 5.6 and above, and the HDF5 library. The routines also have limited support for data files written with netCDF, although this output format has not been thoroughly tested. The routines are intended to be backwards-compatible with FLASH2, although again extensive testing has not been performed.

29.1 Installing and Running fidlr3.0

fidlr3.0 is distributed with FLASH and is contained in the tools/fidlr3.0/ directory. These routines were written and tested using IDL v6.1 for Linux. They should work without difficulty on any UNIX machine with IDL installed—any functionality of fidlr3.0 under Windows is purely coincidental. Due to copyright difficulties with GIF files, output image files are in PNG or Postscript format. Most graphics packages, like xv or the GIMP, should be able to convert between PNG format and other commonly used formats.

Installation of fidlr3.0 requires defining some environment variables, making sure your IDL path is properly set, and compiling the support for HDF5 files. These procedures are described below.

29.1.1 Setting Up fidlr3.0 Environment Variables

The FLASH fidlr3.0 routines are located in the tools/fidlr3.0/ subdirectory of the FLASH root directory. To use them you must set two environment variables. First set the value of XFLASH3_DIR to the location of the FLASH IDL routines; for example, under csh, use

```
setenv XFLASH3_DIR flash-root-path/tools/fidlr3.0
```

where flash-root-path is the absolute path of the FLASH3 root directory. This variable is used in the plotting routines to find the customized color table and setup parameters for xflash3.

Next, make sure that you have an IDL_DIR environment variable set. This should point to the directory in which the IDL distribution is installed. For example, if IDL is installed in idl-root-path, then you would define
setenv IDL\_DIR idl-root-path.

Finally, you need to tell IDL where to find the fidlr3.0 routines. This is accomplished through the IDL\_PATH environment variable

\[
\text{setenv IDL\_PATH } \{\text{XFLASH3\_DIR}\}:\{\text{IDL\_DIR}\}:/\text{lib}
\]

If you already have an IDL\_PATH environment variable defined, just add XFLASH3\_DIR to the beginning of it. You may wish to include these commands in your .cshrc (or the analogous versions in your .profile file, depending on your shell) to avoid having to reissue them every time you log in. It is important that the \{XFLASH3\_DIR\} come before the IDL directories in the path and that the \{IDL\_DIR\}/lib directory be included as well.

### 29.1.2 Running IDL

fidlr3.0 uses 8-bit color tables for all of its plotting. On displays with higher color depths, it may be necessary to use color overlays to get the proper colors on your display. For SGI machines, launching IDL with the start.pro script will enable 8-bit pseudocolor overlays. For Linux boxes, setting the X color depth to 24-bits per pixel and launching IDL with the start_linux.pro script usually produces proper colors.

\[
\text{prompt} \text{> idl start\_linux}
\]

### 29.2 xflash3: A Widget Interface to Plotting FLASH Datasets

The main interface to the fidlr3.0 routines for plotting FLASH datasets is xflash3. Typing xflash3 at the IDL command prompt will launch the main xflash3 widget, shown in Figure 29.1.

IDL> xflash3

xflash3 produces colormap plots of FLASH data with optional overlays of velocity vectors, contours, and the block structure. The basic operation of xflash3 is to specify a single output file (either checkpoint or plotfile) as a prototype for the FLASH simulation. The prototype is probed for the list of variables it contains, and then the remaining plot options become active.

xflash3 can output to the screen, Postscript, or a PNG image file. If the data is significantly higher resolution than the output device, xflash3 will sub-sample the image by one or more levels of refinement before plotting.

Once the image is plotted, the query (2-d data only) and 1-d slice buttons will become active. Pressing query and then clicking anywhere in the domain will pop up a window containing the values of all the FLASH variables in the cell nearest the cursor. The query function uses the actual FLASH data—not the interpolated/uniformly gridded data generated for the plots. Pressing 1-d slice and then left-clicking on the plot will produce a 1-d slice vertically through the point. Right-clicking on the domain produces a horizontal slice through the data.

The widget is broken into several sections, with some features initially disabled. Not all options are available in all dimensions, or in this release of FLASH4. These sections are explained below.

### 29.2.1 File Menu

The file to be visualized is composed of the path, the basename (the same base name used in the flash.par file) with any file type information appended to it (e.g. 'hdf5_chk') and the range of suffixes through which to loop. By default, xflash3 sets the path to the working directory from which IDL was started. xflash3 requires a prototype file to work on a dataset. The prototype can be any of the files in the dataset that has the same name structure (i.e. everything is the same but the suffix) and contains the same variables.
29.2.1 File/Open prototype...

The *Open prototype...* menu option will bring up the file selection dialog box (see Figure 29.2). Once a plotfile or checkpoint prototype is selected, the remaining options on the xflash widget become active, and the variable list box “Mesh Variables” is populated with the list of variables in the file (see Figure 29.3).

*xflash3* will automatically determine if the file is an HDF5 or netCDF file and read the ‘unknown names’ dataset to get the variable list. Some derived variables will also appear on the list (for example, sound speed), if the dependent variables are contained in the datafile. These derived variables are currently inoperable in 3-D.

29.2.2 Defaults Menu

The *Defaults* menu allows you to select one of the predefined problem defaults. This choice is provided for the convenience of users who want to plot the same problem repeated using the same data ranges. This menu item will load the options (data ranges, velocity parameters, and contour options) for the problem as specified in the *xflash_defaults* procedure. When *xflash3* is started, *xflash_defaults* is executed to read in the known problem names. The data ranges and velocity defaults are then updated. To add a problem to *xflash3*, only the *xflash_defaults* procedure needs to be modified. The details of this procedure are provided in the comment header in *xflash_defaults*. It is not necessary to add a problem in order to plot a dataset, since all default values can be overridden through the widget.
29.2.3 Colormap Menu

The colormap menu lists the colormaps available to xflash3. These colormaps are stored in the file flash_colors.tbl in the fidlr3.0 directory and differ from the standard IDL colormaps. The first 12 colors in the colormaps are reserved by xflash3 to hold the primary colors used for different aspects of the plotting. These colormaps are used for 2-d and 3-d data only. At present, there is no control over the line color in 1-d.

29.2.4 X/Y plot count Menu

The X/Y plot count menu specifies how many plots to put on a single page when looping over suffixes in a dataset. At present, this only works for 2-d data. Note, the query and 1-d slice operations will not work if there are multiple plots per page.

29.2.5 Plotting options available from the GUI

Various options are available on the xflash3 user interface to change the appearance of the plots.

29.2.5.1 File Options

The first section below the menu bar specifies the file options. This allows you to specify the range of files in the dataset (i.e. the suffixes) to loop over. The optional step parameter can be used to skip over files when looping through the dataset. For example, to generate a “movie” of the checkpoint files from initial conditions to checkpoint number 17, enter 0000 in the first suffix: box, and enter 0017 in the box following to. Leave the step at the default of 1 to visualize every output.

29.2.5.2 Output Options

A plot can be output to the screen (default), a Postscript file, or a PNG file. The output filenames are composed from the basename + variable name + suffix. For outputs to the screen or PNG, the plot size options allow you to specify the image size in pixels. For Postscript output, xflash3 chooses portrait or landscape orientation depending on the aspect ratio of the domain.
29.2. XFLASH3: A WIDGET INTERFACE TO PLOTTING FLASH DATASETS

29.2.5.3 Parallel Block Distribution Options

A plot of the parallel block distribution on the physical domain can be created. To use this feature, select the “Enable” toggle button located in the “Parallel Block Distribution” row of the xflash GUI. If more than one processor has been used to generate the simulation results, the different levels of refinement in the simulation can then be selected from a drop down menu. The menu shows which processors hold which blocks at a given level of refinement. Each processor is denoted by a unique color. Additionally, the processor number can be superimposed on the plot by selecting the "Show Plot Numbers" checkbox.

29.2.5.4 Mesh Variables

The variables dropbox lists the mesh, or grid variables stored in the ‘unknown names’ record in the data file and any derived variables that xflash3 knows how to construct from these variables (e.g. sound speed). This selection allows you to choose the variable to be plotted. By default, xflash3 reads all the variables in a file in 1- and 2-d datasets, so switching the variable to plot can be done without re-reading. At present, there is no easy way to add a derived variable. Both the widget routine (xflash3.pro) and the plotting backend (xplot#d_amr.pro) will need to be told about any new derived variables. Users wishing to add derived variables should look at how the total velocity (tot_vel) is computed.

29.2.5.5 Options

The options block allows you to toggle various options on/off. Table 29.1 lists the various options available.
Table 29.1: xflash3 options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>Plot the base-10 log of the variable.</td>
</tr>
<tr>
<td>max</td>
<td>When a sequence of files is defined in the file options, plot the maximum of the variable in each cell over all the files.</td>
</tr>
<tr>
<td>annotate</td>
<td>Toggle the title and time information on the plot.</td>
</tr>
<tr>
<td>show ticks</td>
<td>Show the axis tick marks on the plot.</td>
</tr>
<tr>
<td>abs. value</td>
<td>Plot the absolute value of the dataset. This operation is performed before taking the log.</td>
</tr>
<tr>
<td>show blocks</td>
<td>Draw the zone boundaries on the plot.</td>
</tr>
<tr>
<td>colorbar</td>
<td>Plot the colorbar legend for the data range.</td>
</tr>
</tbody>
</table>

29.2.5.6 Data Range

These fields allow you to specify the range of the variable to plot. Data outside of the range will be set to the minimum or maximum values of the colormap. If the auto box is checked, the limits will be ignored, and the data will be scaled to the minimum and maximum values of the variable in the dataset.

29.2.5.7 Slice Plane

The slice plane group is only active for 3-d datasets. This option allows you to select a plane for plotting in \((x-y, x-z, y-z)\).

29.2.5.8 Zoom

The zoom options allow you to set the domain limits for the plot. A value of -1 uses the actual limit of the domain. For 3-d plots, only one field will be available in the direction perpendicular to the slice plane. The zoom box button puts a box cursor on the plot and allows you to select a region to zoom in on by positioning and resizing the box with the mouse. Use the left mouse button to move the center of the zoom box. Use the middle button to resize the box. Finally, right-click with the mouse when you are satisfied with the zoom box selection. (Note that you must choose the "Plot" button again to see the results of the selected zoom. The reset button will reset the zoom limits.

29.2.6 Plotting buttons

Several buttons are located at the bottom of the xflash3 user interface which pop up additional windows. Most of these set additional groups of options, but the actual commands to create plots are located on the bottom row.

29.2.6.1 Contour Options Button

This button launches a dialog box that allows you to select up to 4 contour lines to plot on top of the colormap plot (see Figure 29.4). The variable, value, and color are specified for each reference contour. To plot a contour, select the check box next to the contour number. This will allow you to set the variable from which to make the contour, the value of the contour, and the color. This option is available in 2-d only at present.

29.2.6.2 Vector Options Button

This button launches a dialog box that allows you to set the options used to plot vectors on the plot (see Figure 29.5). This option is usually utilized to overplot velocity vectors. First select the plot vectors checkbox to enable the other options. Choose the variables to generate vectors with the \(x\)-component and \(y\)-component pull-down boxes. These choices are set to velx and vely by default. typical vector sets the vector length to
29.2. XFLASH3: A WIDGET INTERFACE TO PLOTTING FLASH DATASETS

Figure 29.4: The xflash3 contour option subwidget.

which to scale the vectors, and minimum vector and maximum vector specify the range of vectors for which
to plot. xskip and yskip allow you to thin out the arrows. This option is available in 2-d only.

29.2.6.3 Particle Options Button

This button enables the user to set options for plotting particles. Since particle-handling routines are not
present in this release of FLASH4, this option is disabled in xflash3.

29.2.6.4 Histogram Options Button

This button pops up a dialog box that allows you to set the histogram options. Currently, only the number
of bins and the scale of the $y$-axis can be set. This option is disabled in this release of FLASH3.

29.2.6.5 Floating Label Button

This button pops up a dialog box that allows you to add annotation to the plot. First select the use floating
label checkbox to enable the other options. Choose the size of the text in pixels, the thickness of the font,
and the color of the text. Also select the relative position on the screen. The multiple plots button allows
different annotation to be placed on each of plot displayed.

29.2.6.6 Plot Button

Create the colormap plot after selecting the desired options described above. The status of the plot will
appear on the status bar at the bottom.

29.2.6.7 Histogram Button

Create a histogram of the data. This option is disabled in this release of FLASH3.

29.2.6.8 Query Button

The query button becomes active once the plot is created. Click on Query and then click somewhere in the
domain to generate a popup window listing the data values at the cursor location (see Figure 29.6). Use the
Close button to dismiss the results.
29.2.6.9 1-d Slice Button

This button is available for 2-d and 3-d datasets only. Clicking on 1-d Slice and then left-clicking in the domain will plot a one-dimensional slice of the current variable vertically through the point selected. A right-click will produce a horizontal slice. This function inherits the options chosen in the Options block.

29.3 Comparing two datasets

From the IDL prompt, a plot showing visual difference can be created with the command `diff3`. Comparisons can be made between any two variables in different files or within the same file. However, this command currently is supported ONLY for two-dimensional datasets. For example, the command:

```
IDL> diff3, '<path_to_file>/flash_hdf5_0001','dens', '<path_to_file>/flash_hdf5_0002','dens'
```

plots the difference between the 'dens' variable in two different Flash files.
Figure 29.6: The xflash3 query widget, displaying information for a cell.
Chapter 30

convertspec3d

The Spect3D software, developed by Prism Scientific Computing:


is a widely used tool that can be used to model simulated diagnostic responses generated by radiation hydrodynamics codes. The simulated images mimic common diagnostics fielded in many HEDP experiments. The `convertspec3d` python script can be used to convert FLASH output into input for the Spect3D code.

30.1 Installation

The `convertspec3d` script is written in python and depends on several popular python packages. The dependencies include:

- Python 2.7
- PyTables
- The python NETCDF package: [http://code.google.com/p/netcdf4-python/](http://code.google.com/p/netcdf4-python/)
- The setuptools package

Once these dependencies are installed, you can install the `convertspec3d` script. To do so, enter the tools directory of the FLASH source tree and enter:

```
python setup.py install
```

By default, this will try to install the FLASH python tools to a system wide location. You will need root/administrator privileges for this. If you are on a system where you do not have root privileges, just enter:

```
python setup.py install --user
```

for a user space installation. Type `convertspec3d` in the command line an you should see some help information - if the installation went well. For example, you should see something like:

```
> convertspec3d
usage: convertspec3d [-h] --species SPECIES [--extra EXTRA]
               file_names [file_names ...]
convertspec3d: error: too few arguments
```
30.2 Usage

To describe the usage of the script, it will be assumed that the user is modeling a laser driven HEDP experiment. The example will be based on the LaserSlab simulation described in 25.7.5. The script essentially assumes you are running a simulation using the MTMMMT EOS, where the plasma electron number density and ion number density can be determined using the SUMY and YE variables. Spect3D requires this information for most applications. The simulation must have material information where species are used to model each material. In the case of the LaserSlab simulation, there are two species: cham and targ.

The script takes, as arguments, a list of one or more FLASH checkpoint/plot files and converts each of these files. The new files have the same names as the old files, but have a .exo extension. For example, the checkpoint file laslab_hdf5_chk_0000 would be converted into the file laslab_hdf5_chk_0000.exo. The minimal information required by the script is the list of the FLASH output files and a list of the species. For example, for the LaserSlab simulation described in section 25.7.5, the command for convert the checkpoint files would be:

%> convertspec3d laslab_hdf5_chk_* --species=cham,targ

You should then see a series of .exo files generated. The new files contain the following information by default:

- The electron temperature in eV
- The ion temperature in eV
- The electron number density in 1/cc
- The ion number density in 1/cc
- The mass density in g/cc
- The “partial” mass density for each species (i.e. the fraction of the total density in each cell accounted for by each species)

With this information, users can run many Spect3D simulations.

If FLASH plot files are being converted, they must contain certain variables. The required variables are:

- dens
- tele
- tion
- sumy
- ye
- The variables for each species. In this example, cham and targ

In addition to simulating diagnostic responses, Spect3D can also plot simulation code output directly. Thus, even though Spect3D doesn’t require it, users might want to plot the radiation temperature, or some other variable, in Spect3D. The --extra command line argument for convertspec3d allows users to add additional variables. For example, let’s extend the convertspec3d command to include the total specific internal energy and radiation temperature:

%> convertspec3d laslab_hdf5_chk_* --species=cham,targ --extra=trad,eint
Part IX

Going Further with FLASH
Chapter 31

Adding new solvers

Adding new solvers (either for new or existing physics) to FLASH starts with a new subdirectory at an appropriate location in the source tree. If the solver provides an alternative implementation for the already supported physics, then the subdirectory should be added below the main sub-unit directory for that sub-unit. For example, adding a `Constant` implementation to the `Heatexchange` unit requires adding a subdirectory `Constant` under “source/physics/sourceTerms/Heatexchange/HeatexchangeMain”. If the solver is adding new physics to the code, a new capitalized subdirectory should be placed under the `physics` subdirectory, effectively creating a new unit, and its API should be defined by placing the null implementations of the relevant functions in the subdirectory. The implementations themselves should go into the `UnitMain` subdirectory placed in the unit. For example adding a `Heat` unit the to code requires putting a directory `Heat` under “source/physics”, and adding files `Heat_init.F90`, `Heat_finalize.F90` and at least one “doer” function such as `Heat.F90` with null implementations of the corresponding functions. Other than the source files, a Makefile and a Config file may need to be added to the subdirectory. If the solver is a new unit, both those files will be needed at the Unit and UnitMain levels. If the solver is an alternative implementation, the files may or may not be needed. See http://flash.uchicago.edu/site/flashcode/user_support/flash_howtos/ExampleFlashUnit/ for the architecture of a unit.

Makefile: The Makefile snippet added to any directory should reflect all the files in the subdirectory that are not inherited from a higher level directory. At the API level, the Makefile should include all the functions that constitute the API for the unit. At any lower level, those functions should not be included in the Makefile, instead all the local files at that level should be included. The implemented source files appear at UnitMain or some other peer level (if there is more than one sub-unit in the unit). An example of Makefile for a unit “Heat” with only three functions is as follows:

```make
# Makefile for Heat unit API
Heat = Heat_init.o Heat_finalize.o Heat.o
```

All the subdirectories lying below this level have their Makefiles use macro concatenation to add to the list. For example, if the implementation of the Heat unit adds one file `he_getTemp.F90` in the HeatMain implementation, the corresponding Makefile will be

```make
# Makefile for HeatMain sub-unit
Heat += Heat\_data.o he_getTemp.o
```

Note that the sub-unit’s Makefile only makes reference to files provided at its own level of the directory hierarchy, and the units will have at least one data module (in this instance `Heat\_data`) which contains all the unit scope data. If the sub-unit provides special versions of routines to override those supplied by the any implementation at a higher level directory in the unit, they do not need to be mentioned again in the object list.

Config: Create a configuration file for the unit, sub-unit or alternative implementation you are creating. All configuration files in a sub-unit path are used by setup, so a sub-unit inherits its parent unit’s configuration. Config should declare any runtime parameters you wish to make available to the code when this
unit is included. It should indicate which (if any) other units your unit requires in order to function, and it should indicate which (if any) of its sub-units should be used as a default if none is specified when setup is run. The configuration file format is described in Section 3.1.

This is all that is necessary to add a unit or sub-unit to the code. If you are creating a new solver for an existing physics unit, the unit itself should provide the interface layer to the rest of the code. As long as your sub-unit provides the routines expected by the interface layer, the sub-unit should be ready to work. However, if you are adding a new unit, you will need to add calls to your API routines elsewhere in the code as needed. The most likely place where these calls will need to be placed will be in the Driver unit.

It is difficult to give complete general guidance; here we simply note a few things to keep in mind. If you wish to be able to turn your unit on or off without recompiling the code, create a new runtime parameter (e.g., useUnit) at the API level of the your unit. You can then test the value of this parameter before calling your API routines from the main code. For example, the Burn unit routines are only called if (useBurn==true). (Of course, if the Burn unit is not included in the code, setting useBurn to true will result in empty subroutine calls.)

You will need to add Unit_data.F90 file which will contain all the data you want to have visible everywhere within the unit. Note that this data is not supposed to be visible outside the unit. You may wish have Driver_initFlash call your unit’s initialization routine and Driver_finalizeFlash to call your unit’s finalize routine.

If your solver introduces a constraint on the timestep, you should create a routine named Unit_computeDt() that computes this constraint. Add a call to this routine in Drive_computeDt.F90 (part of the Driver unit). Your routine should operate on a single block and take three parameters: the timestep variable (a real variable which you should set to the smaller of itself and your constraint before returning), the minimum timestep location (an integer array with five elements), and the block identifier (an integer). Returning anything for the minimum location is optional, but the other timestep routines interpret it in the following way. The first three elements are set to the coordinates within the block of the cell contributing the minimum timestep. The fourth element is set to the block identifier, and the fifth is set to the current processor identifier. This information tags, along with the timestep constraint, when blocks and solvers are compared across processors, and it is printed on stdout by the master processor along with the timestep information as FLASH advances.

If your solver is time-dependent, you will need to add a call to your solver in the Drive_evolveFlash routine. Limit the entry points into your unit to the API functions. It is a FLASH architecture requirement, violating this will make the code unmaintainable and will make it hard to get support from the code developers.
Chapter 32
Porting FLASH to other machines

Porting FLASH to new architectures should be fairly straightforward for most Unix or Unix-like systems. For systems which look nothing like Unix or which have no ability to interpret the setup script or makefiles, extensive reworking of the meta-code which configures and builds FLASH would be necessary. We do not treat such systems here; rather than do so, it would be simpler for us to do the port ourselves.

For Unix-like systems, you should make sure that your system has csh, a gmake that permits included makefiles, awk, sed, and Python. Next, create a directory in sites/ with the name of your site. It is best to name your site according to the output from the hostname -f command (it will return something like code.uchicago.edu) so that the setup script can automatically determine your site without you needing to manually specify the -site setup script option. Copy a Makefile.h from a pre-existing site directory into your new site directory.

32.1 Writing a Makefile.h

To reflect your system, you must modify the different macros defined in Makefile:

FCOMP : the name of the Fortran 90 compiler
CCOMP : the name of the C compiler
CPPCOMP : the name of the C++ compiler
LINK : the name of the linker (usually the Fortran compiler should serve as the linker)
PP : a flag (if any) that should precede a preprocessor directive (typically -D)
FFLAGS_OPT : the Fortran compilation flags to produce an optimized executable. These are the flags used when -auto is given to setup.
FFLAGS_DEBUG : the Fortran compilation flags to produce an executable that can be used with a debugger (e.g. totalview). These flags are used when -debug is passed to setup.
FFLAGS_TEST : Fortran compilation flags to produce an executable suitable for testing. These usually involve less optimization. These flags are used when -test is passed to setup.
CFLAGS_OPT : the optimized set of compilation flags for C/C++ code.
CFLAGS_DEBUG : the debug compilation flags for C/C++ code.
CFLAGS_TEST : the test compilation flags for C/C++ code.
LFLAGS_OPT : linker flags used with the _OPT compilation flags. This usually ends in `-o' to rename the executable.
LFLAGS_DEBUG : linker flags used with the _DEBUG compilation.

LFLAGS_TEST : linker flags used with the _TEST compilation.

LIB_OPT : libraries to link in with the _OPT compilation. This should include the MPI library if an MPI wrapper for the linker was not used (e.g. mpif90).

LIB_DEBUG : libraries to link in with the _DEBUG compilation.

LIB_TEST : libraries to link in with the _TEST compilation.

LIB_HDF5 the necessary link line required to link in the HDF5 library. This will look something like

- L /path/to/library -1hdf5

For example, here's how you might modify the macros defined in the Makefile.h for code.uchicago.edu. The first part of Makefile.h defines the paths for the MPI wrapper script and various I/O and numerical libraries.

MPI_PATH = /opt/mpich2/intel/1.4.1p1
HDF5_PATH = /opt/hdf5/intel/1.8.7
HYPRE_PATH = /opt/hypre/intel/2.7.0b
NCMPI_PATH = /opt/netcdf/intel/1.2.0

These should be modified to reflect the locations on your system. Next we setup the compilers and linker. We almost always use the Fortran 90 compiler as the linker, so the Fortran libraries are automatically linked in.

FCOMP = ${MPI_PATH}/bin/mpif90
CCOMP = ${MPI_PATH}/bin/mpicc
CPPCOMP = ${MPI_PATH}/bin/mpicxx
LINK = ${MPI_PATH}/bin/mpif90

# pre-processor flag
PP = -D

These commands will need to be changed if your compiler names are different. You are encouraged to use the compiler wrapper scripts instead of the compilers directly. Note that some older versions of MPICH do not recognize .F90 as a valid extension. For these, you can either update MPICH to a later version or edit mpif90 and add .F90 to the line that checks for a valid extension. The PP macro refers to the pre-processor and should be set to the flag that the compiler uses to pass information to the C preprocessor (usually -D).

We define three different setups of compiler flags as described earlier: the "_OPT" set for normal, fully optimized code, the "_DEBUG" set for debugging FLASH, and the "_TEST" set for regression testing. This latter set usually has less optimization. These three sets are picked with the -auto, -debug, and -test flags to setup respectively.

FFLAGS_OPT = -c -g -r8 -i4 -O3 -real_size 64 -diag-disable 10120
FFLAGS_DEBUG = -c -g -r8 -i4 -O0 -check bounds -check format -traceback -fp-stack-check -diag-disable 10120 -fpe0 -check pointers
FFLAGS_TEST = \${FFLAGS_OPT} -fp-model precise
F90FLAGS =

CFLAGS_OPT = -c -O3 -g -D_LARGEFILE64_SOURCE -D_FORTIFY_SOURCE=2 -diag-disable 10120
CFLAGS_DEBUG = -c -O0 -g -traceback -debug all -debug extended -D_LARGEFILE64_SOURCE -diag-disable 10120 -ftrapuv -fp-stack-check
CFLAGS_TEST = \${CFLAGS_OPT} -fp-model precise
Next come the linker flags. Typically, these have only `-o` to rename the executable, and some debug flags
(e.g. `-g`) for the “DEBUG” set.

LFLAGS_OPT = -diag-disable 10120 -O3 -o
LFLAGS_DEBUG = -diag-disable 10120 -o
LFLAGS_TEST = \${LFLAGS_OPT}

There are library macros for any libraries that are required by a specific FLASH unit (e.g. HDF5).

CFLAGS_MPI = -I\$(MPI_PATH)/include
CFLAGS_HDF5 = -I\$(HDF5_PATH)/include -DH5_USE_16_API
CFLAGS_NCMPI = -I\$(NCMPI_PATH)/include
FFLAGS_HYPRE = -I\$(HYPRE_PATH)/include

LIB_MPI =
LIB_HDF5 = -L\$(HDF5_PATH)/lib -lhdf5_fortran -lhdf5 -lz
LIB_NCMPI = -L\$(NCMPI_PATH)/lib -lpnetcdf
LIB_HYPRE = -L\$(HYPRE_PATH)/lib -lHYPRE

Finally, we have a macro to specify any platform dependent code and some macros for basic file manipulation and library tools.

MACHOBJ =

MV = mv -f
AR = ar -r
RM = rm -f
CD = cd
RL = ranlib
ECHO = echo

On most platforms, these will not need to be modified.
Chapter 33

Multithreaded FLASH

33.1 Overview

We have added the capability to run multiple threads of execution within each MPI task. The multithreading is made possible by usage of OpenMP which is described at www.openmp.org. For information about building a multithreaded FLASH application please refer to Section 5.8.

The OpenMP parallel regions do not cover all FLASH units in FLASH4. At this time we have only focussed on the units which are exercised most heavily in typical FLASH applications: these are the split and unsplit hydrodynamics solvers, the unsplit magnetohydrodynamic solvers, Gamma law and multigamma EOS, Helmholtz EOS, Multipole solver (improved version (support for 2D cylindrical and 3D cartesian)) and energy deposition. The level of OpenMP coverage will increase in future releases.

The setup script will prevent you from setting up invalid threaded applications by exiting with a conflict error. We have added the conflicts to prevent using, for example, a non thread-safe EOS in a OpenMP parallel region. Example setup lines can be found in sites/code.uchicago.edu/openmp_test_suite/test.info.

33.2 Threading strategies

We have experimented with two different threading strategies for adding an extra layer of parallelism to the standard MPI decomposition described in Chapter 8.

The first strategy makes use of the fact that the solution updates on different Paramesh blocks are independent, and so, multiple threads can safely update the solution on different blocks at the same time. This is a coarse-grained threading approach which is only applicable to Paramesh. It can be included in a FLASH application by adding threadBlockList=True to the FLASH setup line.

```fortran
 call Grid_getListOfBlocks(LEAF,blockList,blockCount)
 !$omp do schedule(static)
 do b=1,blockCount
   blockID = blockList(b)
   call update_soln_on_a_block(blockID)
 enddo
```

The second strategy parallelises the nested do loops in the kernel subroutines such that different threads are updating the solution on independent cells from the same block at the same time. This is a fine-grained threading approach which is applicable to both Paramesh and UG. It can be included in a FLASH application by adding threadWithinBlock=True to the FLASH setup line. Notice that we parallelise the outermost do loop for a given dimensionality to improve cache usage for each thread.

```fortran
#if NDIM == 3
   !$omp do schedule(static)
#endif
 do k=k0-2,kmax+2
```
The final threading option is only applicable to the energy deposition unit. In this unit we assign different rays to each thread, where the number of rays assigned is dynamic because the work per ray is not fixed. It can be included in a FLASH application by adding `threadRayTrace=True` to the FLASH setup line.

It is perfectly fine to mix these setup options except for `threadBlockList=True` and `threadWithinBlock=True`.

### 33.3 Running multithreaded FLASH

#### 33.3.1 OpenMP variables

There are OpenMP environmental variables which control the threaded runtime. The most frequently used is `OMP_NUM_THREADS` which controls the number of OpenMP threads in each parallel region. It should be exported to your environment from your current shell. In the following example we specify that each OpenMP parallel region will be executed by 4 threads (assuming bash shell)

```bash
export OMP_NUM_THREADS=4
```

You may also need to set `OMP_STACKSIZE` to increase the stack size of the threads created by the OpenMP runtime. This is because some test problems exceed the default limit which leads to a segmentation fault from an innocent piece of source code. In practice we have only encountered this situation in a White Dwarf Supernova simulation with block list threading on a machine which had a default stack size of 4MB. For safety we now set a default value of 16MB for all of our runs to provide plenty of stack space for each thread.

```bash
export OMP_STACKSIZE=16M
```

If you need to use a job submission script to run FLASH on compute nodes then you should export the OpenMP variables in your job submission script.

#### 33.3.2 FLASH variables

There are FLASH runtime parameters which can be set to `.true.`, or `.false.` to turn on/off the parallel regions in different units. They are

<table>
<thead>
<tr>
<th>Unit</th>
<th>Block list threading</th>
<th>Within block threading</th>
<th>Ray trace threading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hydro</td>
<td>threadHydroBlockList</td>
<td>threadHydroWithinBlock</td>
<td>N/A</td>
</tr>
<tr>
<td>EOS</td>
<td>N/A</td>
<td>threadEosWithinBlock</td>
<td>N/A</td>
</tr>
<tr>
<td>Multipole</td>
<td>threadMpoleBlockList</td>
<td>threadMpoleWithinBlock</td>
<td>N/A</td>
</tr>
<tr>
<td>Energy Deposition</td>
<td>N/A</td>
<td></td>
<td>threadRayTrace</td>
</tr>
</tbody>
</table>

There is no parameter for block list threading of EOS because, quite simply, there is no block list in EOS. In many FLASH simulations the EOS subroutines are only called from the hydrodynamic subroutines which means the EOS will be called in parallel if you thread the Hydro unit block list loop.

In general you do not need to worry about these runtime parameters because the default values will be set according to the setup variable passed to the setup script. This means that if you setup a Sedov simulation with block list threading then `threadHydroBlockList` will default to `.true.` and `threadHydroWithinBlock` will default to `.false.`. The runtime control of parallel regions may be useful if we ever need to investigate a possible multithreaded bug in a particular unit.
### 33.3.3 FLASH constants

There are FLASH constant runtime parameters which describe the threading strategies included in your FLASH application. The constants are named `threadBlockListBuild`, `threadWithinBlockBuild` and `threadRayTraceBuild`. We query these constants in the unit initialization file (e.g. `Hydro_init.F90`) to verify that your FLASH application supports the parallel regions requested by the runtime parameters in 33.3.2. If the parallel regions are not supported you will get a warning message in your FLASH log file and the parallel regions will be disabled. If this happens you can still use the requested parallel regions but you must re-set up FLASH with the appropriate threading strategy. In general you do not need to worry about the existence of these constants because if you setup the application with, say, block list threading it only makes sense for you to adjust the block list threading runtime parameters in your flash.par.

A nice property of using constant runtime parameters instead of pre-processor defines is that `-noclobber` rebuilds are extremely fast. This means you can reset up a `threadBlockList` application with `threadWithinBlock` and only rebuild the handful of source files that actually changed. If we used pre-processor defines to identify the thread strategy then we would have to rebuild every single file because any file could make use of the define. We actually take advantage of the forced rebuild from changing/adding/removing a pre-processor define when we switch from a threaded to a non-threaded application. We do this because OpenMP directives are conditionally compiled when a compiler option such as `-fopenmp` (GNU) is added to the compile line. Any source file could contain OpenMP and so we must do a complete rebuild to ensure correctness. We force a complete rebuild by defining `FLASH_OPENMP` in `Flash.h` for any type of multithreaded build.

### 33.4 Verifying correctness

The hybrid (MPI+OpenMP) tests using only the multithreaded Hydro and Eos units have the nice property that they can give identical results to the corresponding MPI only test. I emphasise can because it is possible for an optimized build of the MPI+OpenMP application to give a different answer to the MPI only application. This is because the OpenMP directives can impede the compiler from optimizing the floating point calculations. Therefore, to obtain an exact match between an unthreaded and threaded test you should add a compiler option which prevents the compiler from making any aggressive floating point transformations. For the Intel compiler we needed to add `-fp-model precise` to both the MPI-only and MPI+OpenMP compiler flags to get the same answer. It should be possible to get identical answers for tests like Sod 25.1.1 and Sedov 25.1.4.

This approach does not hold if you use the multithreaded Multipole or energy deposition unit because the floating point accumulation order is different to the order in the unthreaded versions. In the multithreading of the Multipole solver we give each thread its own private moments array and then accumulate the private moment arrays into a single moments array. Similarly, in the energy deposition unit we give each thread a private energy deposition array which is then accumulated into `DEPO_VAR` mesh variable. It is reasonable to expect a small discrepancy, e.g. a magnitude error of 1E-14 in `DEPO_VAR`, in a single time step, but be aware that over a large number of time steps the accumulation of small differences can become much larger.

### 33.5 Performance results

We present some performance results for each of the multithreaded FLASH units. It is important to note that the multithreaded speedup of each unit is not necessarily representative of a full application because a full application spends time in units which are not (currently) threaded, such as Paramesh, Particles and I/O.

All of the experiments are run on Eureka, a machine at Argonne National Laboratory, which contains 100 compute nodes each with 2 x quad-core Xeon w/32GB RAM. We make use of 1 node and run FLASH with 1 MPI task and 1-8 OpenMP threads.

### 33.5.1 Multipole solver

We tested the MacLaurin Spheroid problem described in Section 25.3.4 with the new Multipole solver described in Section 8.10.2.2 using three FLASH applications: one with Paramesh and block list threading,
one with Paramesh and within block threading and one with UG and within block threading. The FLASH
setup lines are

```
./setup unitTest/Gravity/Poisson3 -auto -3d -maxblocks=600 -opt \\
+pm4dev +newMpole +noio threadBlockList=True timeMultipole=True
```

```
./setup unitTest/Gravity/Poisson3 -auto -3d -maxblocks=600 -opt \\
+pm4dev +newMpole +noio threadWithinBlock=True timeMultipole=True
```

```
./setup unitTest/Gravity/Poisson3 -auto -3d +ug -opt +newMpole +noio \\
threadWithinBlock=True timeMultipole=True -nxb=64 -nyb=64 -nzb=64
```

The effective resolution of the experiments are the same: the Paramesh experiments have `lrefine_max=4`
and blocks containing 8 internal cells along each dimension and the UG experiment has a single block
containing 64 internal cells along each dimension. We add the setup variable `timeMultipole` to include
a custom implementation of `Gravity_potentialListOfBlocks.F90` which repeats the Poisson solve 100
times.

![Figure 33.1: Speedup of the threaded Multipole solver in the Maclaurin Spheroid test problem.](image)

The results in Figure 33.1 demonstrate multithreaded speedup for three different configurations of
FLASH. The speedup for the AMR `threadWithinBlock` application is about the same for 4 threads as
it is for 5,6 and 7 threads. This is because we parallelize only the slowest varying dimension with a static
loop schedule and so some threads are assigned 64 cells per block whilst others are assigned 128 cells per
block. We do not see the same issue with the UG `threadWithinBlock` application because the greater
number of cells per block avoids the significant load imbalance per thread.

### 33.5.2 Helmholtz EOS

We tested the Helmholtz EOS unit test described in Section 15.6 with a UG and a NoFBS FLASH application.
The FLASH setup lines are

```
./setup unitTest/Eos/Helmholtz -auto -3d +ug threadWithinBlock=True \\
  timeEosUnitTest=True +noio
```
33.5. PERFORMANCE RESULTS

./setup unitTest/Eos/Helmholtz -auto -3d +nofbs threadWithinBlock=True \ timeEosUnitTest=True +noio

The `timeEosUnitTest` setup variable includes a version of `Eos_unit.F90` which calls `Eos_wrapped` 30,000 times in `MODE_DENS_TEMP`.

![Speedup of the threaded Helmholtz EOS in the EOS unit test.](image_url)

The results in Figure 33.2 show that the UG version gives better speedup than the NoFBS version. This is most likely because the NoFBS version contains dynamic memory allocation / deallocation within `Eos_wrapped`.

The UG application does not show speedup beyond 4 threads. This could be because the work per thread for each invocation of `Eos_wrapped` is too small, and so it would be interesting to re-run this exact test with blocks larger than `8 \times 8 \times 8` cells. Unfortunately the Helmholtz EOS unit test does not currently support larger blocks – this is a limitation of the Helmholtz EOS unit test and not Helmholtz EOS.

33.5.3 Sedov

We ran 2d and 3d multithreaded versions of the standard Sedov simulation described in Section 25.1.4. We tested both split and unsplit hydrodynamic solvers and both `threadBlockList` and `threadWithinBlock` threading strategies. The 2d setup lines are

```
./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \ -nxb=16 -nyb=16 threadBlockList=True +uhd

./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \ -nxb=16 -nyb=16 threadWithinBlock=True +uhd

./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \ -nxb=16 -nyb=16 threadBlockList=True

./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \ -nxb=16 -nyb=16 threadWithinBlock=True
```

and the 3d setup lines are
The results are shown in Figures 33.3 and 33.4. The speedup graph is based on times obtained from the “hydro” timer label in the FLASH log file.

![Speedup Graph](image)

Figure 33.3: Speedup of the threaded hydrodynamic solvers in a 2d Sedov test.

Part of the reason for loss of speedup in Figures 33.3 and 33.4 is that the hydrodynamic solvers in FLASH call Paramesh subroutines to exchange guard cells and conserve fluxes. Paramesh is not (yet) multithreaded and so we call these subroutines with a single thread. All other threads wait at a barrier until the single thread has completed execution.

The unsplit hydro solver has one guardcell fill per hydro timestep and the split hydro solver has NDIM guardcell fills per hydro timestep (one per directional sweep). The extra guardcell fills are part of the reason that the split applications generally give worse speedup than the unsplit applications.

### 33.5.4 LaserSlab

We finish with a scaling test for the ray trace in the energy deposition unit. We setup the LaserSlab simulation described in Section 25.7.5 with the additional setup option threadRayTrace=True. The setup line and our parameter modifications are

```bash
./setup LaserSlab -2d -auto +pm4dev -geometry=cylindrical \ 
-uxb=16 -nyb=16 +ntmmmt +laser +mgd +uhd3t species=cham,targ \ 
-mgd_meshgroups=6 -without-unit=flashUtilities/contiguousConversion \ 
-without-unit=flashUtilities/interpolation/oneDim \
```
33.6. Conclusion

At the current time we have only really exposed an extra layer of parallelism in FLASH and have not yet focused on tuning the multithreading. We do not have enough experience to suggest the most efficient ways to run the multithreaded code, however, we can suggest some things that may help the efficiency. In an application setup with `threadBlockList` it makes sense to maintain at least 5 blocks per thread. This is because the computational load imbalance between thread A being assigned 1 block and thread B being assigned 2 blocks is larger than thread A being assigned 5 blocks and thread B being assigned 6 blocks. In an application setup with `threadWithinBlock` you should probably use larger blocks, perhaps $16 \times 16 \times 16$ or even $32 \times 32 \times 32$, so that each thread has more cells to work on.

As a closing note, you should be aware of the amount of time spent in the threaded FLASH units compared to the non-threaded FLASH units in your particular FLASH application - perfect speedup in a threaded unit may be insignificant if most of the time is spent in a non-threaded FLASH unit.

Figure 33.4: Speedup of the threaded hydrodynamic solvers in a 3d Sedov test.

```
-parfile=coldstart_pm_rz.par threadRayTrace=True +noio

ed_maxRayCount = 80000
ed_numRays_1 = 50000
nend = 20

The speedup is calculated from times obtained from the “Transport Rays” label in the FLASH log file and the speedup graph is shown in Figure 33.5. The timer records how long it takes to follow all rays through the complete domain during the timestep. As an aside we remove the flashUtilities units in the setup line to keep the number of line continuation characters in a setup script generated Fortran file less than 39. This is just a workaround for picky compilers; another strategy could be a compiler option that allows compilation of Fortran files containing more than 39 continuation characters.
Figure 33.5: Speedup of the energy deposition ray trace in the LaserSlab test.
References

Bader, G. & Deuflhard, P. 1983, NuMat, 41, 373
Balsara, D. S., 2001, JCP, 174, 614
Balsara, D. S., & Spicer, D. S. 1999, JCP, 149, 270
Barnes, J. & Hut, P. 1986, Nature, 324, 446
Berger, M. J. & Collela, P. 1989, JCP, 82, 64
Berger, M. J. & Oliger, J. 1984, JCP, 53, 484
Brackbill, J. & Barnes, D. C. 1980 JCP, 35, 426
Brio, M. & Wu, C. C. 1988 JCP, 75, 400
Brysk, H., Campbell, P.M., Hammerling, P. 1974, Plasma Physics, 17, 473
Caughlan, G. R. & Fowler, W. A. 1988, Atomic Data and Nuclear Data Tables, 40, 283
Chandrasekhar, S. 1939, An Introduction to the Study of Stellar Structure (Toronto: Dover)


Chapman, D. L. 1899, Philos. Mag., 47, 90


Colella, P. & Glaz, H. M. 1985, JCP, 59, 264

Colella, P. & Woodward, P. 1984, JCP, 54, 174


DeZeeuw, D. & Powell, K. G. 1993, JCP, 104, 56


Emery, A. F. 1968, JCP, 2, 306


Ewald, P. P. 1921, Ann. Phys., 64, 253


Federrath, C., Roman-Duval, J., Klessen, R. S., Schmidt, W., Mac Low, M.-M. 2010, A&A, 512, A81


Forester, C. K. 1977, JCP, 23, 1


Gardiner, T., & Stone, J. 2005, JCP, 205, 509

Gittings, M., et al. 2008, Computational Science and Discovery, 1, 015005

Godunov, S. K. 1959, Mat. Sbornik, 47, 271
33.6. CONCLUSION

Harten, A. 1983, JCP, 49, 357–393
James, R. A. 1977, JCP, 25, 71
Jeans, J. H. 1902, Phil. Trans. Roy. Soc. (London), 199, 1
Khokhlov, A. M. 1997, Naval Research Lab memo 6406-97-7950
Kurganov, A., & Tadmor, E. 2000, JCP, 160, 241
Lee, D. 2006, Ph.D. Dissertation, Univ. of Maryland
Lee, D. & Deane, A. 2009, JCP, 228, 952
Lee, D. 2012, JCP, submitted
Li, S., & Li, H. 2004, JCP, 199, 1
Li, S. 2005, JCP, 203, 344–357
LeVeque, R. J. 1997, JCP, 131, 327
Lowrie, R. 2008, Shock Waves, 18, 129
MacNeice, P., Olson, K. M., Mobarry, C., de Fainchtein, R., & Packer, C. 1999, CPC, 126, 3
Marder, B. 1987, JCP, 68, 48


Miyoshi, T. & Kusano K. 2005, JCP, 208, 315–344


Munz, C. D., Omnes, P., Schneider, R., Sonnendrücker, & Voß, U. 2000, JCP, 161, 484

Nadyozhin, D. K. 1974, Nauchnye informatsii Astron, Sov. USSR, 32, 33

Noh, W. F. 1987, JCP, 72, 78

Orszag, S. A., Tang, C.-M., 1979, JFM, 90, 129


Parashar, M., 1999, private communication (http://www.caip.rutgers.edu/~parashar/DAGH)


Ricker, P. M. 2007, arXiv: 0710.4397


Sedov, L. I. 1959, Similarity and Dimensional Methods in Mechanics (New York: Academic)


Shu, C.-W. & Osher, S. 1989, JCP, 83, 32


Spitzer, L. 1962, Physics of Fully Ionized Gases. (New York: Wiley)

Sportisse, B. 2000, JCP, 161, 140
33.6. CONCLUSION

Sod, G. 1978, JCP, 27, 1
Toro, E. F. 1999, Riemann Solvers and Numerical Methods for Fluid Dynamics, 2nd Ed. (New York:
Springer)
Tóth, G. 2000, JCP, 161, 605
Weide, K. 2012 (in print) HEDP, doi:10.1016/j.hedp.2012.08.001
van Leer, B. 1979, JCP, 32, 101
Soc.)
Williams, F. A. 1988, Combustion Theory (Menlo Park: Benjamin-Cummings)
Williamson, J. H. 1980, JCP, 35, 48
Winske, D. and Leroy, M. M., Diffuse ions produced by electromagnetic ion beam instabilities, Journal of
Winske, D. and Quest, K. B., Electromagnetic Ion Beam Instabilities: Comparison of one- and two-
Woodward, P. & Colella, P. 1984, JCP, 54, 115
Xin, J. 2000, SIAM Rev., 42, 161
Yakovlev, D. G., & Urpin, V. A. (YU) 1980 24, 303

Yee, H. C., Vinokur, M., and Djomehri, M. J. 2000, JCP, 162, 33


Zachary, A., Malagoli, A., Colella, P. 1994, SIAM, 15, 263


Runtime Parameters

Burn
- algebra, 232, 235
- enucDtFactor, 238
- odeStepper, 232, 237
- useBurnTable, 237
- useShockBurn, 237

Conductivity
- useConductivity, 326

Cosmology
- CosmologicalConstant, 323
- MaxScaleChange, 323
- OmegaMatter, 323
- OmegaRadiation, 323

Diffuse
- useDiffuse, 286
- useDiffuseTherm, 286
- useDiffuseVisc, 286

Driver
- dtmax, 316
- meshCopyCount, 96
- nend, 158
- restart, 159
- tmax, 157

EnergyDeposition
- ed_cellRayDifferentialFactor, 272
- ed_cellWallThicknessFactor, 272
- ed_computeGradNeleX, 271
- ed_computeGradNeleY, 271
- ed_computeGradNeleZ, 271
- ed_crossSectionFunctionType_n, 270
- ed_enforcePositiveNele, 271
- ed_enforcePositiveTele, 271
- ed_gaussianCenterMajor_n, 270
- ed_gaussianCenterMinor_n, 270
- ed_gaussianExponent_n, 270
- ed_gaussianRadiusMajor_n, 270
- ed_gaussianRadiusMinor_n, 270
- ed_gradOrder, 271
- ed_gridnAngularTics_n, 271
- ed_gridnRadialTics_n, 271
- ed_gridType_n, 271
- ed_ignoreBoundaryCondition_n, 270
- ed_initialRaySpeed_n, 270
- ed_laser3Din2D, 271
- ed_laser3Din2DwedgeAngle, 271
- ed_laserIOMaxNumberOfPositions, 272
- ed_laserIOMaxNumberOfRays, 272
- ed_lensSemiAxisMajor_n, 270
- ed_lensX_n, 270
- ed_lensY_n, 270
- ed_lensZ_n, 270
- ed_maxRayCount, 271
- ed_numberOfBeams, 270
- ed_numberOfPulses, 269
- ed_numberOfRays_n, 270
- ed_numberOfSections_n, 269
- ed_power_n_i, 269
- ed_printBeams, 271
- ed_printMain, 271
- ed_printPulses, 271
- ed_printRays, 271
- ed_pulseNumber_n, 270
- ed_rayDeterminism, 271
- ed_rayZeroPower, 271
- ed_saveOutOfDomainRays, 272
- ed_semiAxisMajorTorsionAngle_n, 270
- ed_semiAxisMajorTorsionAxis_n, 270
- ed_targetSemiAxisMajor_n, 270
- ed_targetSemiAxisMinor_n, 270
- ed_targetX_n, 270
- ed_targetY_n, 270
- ed_targetZ_n, 270
- ed_time_n_i, 269
- ed_useLaserIO, 272
- ed_wavelength_n, 270
- threadRayTrace, 272
- useEnergyDeposition, 272

Eos
- eintSwitch, 198, 209
- eos_coulumbMult, 228
- eos_maxNewton, 228
- eos_singleSpeciesA, 226, 228
- eos_singleSpeciesZ, 226, 228
- eos_tolerance, 228
- eosModeInit, 338
- gamma, 226, 228, 358, 368, 376
gammaEle, 226
gammaIon, 226
gammaRad, 226

Gravity
grav_boundary_type, 100, 101, 297, 298
grav_boundary_type_x, 297, 298
grav_boundary_type_y, 297, 298
grav_boundary_type_z, 297, 298
grv_bhAccErr, 296–298
grv_bhEwaldAlwaysGenerate, 298
grv_bhEwaldFieldNx, 297, 298
grv_bhEwaldFieldNy, 297, 298
grv_bhEwaldFieldNz, 297, 298
grv_bhEwaldFName, 298
grv_bhEwaldNRef, 297, 298
grv_bhEwaldSeriesN, 297, 298
grv_bhMAC, 297, 298
grv_bhMDegree, 297, 298
grv_bhNewton, 297, 298
grv_bhUseRelAccErr, 296–298
useGravity, 297

Grid
convertToConsVdForMeshCalls, 59, 60, 108, 112
convertToConsVdInMeshInterp, 59, 60, 108, 112
derefine_cutoff_, 109
eosMode, 207
eosModeInit, 207
flux_correct, 103
geometry, 147
gr_bhPhysMACComm, 141, 297
gr_bhPhysMACTW, 141, 297
gr_bhTreeLimAngle, 141
gr_bhTreeSafeBox, 134, 141
gr_bhUseUnifiedTW, 134, 141
gr_lrefineMaxRedDoByLogR, 110
gr_lrefineMaxRedDoByTime, 111
gr_lrefineMaxRedLogBase, 111
gr_lrefineMaxRedRadiusFact, 111
gr_lrefineMaxRedTimeScale, 111
gr_lrefineMaxRedTRef, 111
gr_pfftDiffOpDiscretize, 142
gr_pmrpForceConsistency, 5
iGridSize, 104
interpol_order, 107, 138
jGridSize, 104
kGridSize, 104
lrefine_max, 110
max_particles_per_blk, 109
mg_maxCorrections, 142
mg_MaxResidualNorm, 142

mg_printNorm, 142
mpole_2DSymmetryPlane, 139, 140
mpole_3DAxisymmetry, 139, 140
mpole_DumpMoments, 139, 140
mpole_IgnoreInnerZone, 139, 140
mpole_IInnerZoneResolution, 139, 140
mpole_InnerZoneSize, 139, 140
mpole_Lmax, 139, 140
mpole_Lmax, 142, 417, 418
mpole_MaxRadialZones, 140
mpole_PrintRadialInfo, 139, 140
mpole_subSample, 126
mpole_ZoneExponent_n, 140
mpole_ZoneRadiusFraction_n, 140
mpole_ZoneScalar_n, 140
mpole_ZoneType_n, 140
Nblockx, 99, 376
nblockx, 98
Nblocky, 99, 376
nblocky, 98
Nblockz, 99
nblockz, 98
pfft_setupOnce, 122
refine_cutoff_, 109
refine_filter_, 109
refine_on_particle_count, 109
refine_var_, 108, 109
smlrho, 417
tagradius, 112
xl_boundary_type, 99, 100, 376
xmax, 98, 371, 376
xmin, 98, 371
xr_boundary_type, 376
ymax, 98, 371, 376
ymin, 98, 371
zmax, 98
zmin, 98

Hydro
chomboLikeUpdateSoln, 208
hy_eosModeAfter, 207
hybrid_riemann, 200
HydroMain, 199
ppm_modifystates, 200
use_cma_flattening, 200
use_steepening, 200

IO
alwaysComputeUserVars, 171
alwaysRestrictCheckpoint, 171
checkpointFileIntervalStep, 176
checkpointFileNumber, 159
chkGuardCellsInput, 165
chkGuardCellsOutput, 165
RUNTIME PARAMETERS

outputSplitNum, 171
particleFileNumber, 159
plot_grid_var_1, 164
plot_grid_var_2, 164
plot_var_1, 159
plot_var_2, 159
plotFileNumber, 177
plotfileNumber, 159

Ioniz
dneimax, 239
dneimin, 239
tneimax, 239
tneimin, 239

Logfile
log_file, 345

Particles
particle_attribute_1, 315
particle_attribute_2, 315
pt_dtChangeToleranceDown, 309
pt_dtChangeToleranceUp, 309
pt_dtFactor, 314
pt_initialXMin, 314
pt_numX, 314
pt_numx, 176
pt_numy, 176
useParticles, 314

PhysicalConstants
pc_unitsBase, 188

RadTrans
rt_mgdFlCoef, 336
rt_mgdFlMode, 336
rt_mgdNumGroups, 336, 337
rt_useMGD, 337

Simulation
sim_maxTolCoeff0, 315
sim_maxTolCoeff1, 315
sim_maxTolCoeff2, 315
sim_schemeOrder, 315
sim_smlRho, 416

Stir
st_computeDt, 241, 242
st_decay, 240
st_energy, 240
st_infilename, 241, 242
useStir, 242

Timers
eachProcWritesSummary, 352
writeStatSummary, 352

Turb
turb_c2, 284

Viscosity
useViscosity, 327
visc_whichCoefficientIsConst, 327

turb_stepSize, 284
FLASH4 API

Driver
- Driver_abortFlash, 89, 163, 177, 348
- Driver_abortFlashC, 89
- Driver_evoeFlash, 35, 38, 86, 286, 306
- Driver_finalizeFlash, 88
- Driver_getDt, 88
- Driver_getElaspedWCTime, 88
- Driver_getNStep, 88
- Driver_getSimTime, 88
- Driver_initFlash, 19, 36, 86, 157, 176, 188
- Driver_initParallel, 86
- Driver_sendOutputData, 163

Grid
- Grid_advanceDiffusion, 142
- Grid_applyBCEdge, 101, 102
- GridApplyBCEdgeAllUnkVars, 101, 102
- Grid_bcApplyToRegion, 101, 102
- Grid_bcApplyToRegionSpecialized, 101, 102
- Grid_fillGuardCells, 34, 102, 106, 114, 117
- Grid_getBlkBoundBox, 106
- Grid_getBlkCenterCoords, 106, 107, 147
- Grid_getBlkCornerID, 107
- Grid_getBlkData, 34, 113, 147
- Grid_getBlkIndexLimits, 23, 113
- Grid_getBlkPhysicalSize, 106, 107, 147
- Grid_getBlkPtr, 113
- Grid_getBlkRefineLevel, 106
- Grid_getBlkType, 106
- Grid_getBlkPtr, 328
- Grid_getCellCoords, 23, 34, 107, 147
- Grid_getDeltas, 147
- Grid_getGeometry, 147
- Grid_getListOfBlocks, 35, 113
- Grid_getPlaneData, 113, 147
- Grid_getPointData, 113, 147
- Grid_getRowData, 113, 147
- Grid_getSingleCellVol, 147
- Grid_init, 98, 113
- Grid_initDomain, 98, 113
- Grid_mapParticlesToMesh, 298, 313
- Grid_markRefineDerefine, 17, 110
- Grid_markRefineSpecialized, 110
- Grid_pfft, 122
- Grid_pfftFinalize, 122
- Grid_pfftInit, 122
- Grid_putBlkData, 34, 113
- Grid_restrictAllLevels, 5
- Grid_sendOutputData, 114, 163
- Grid_solvePoisson, 137
- Grid_updateRefinement, 34, 35, 106, 114

IO
- IO_getPrevScalar, 163
- IO_getScalar, 163
- IO_init, 157, 177
- IO_output, 157, 159, 171
- IO_readUserArray, 163
- IO_setScalar, 163, 166
- IO_writeCheckpoint, 157, 171
- IO_writeIntegralQuantities, 161, 315
- IO_writeParticles, 161
- IO_writePlotfile, 159, 171
- IO_writeUserArray, 163

monitors
- Logfile
  - Logfile_init, 345
  - Logfile_stamp, 348
  - Logfile_stampMessage, 348
  - Logfile_writeSummary, 349
- Timers
  - Timers_create, 352
  - Timers_getSummary, 352
  - Timers_init, 352
  - Timers_reset, 352
  - Timers_start, 352
  - Timers_stop, 352
- Multispecies
  - Multispecies_getAvg, 184
  - Multispecies_getProperty, 183
  - Multispecies_getSum, 184
  - Multispecies_getSumFrac, 184
  - Multispecies_getSumInv, 184
  - Multispecies_getSumSqr, 184
  - Multispecies_list, 184
  - Multispecies_setProperty, 183
  - Multispecies_unitTest, 185
Particles
  - Particles_advance, 301, 306, 315
  - Particles_computeDt, 306
  - Particles_dump, 315
  - Particles_init, 315
  - Particles_initPositions, 312, 313
  - Particles_mapToMeshOneBlk, 313
  - ParticlesSpecifyMethods, 67, 312
  - Particles_unitTest, 315
  - Particles_updateAttributes, 314, 315

PhysicalConstants
  - PhysicalConstants_get, 188
  - PhysicalConstants_init, 188
  - PhysicalConstants_list, 188
  - PhysicalConstants_listUnits, 188
  - PhysicalConstants_unitTest, 188, 189

physics
  - Cosmology
    - Cosmology_cdmPowerSpectrum, 323
    - Cosmology_computeDeltaCrit, 324
    - Cosmology_computeDt, 323
    - Cosmology_computeVariance, 324
    - Cosmology_getRedshift, 322
    - Cosmology_massToLength, 324
    - Cosmology_redshiftToTime, 324
    - Cosmology_solveFriedmannEqn, 323
  - Diffuse
    - Diffuse_therm, 437
  - Eos
    - Eos, 60, 113, 221, 222, 225, 226, 228, 229
    - Eos_getData, 60
    - Eos_init, 228
    - Eos_putData, 60
    - Eos_wrapped, 60, 61, 222, 226, 229
  - Gravity
    - Gravity_accelOneRow, 138
    - Gravity_bhBotNodeContrib, 134, 296
    - Gravity_bhFillBotNode, 133
    - Gravity_bhNodeContrib, 134, 296
    - Gravity_init, 100
    - Gravity_potentialListOfBlocks, 297
  - RadTrans
    - RadTrans_mgdEFromT, 338
    - RadTrans_sumEnergy, 341

RuntimeParameters
  - RuntimeParameters_get, 176, 177
  - RuntimeParameters_getPrev, 163
  - RuntimeParameters_mapStrToInt, 99, 100
  - RuntimeParameters_set, 177

Simulation
  - Simulation_adjustEvolution, 340
  - Simulation_defineDomain, 98, 100, 101, 361, 376
Index

.dump_checkpoint, see IO
.dump_restart, see IO
subunit, 36
  ptherimary, 36
adaptive mesh, 104
adiabatic indices, 222
angular coordinates, 98
block, 95
boundary conditions, 99
computational domain, 98
Config, 17, 18
double precision, 23
energy deposition, 244
  laser: 3D ray tracing in 2D domain, 259
  laser: beam cross section power structure, 257
  laser: beam definition and setup, 252
  laser: Cubic interpolation, 249
  laser: implementation details (algorithm), 244, 246, 264
  laser: Kaiser algorithm, 247
  laser: power deposition, 245
  laser: proper beam placement, 254, 261
  laser: pulse definition, 251
  laser: ray definition and setup, 255, 258, 262
  laser: ray grid inside beam, 255, 257
  laser: synchronous and asynchronous ray tracing, 265
equation of state, 221, 222
  Eos, 228
  Eos_wrapped, 229
  gamma-law, 222
  Helmholtz, 222, 223
  multi-gamma, 222, 223
  rhd-gamma, 222

fidlr
  xflash3, 12
FIXEDBLOCKSIZE mode, 76, 82, 103
Flash.h, 75
flash.par, 11, 25, 159, 346

flux conservation, 106
geometry, see mesh
gmake
  Makefile.h, 10
grid
  boundary, see boundary conditions
data averaging, 106
guardcells, see guardcells
  interpolation, 103, 106, 107, 138, 145, 150
  order, 107
  prolongation, 107, 150
  restriction, 107, 150, 151
  xmin/xmax, 98
  ymin/ymax, 98
  zmin/zmax, 98
Grid_getCellCoords, 23
guardcells, 95
HDF5, see IO
I/O, 154
IDL, 10
IO
  .dump_checkpoint, 158
  .dump_particle_file, 161
  .dump_plotfile, 159
  .dump_restart, 158
  checkpoint file, 157
  forced Plotfile, 159
  HDF5, 165
  output file names, 164
  particle file, 160
  plotfile, 159
kernel, 37

make
  make clean, 10
    parallel build, 10
gmake
  make realclean, 10
Makefile.h, 39
MAXBLOCKS, 42, 105
memory, 104, 105
mesh
  geometry, 145
  CARTESIAN, 147
  Cartesian, 148
  CYLINDRICAL, 147
  cylindrical, 149
  POLAR, 147
  polar, 150
  SPHERICAL, 147
  spherical, 150
  refinement criteria, 21

NONFIXEDBLOCKSIZE mode, 3, 76, 104, 113

PARAMESH, 104
paramesh, 96

Physical Constants, 187
  CGS, CGS, 187
  MKS, MKS, 187

sedov.dat, 12
sedov.log, 12

setup
  +default, 54
  -[123]d, 42
  -auto, 41
  -curvilinear, 43
  -debug, 42
  -defines, 43
  -fbs, 43
  -geometry, 43
  -gridinterpolation, 44, 138
  -index-reorder, 44
  -library, 46
  -makefile, 44
  -makehide, 44
  -maxblocks, 42
  -n[xyz]b, 42
  -noclobber, 45
  -nofbs, 43
  -objdir, 42
  -opt, 42
  -os, 45
  -parfile, 45
  -particlemethods, 45
  -portable, 45
  -site, 45
  -tau, 46
  -test, 42
  -unit, 42
  -unitsfile, 45
  -verbose, 41
  -with-library, 46
  -with-unit, 42
  -without-library, 46
  -without-unit, 46
  8wave, 48
  cartesian, 49
  chombo_amr, 47
  chombo_ug, 47
  cube16, 48
  cube32, 48
  cube64, 48
curv-pm2, 49
curvilinear, 49
cylindrical, 49
default, 46
ed_maxBeams, 53
ed_maxPulses, 53
grid, 47
GridIndexOrder, 52

hdf5, 46
hdf5TypeIO, 47
IO, 51
io, 46
laser, 51
longrange, 50
maxBlocks, 52
mgd, 51
mgd_meshgroups, 53
mpi1, 50
mpi2, 50
mpole, 49
Mtmmmt, 52
mtmmmt, 50
nDim, 51
newMpole, 50
noDefaultMpole, 50
nofbs, 49
noio, 46
nolog, 47
noMgrid, 50
npg, 49
nxb, 52
nyb, 52
nzb, 52
parallelIO, 46, 51
ParameshLibraryMode, 52
PfftSolver, 52
pic, 51
pm2, 47
pm3, 47
pm40, 47
pm4dev, 47
pm4dev_clean, 47
pnetcdf, 46
pnetcdfTypeIO, 47
polar, 49
ptdens, 49
ptio, 48
pureHydro, 48
rnf, 48
serialIO, 46
shortcuts, 165, 171
species, 53
spherical, 49
spherical-pm2, 49
supportPPMUupwind, 48
supportWENO, 48
threadEosWithinBlock, 53
threadHydroBlockList, 53
threadHydroWithinBlock, 53
threadMpoleBlockList, 53
threadMpoleWithinBlock, 54
threadRayTrace, 53
ug, 47
uhd, 48
uhd3t, 50
unsplitHydro, 48
usm, 48
usm3t, 50
setup, 10, 39
setup_params, 27
setups
  example, 17
  sedov, 10
shortcuts, 156
Simulation_data, 19
Simulation_init, 19
Simulation_initBlock, 17, 22
  creating new, 17

unit, 33

variable
type
  conserved, 59
  mass-specific, 59
variable type, 59

XFLASH3_DIR, 12