FLASH User’s Guide

Version 4.0-alpha
April 2011 (last updated June 8, 2011)

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."

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

- Public release. We expect to be able to publicly release versions of the Flash Code via the Center’s web site, and expect to release any given version of the Flash Code within a year of its formal deposition in the Center’s code archive.

- 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 Director and the Management Committee; decisions related to public release of the Flash Code will be made in the same manner.

- Distribution rights. The Flash Code, and any part of this code, can only be released and distributed by the Flash Center; individual users of the code are not free to re-distribute the Flash Code, or any of its components, outside the Center. While the Flash Code is currently not export-controlled, we are nevertheless required to insure that we identify all users of this code. For this reason, we require that all users sign a hardcopy version of this License Agreement, and send it to the Flash Center. Distribution of the Flash Code can only occur once a signed License Agreement is received by us.

- Modifications and Acknowledgments. You may make modifications to the Flash Code, and we encourage you to send such modifications to the Center; as noted above, you are not free to distribute this code to others. As resources permit, we plan to incorporate such modifications in subsequent releases of the Flash Code, and we will acknowledge your contributions. Note that modifications that do not make it into an officially-released version of Flash will not be supported by us.

If you do modify a copy or copies of the Flash Code or any portion of it, thus forming a work based on the Flash Code, you must meet the following conditions:

a) The software must carry prominent notices stating that you changed specified portions of the Flash Code. This will assist us in clearly identifying the portions of the code that you have contributed.

b) The software must display the following acknowledgment:

"This product includes software developed by and/or derived from the DOE NNSA-ASC OASCR Flash Center (http://flash.uchicago.edu) to which the U.S. Government retains certain rights."
c)

d) Commercial use. All users interested in commercial use of the Flash Code must obtain prior written approval from the director of the Flash 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 code dissemination process, the Center has set up and will maintain as part of its web site mechanisms for announcing new code releases, for collecting requests for code use, and for collecting and disseminating frequently asked questions (FAQs). We do not plan to provide direct technical support, however, we do support a list (flash-users@flash.uchicago.edu) for discussion of user’s questions. There is also a list (flash-bugs@flash.uchicago.edu) for reporting bugs.

- Use 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 this code, or modified versions of this code or its components. All such information can be posted on the Center’s web site, at http://flash.uchicago.edu.

- The Flash Code was prepared, in part, as an account of work sponsored by an agency of the United States Government. Neither the United States, nor the University of Chicago, nor any contributors to the Flash Code, nor any of their employees, makes any warranty, express or implied, 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 CS/Applications Group is supported by the DOE NNSA-ASC OASCR FLASH Center at the University of Chicago. Some of the test calculations described here were performed on machines at LLNL, LANL, San Diego Supercomputing Center, and ANL. The current members of the Group include:

John Bachan, Sean Couch, Christopher Daley, Anshu Dubey, Milad Fatenejad, Norbert Flocke, J. Brad Gallagher, Shravan K. Gopal, Dongwook Lee, Samuel Olofin, Marcos Vanella, Klaus Weide,

Current Flash Center members outside the group who made significant contributions to the FLASH code include:

Randy Hudson, George Calhoun Jordan IV

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.
5.5 Config Files .................................................. 48
5.5.1 Configuration file syntax .............................. 48
5.6 Creating a Site-specific Makefile ...................... 56
5.7 Files Created During the setup Process ............... 57
5.7.1 Informational files .................................. 57
5.7.2 Code generated by the setup call ................... 57
5.7.3 Makefiles generated by setup ........................ 58
5.8 Setup FLASH applications that use Chombo library .... 59
5.8.1 Overview ............................................... 59
5.8.2 Build procedure ...................................... 59

6 The Flash.h file ........................................... 65
6.1 UNK, FACE(XYZ) Dimensions .......................... 65
6.2 Property Variables, Species and Mass Scalars ......... 66
6.3 Fluxes .................................................. 67
6.4 Scratch Grid Vars ...................................... 67
6.5 Fluid Variables Example ................................ 67
6.6 Particles .............................................. 69
6.6.1 Particles Types ...................................... 69
6.6.2 Particles Properties ................................ 69
6.7 Non-Replicated Variable Arrays ....................... 70
6.7.1 Per-Array Macros ................................... 70
6.7.2 Array Partitioning Macros .......................... 70
6.7.3 Example ........................................... 71
6.8 Other Preprocessor Symbols ............................ 71

III Driver Unit ............................................. 73
7 Driver Unit .............................................. 75
7.1 Driver Routines ........................................ 75
7.1.1 Driver_initFlash .................................... 76
7.1.2 Driver_evolveFlash .................................. 76
7.1.3 Driver_finalizeFlash ................................. 78
7.1.4 Driver accessor functions ........................... 78

IV Infrastructure Units ................................... 81
8 Grid Unit .................................................. 83
8.1 Overview .............................................. 85
8.2 GridMain Data Structures ............................... 86
8.3 Computational Domain .................................. 87
8.4 Boundary Conditions .................................... 88
8.4.1 Boundary Condition Types .......................... 89
8.4.2 Boundary Conditions at Obstacles .................. 89
8.4.3 Implementing Boundary Conditions ................. 89
8.5 Uniform Grid ........................................... 92
8.5.1 FIXEDBLOCKSIZE Mode ............................. 92
8.5.2 NONFIXEDBLOCKSIZE mode ........................ 92
8.6 Adaptive Mesh Refinement (AMR) Grid with Paramesh .. 93
8.6.1 Additional Data Structures .......................... 95
8.6.2 Grid Interpolation .................................... 96
8.6.3 Refinement .......................................... 97
8.7 Chombo ................................................ 99
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.7.1 Using Chombo in a UG configuration</td>
<td>100</td>
</tr>
<tr>
<td>8.7.2 Using Chombo in a AMR configuration</td>
<td>100</td>
</tr>
<tr>
<td>8.8 GridMain Usage</td>
<td>101</td>
</tr>
<tr>
<td>8.9 GridParticles</td>
<td>103</td>
</tr>
<tr>
<td>8.9.1 GridParticlesMove</td>
<td>104</td>
</tr>
<tr>
<td>8.9.2 GridParticlesMapToMesh</td>
<td>106</td>
</tr>
<tr>
<td>8.10 GridSolvers</td>
<td>109</td>
</tr>
<tr>
<td>8.11 Grid Geometry</td>
<td>130</td>
</tr>
<tr>
<td>8.11.1 Understanding Curvilinear</td>
<td>131</td>
</tr>
<tr>
<td>8.11.2 Choosing a Geometry</td>
<td>132</td>
</tr>
<tr>
<td>8.11.3 Geometry Information in Code</td>
<td>132</td>
</tr>
<tr>
<td>8.11.4 Available Geometries</td>
<td>133</td>
</tr>
<tr>
<td>8.11.5 Conservative Prolongation/Restriction on Non-Cartesian Grids</td>
<td>135</td>
</tr>
<tr>
<td>8.12 Custom Regions</td>
<td>136</td>
</tr>
<tr>
<td>8.12.1 Quiet Start</td>
<td>136</td>
</tr>
<tr>
<td>8.12.2 Piston</td>
<td>137</td>
</tr>
<tr>
<td>8.13 Unit Test</td>
<td>137</td>
</tr>
<tr>
<td>9 IO Unit</td>
<td>139</td>
</tr>
<tr>
<td>9.1 IO Implementations</td>
<td>141</td>
</tr>
<tr>
<td>9.2 Output Files</td>
<td>143</td>
</tr>
<tr>
<td>9.2.1 Checkpoint files - Restarting a Simulation</td>
<td>143</td>
</tr>
<tr>
<td>9.2.2 Plotfiles</td>
<td>145</td>
</tr>
<tr>
<td>9.2.3 Particle files</td>
<td>146</td>
</tr>
<tr>
<td>9.2.4 Integrated Grid Quantities - flash.dat</td>
<td>147</td>
</tr>
<tr>
<td>9.2.5 General Runtime Parameters</td>
<td>148</td>
</tr>
<tr>
<td>9.3 Restarts and Runtime Parameters</td>
<td>149</td>
</tr>
<tr>
<td>9.4 Output Scalars</td>
<td>149</td>
</tr>
<tr>
<td>9.5 Output User-defined Arrays</td>
<td>149</td>
</tr>
<tr>
<td>9.6 Output Grid Variables</td>
<td>149</td>
</tr>
<tr>
<td>9.7 Face-Centered Data</td>
<td>150</td>
</tr>
<tr>
<td>9.8 Output Filenames</td>
<td>150</td>
</tr>
<tr>
<td>9.9 Output Formats</td>
<td>150</td>
</tr>
<tr>
<td>9.9.1 HDF5</td>
<td>151</td>
</tr>
<tr>
<td>9.9.2 Parallel-NetCDF</td>
<td>157</td>
</tr>
<tr>
<td>9.9.3 Direct IO</td>
<td>157</td>
</tr>
<tr>
<td>9.9.4 Output Side Effects</td>
<td>157</td>
</tr>
<tr>
<td>9.10 Working with Output Files</td>
<td>157</td>
</tr>
<tr>
<td>9.11 Unit Test</td>
<td>157</td>
</tr>
<tr>
<td>9.11.1 Online tips for working with the IO Unit</td>
<td>159</td>
</tr>
<tr>
<td>9.12 Chombo</td>
<td>159</td>
</tr>
<tr>
<td>10 Runtime Parameters Unit</td>
<td>161</td>
</tr>
<tr>
<td>10.1 Defining Runtime Parameters</td>
<td>161</td>
</tr>
<tr>
<td>10.2 Identifying Valid Runtime Parameters</td>
<td>161</td>
</tr>
<tr>
<td>10.3 Routine Descriptions</td>
<td>162</td>
</tr>
<tr>
<td>10.4 Example Usage</td>
<td>163</td>
</tr>
</tbody>
</table>
CONTENTS

11.1 Defining Species .................................................. 165
11.2 Initializing Species Information in Simulation_initSpecies .... 166
11.3 Routine Descriptions ............................................. 167
11.4 Example Usage ................................................... 169
11.5 Unit Test .......................................................... 169

12.1 Available Constants and Units .................................... 172
12.2 Applicable Runtime Parameters .................................. 172
12.3 Routine Descriptions ............................................. 172
12.4 Unit Test .......................................................... 173

V Physics Units ......................................................... 175

13.1 Gas hydrodynamics .................................................. 179
  13.1.1 Usage .......................................................... 179
  13.1.2 The piecewise-parabolic method (PPM) ....................... 179
  13.1.3 The unsplit hydro solver ..................................... 180
  13.1.4 Multitemperature extension for Hydro ....................... 183
  13.1.5 Chombo compatible Hydro .................................... 184
  13.2 Relativistic hydrodynamics (RHD) ................................ 185
    13.2.1 Overview .................................................. 185
    13.2.2 Equations ................................................ 185
    13.2.3 Relativistic Equation of State ............................. 186
    13.2.4 Additional Runtime Parameter ............................. 186
  13.3 Magnetohydrodynamics (MHD) ................................... 186
    13.3.1 Description ................................................ 186
    13.3.2 Usage ..................................................... 187
    13.3.3 Algorithm: The Eight-wave Solver ......................... 188
    13.3.4 Algorithm: The Unsplit Staggered Mesh Solver .......... 190
    13.3.5 Non-ideal MHD ............................................ 194

14.1 Introduction ..................................................... 197
14.2 Gamma Law and Multigamma ..................................... 198
  14.2.1 Ideal Gamma Law for Relativistic Hydrodynamics .......... 199
14.3 Helmholtz ........................................................ 199
14.4 Multitemperature extension for Eos ............................. 201
  14.4.1 Gamma Law and Multigamma ................................ 201
  14.4.2 Multigamma ................................................ 203
14.5 Usage ........................................................... 203
  14.5.1 Initialization ................................................ 203
  14.5.2 Runtime Parameters ......................................... 203
  14.5.3 Direct and Wrapped Calls ................................... 204
14.6 Unit Test ........................................................ 204

15.1 Burn Unit ......................................................... 206
  15.1.1 Algorithms .................................................. 206
  15.1.2 Reaction networks .......................................... 207
  15.1.3 Detecting shocks ............................................ 211
  15.1.4 Energy generation rates and reaction rates ................ 211
15.1.5 Temperature-based timestep limiting ........................................ 212
15.2 Ionization Unit ............................................................................ 212
  15.2.1 Algorithms ............................................................................ 213
  15.2.2 Usage ................................................................................... 213
15.3 Stir Unit ....................................................................................... 213
15.4 Energy Deposition Unit ................................................................. 214
  15.4.1 Ray Tracing in the Geometric Optics Limit ............................... 214
  15.4.2 Laser Power Deposition .......................................................... 215
15.5 Usage ........................................................................................... 215
15.6 Heatexchange .............................................................................. 215

16 Diffusive Terms ............................................................................ 217
  16.1 Diffuse Unit ............................................................................... 217
    16.1.1 Diffuse Flux-Based Implementations .................................... 218
    16.1.2 Diffuse Split/Unsplit implementation ................................... 218

17 Gravity Unit ................................................................................... 223
  17.1 Introduction ............................................................................... 223
  17.2 Externally Applied Fields ............................................................. 224
    17.2.1 Constant Gravitational Field .............................................. 224
    17.2.2 Plane-parallel Gravitational field ........................................ 224
    17.2.3 Gravitational Field of a Point Mass ..................................... 224
  17.3 Self-gravity ............................................................................... 224
    17.3.1 Coupling Gravity with Hydrodynamics ............................... 225
  17.4 Usage ....................................................................................... 225
  17.5 Unit Tests .................................................................................. 226

18 Particles Unit .................................................................................. 227
  18.1 Time Integration ........................................................................ 229
    18.1.1 Active Particles (Massive) .................................................. 229
    18.1.2 Charged Particles - Hybrid PIC ......................................... 231
    18.1.3 Passive Particles ................................................................ 233
  18.2 Mesh/Particle Mapping ............................................................... 237
    18.2.1 Quadratic Mesh Mapping .................................................... 238
    18.2.2 Cloud in Cell Mapping ....................................................... 240
  18.3 Using the Particles Unit ............................................................... 240
    18.3.1 Particles Runtime Parameters ........................................... 242
    18.3.2 Particle Attributes .............................................................. 242
    18.3.3 Particle I/O ........................................................................ 242
    18.3.4 Unit Tests ........................................................................... 243

19 Cosmology Unit ............................................................................. 245
  19.1 Algorithms and Equations ........................................................... 245
  19.2 Using the Cosmology unit ............................................................ 247
  19.3 Unit Test ................................................................................... 248

20 Material Properties Units ............................................................... 249
  20.1 Thermal Conductivity ................................................................ 250
  20.2 Magnetic Resistivity .................................................................. 250
    20.2.1 Constant resistivity ............................................................ 250
  20.3 Viscosity .................................................................................. 251
  20.4 Opacity .................................................................................... 251
  20.5 Mass Diffusivity ....................................................................... 252
21 Radiative Transfer Unit

21.1 Multigroup Diffusion ............................................. 253
21.1.1 Config File and Mesh Replication .......................... 254
21.1.2 Runtime Parameters ......................................... 255
21.1.3 Specifying Initial Conditions .............................. 256
21.1.4 Summary ..................................................... 257

VI Monitor Units

22 Logfile Unit

22.1 Meta Data ....................................................... 262
22.2 Runtime Parameters, Physical Constants, and Multispecies Data ............................................... 262
22.3 Accessor Functions and Timestep Data ..................... 264
22.4 Performance Data .............................................. 265
22.5 Example Usage ................................................ 266

23 Timer and Profiler Units

23.1 Timers ............................................................. 267
23.1.1 MPINative .................................................... 267
23.1.2 Tau ............................................................ 268
23.2 Profiler ........................................................... 269

VII Simulation Units

24 The Supplied Test Problems

24.1 Hydrodynamics Test Problems .............................. 273
24.1.1 Sod Shock-Tube ............................................ 274
24.1.2 Variants of the Sod Problem in Curvilinear Geometries ......................................................... 277
24.1.3 Interacting Blast-Wave Blast2 ............................ 278
24.1.4 Sedov Explosion ............................................. 282
24.1.5 Isentropic Vortex .......................................... 287
24.1.6 Wind Tunnel With a Step .................................. 290
24.1.7 The Shu-Osher problem .................................... 295
24.1.8 Driven Turbulence StirTurb ............................... 296
24.1.9 Relativistic Sod Shock-Tube ............................. 300
24.1.10 Relativistic Two-dimensional Riemann ............... 302
24.1.11 Shafranov Shock ......................................... 304
24.2 Magnetohydrodynamics Test Problems .................. 305
24.2.1 Brio-Wu MHD Shock Tube ............................... 305
24.2.2 Orszag-Tang MHD Vortex ............................... 307
24.2.3 MHD Rotor ................................................ 312
24.2.4 MHD Current Sheet ...................................... 312
24.2.5 Field Loop ................................................ 317
24.2.6 3D MHD Blast ............................................ 317
24.3 Gravity Test Problems ....................................... 319
24.3.1 Jeans Instability .......................................... 319
24.3.2 Homologous Dust Collapse ............................. 323
24.3.3 Huang-Greengard Poisson Test ......................... 325
24.3.4 MacLaurin ................................................ 326
24.4 Particles Test Problems ...................................... 331
24.4.1 Two-particle Orbit ....................................... 331
24.4.2 Zel’dovich Pancake ..................................... 332
24.4.3 Modified Huang-Greengard Poisson Test ............ 335
24.5 Burn Test Problem ................................................. 335
  24.5.1 Cellular Nuclear Burning ................................. 335
24.6 RadTrans Test Problems ......................................... 339
  24.6.1 Infinite Medium, Energy Equilibration ................. 339
  24.6.2 Radiation Step Test ..................................... 339
24.7 Other Test Problems ............................................. 340
  24.7.1 The non-equilibrium ionization test problem .......... 340
  24.7.2 The Delta-Function Heat Conduction Problem .......... 346
  24.7.3 The HydroStatic Test Problem ........................... 346
  24.7.4 Hybrid-PIC Test Problems ............................... 346

VIII Tools ........................................................................ 351

25 VisIt ......................................................................... 355

26 Serial FLASH Output Comparison Utility (sfocu) ............. 357
  26.1 Building sfocu .................................................... 357
  26.2 Using sfocu ....................................................... 357

27 FLASH IDL Routines (fidlr3.0) ..................................... 359
  27.1 Installing and Running fidlr3.0 ............................... 359
    27.1.1 Setting Up fidlr3.0 Environment Variables ............. 359
    27.1.2 Running IDL ................................................. 360
  27.2 xflash3: A Widget Interface to Plotting FLASH Datasets .... 360
    27.2.1 File Menu ..................................................... 360
    27.2.2 Defaults Menu .............................................. 361
    27.2.3 Colormap Menu .............................................. 362
    27.2.4 X/Y plot count Menu ..................................... 362
    27.2.5 Plotting options available from the GUI ............... 362
    27.2.6 Plotting buttons ............................................ 364
  27.3 Comparing two datasets ....................................... 366

References ...................................................................... 369

Runtime Parameters ..................................................... 375

API Index .................................................................... 377

Index ............................................................................ 379
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 PnetCDF library for parallel I/O to achieve portability and scalability on a variety of different parallel computers. FLASH4-alpha 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. FLASH4-alpha 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 the time of this writing the use of Chombo for production must exercise great care.

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 scientific 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-alpha

This is the alpha version of FLASH4-alpha release, which also 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 the alpha release of FLASH4-alpha, 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-alpha 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-alpha closely follows the architecture of 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-alpha, 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 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-alpha 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 FLASH4-alpha” link on the Code Support Web Page for help on transitioning to FLASH4-alpha from FLASH2. The transition from FLASH3 to FLASH4-alpha does not require much effort from the users except in any custom implementation they may have.

The new capabilities in FLASH4-alpha include

- Ability to model 2 temperatures and radiation in the fluid
- Diffusion and conduction with implicit solvers
- Multimaterial equation-of state
- Multimaterial tabular opacities
- Radiation transfer with multigroup flux-limited diffusion
- Laser energy deposition
- Hybrid PIC (contributed by Mats Holmstrom, for support please contact matsh@irf.se)
- 3D MHD with unsplit staggered mesh implementation
- Super-time-stepping
- Oscillation suppression for slow shock handling in PPM
- Quiet start
- A new more robust implementation of the multipole method
- Chombo AMR package (fully functional in UG mode, AMR functional with restrictions)
- Mesh replication which allows processors to be divided into groups where each group gets a copy of some or all mesh variables. The non-replicated variables are distributed among the groups such that each group has a unique set.
- A new, more efficient PnetCDF base I/O implementation. The implementation is dependent upon some features of MPI-2, therefore does not work with MPI-1.
1.2 Known Issues in This Release

- Multitemperature Hydro has not been tested in 3D.

- 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.

- Unsplit diffusion implementation works in 1D or 2D, but not in 3D. The solver uses hypre-2.7.0b (which might not be backward compatible). The Split version of the solver is essentially meant for testing purposes. Moreover, it has not been exercised with PARAMESH.

- Interoperability of Unsplit hydro and diffusion solver has not been tested extensively.

- Some configurations of hydrodynamic test problems with Chombo grid show worse than expected mass and total energy conservation. Please see the Chombo section in Hydro chapter of the FLASH user guide for details.

- We have experienced the following abort when running IsentropicVortex problem with Chombo Grid: "MayDay: TreeIntVectSet.cpp:1995: Assertion ‘hxNumPts != 0’ failed. !!!" We are currently in contact with the Chombo team to resolve this issue.

- Performance may be poor when using the default settings for parallel-netcdf I/O implementation. This only affects simulations that include particles. It happens because the application does not define the total size of the file before entering data mode. The problem can be avoided by setting `nc'var'align'size=128000. See http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/VariableAlignment

- FLASH will abort in the HDF5 parallel I/O implementation when there are zero blocks on some processors. Solutions to this problem include: 1) Run your FLASH application on fewer processors, 2) Setup your FLASH application with HDF5 serial I/O implementation, 3) Use the experimental PM'argonne parallel I/O implementation, 4) Use pnetcdf I/O implementation.

- Collective I/O optimizations are always disabled in NoFbs grid applications that use HDF5 parallel I/O implementation. This is to prevent a deadlock during metadata writes. If you have a NoFbs' grid application that needs the performance of collective I/O optimizations with HDF5 library then you can setup your FLASH application with the experimental PM'argonne parallel I/O implementation.
CHAPTER 1. INTRODUCTION

• 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 the parallel HDF5 Paramesh, parallel HDF5 UG, and parallel HDF5 NoFbs implementations. Flash will fail at runtime if multiple meshes are in use without using one of these three IO implementations.

• The unsplit staggered mesh MHD solver shows slight differences in magnitudes in restart comparisons (e.g., sfocu comparison) when there are non-zero values of face-centered magnetic fields. In order to avoid such discrepancy, users can disable a PARAMESH preprocessor FORCE_CONSISTENCY_AT_SRL_INTERFACES in paramesh_preprocessor.fh.

• 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 2nd-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.

1.3 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 supplement 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 want to refer to Chapter 3 (Setting Up New Problems) and Chapter 5 (The FLASH Configuration Script).

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 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.

- 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 ViSat, described in Chapter 25.) 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.myinstitution.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 `gmake`. This will compile the FLASH code.

```
cd object
gmake
```

If you have problems and need to recompile, `gmake clean` will remove all object files from the `object/` directory, leaving the source configuration intact; `gmake realclean` will remove all files and links from `object/`. After `gmake 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 (`gmake -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:

- 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 setup FLASH without I/O by typing

```
./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](http://flash.uchicago.edu/codesupport/). 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"

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 = \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 \textit{variable} = \textit{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 \textit{N} processors, type

\begin{verbatim}
mpirun -np \textit{N} flashX
\end{verbatim}

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-alpha executable accepts an optional command-line argument for the runtime parameters file. If \texttt{“-par\_file filename”} 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

\begin{verbatim}
setenv XFLASH3\_DIR "<flash home dir>/tools/fidlr3.0"
setenv IDL\_PATH "$\{XFLASH3\_DIR\}:$IDL\_PATH"
\end{verbatim}

where \texttt{<flash home dir>} is the location of the FLASH.X.Y directory. If you get a message indicating that \texttt{IDL\_PATH} is not defined, enter

\begin{verbatim}
setenv IDL\_PATH "$\{XFLASH3\_DIR\}:$\{IDL\_DIR\}:$\{IDL\_DIR\}/lib"
\end{verbatim}
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 \texttt{XFLASH3\_DIR}. The main routine name for interactive plotting is \texttt{xflash3}.

Now run IDL (\texttt{idl} or \texttt{idl start\_linux}) and enter \texttt{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 \texttt{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 \texttt{xflash3} which files to plot. \texttt{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 27 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.
CHAPTER 2. QUICK START

Figure 2.2: The main xflash3 widget.
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/Simulation/Main. 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 gmake 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
1. Fortran module `use` statement for the `Simulation_data` module.
2. Fortran module `use` statement for the `Unit_interfaces` to access the interface of the `RuntimeParameters` unit, and any other units being used.
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:

```
idens=dBaseKeyNumber('dens')
ucons(1,i) = solnData(idens,i,j,k)
```

Now, the syntax is simpler:

```
ucons(1,i) = solnData(DENS_VAR,i,j,k)
```

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

```fortran
subroutine Simulation_init()
    use Simulation_data    
    use RuntimeParameters_interface, ONLY : RuntimeParameters_get
    implicit none
```
! get the runtime parameters relevant for this problem

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(1) :: 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 at a grid point at a time.
real :: rhoZone, velxZone, velyZone, velzZone, presZone, & enerZone, 皮肤Zone

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 real in the source code. In the next part of the code we allocate the arrays that will hold the coordinates.

FLASH Transition

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 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))
alocate(xRight(sizeX))
alocate(xCenter(sizeX))
alocate(yCoord(sizeY))
alocate(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 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 blkLimits array from 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 blkLimitsGC array should be used.

xCoord(:) = 0.0
yCoord(:) = 0.0
zCoord(:) = 0.0
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
   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
   else
      rhoZone = sim_rhoRight
      presZone = sim_pRight
      velxZone = sim_uRight * sim_xCos
      velyZone = sim_uRight * sim_yCos
      velzZone = sim_uRight * sim_zCos
   endif
3.6 The runtime parameter file (flash.par)

The FLASH executable expects a flash.par file to be present in the 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 flash.par are kept in their respective Simulation directories for easy distribution.

The flash.par file for the example setup is

```plaintext
# Density, pressure, and velocity on either side of interface
sim_rhoLeft = 1.
sim_rhoRight = 0.125
sim_pLeft = 1.
sim_pRight = 0.1
```

When Simulation_initBlock returns, the Grid data structures for physical variables have the values of the initial model for the current block. As mentioned before, Simulation_initBlock is called for every block that is created as the code refines the initial model.
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

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**

# AMR refinement parameters
lrefine_max = 6
refine_var_1 = "dens"

# These parameters are used only for the uniform grid

#iGridSize = 8  #defined as nxb * iprocs
#jGridSize = 8
#kGridSize = 1
iProcs = 1  #number or procs in the i direction
jProcs = 1
kProcs = 1

# When using UG, iProcs, jProcs and kProcs must be specified.
# These are the processors along each of the dimensions
#FIXEDBLOCKSIZE mode ::
# When using fixed blocksize, iGridSize etc are redundant in
# runtime parameters. These quantities are calculated as
# iGridSize = NXB*iprocs
# jGridSize = NYB*jprocs
# kGridSize = NZB*kprocs
#NONFIXEDBLOCKSIZE mode ::
# iGridSize etc must be specified. They constitute the global
# number of grid points in the physical domain without taking
# the guard cell into account. The local blocksize is calculated
# as iGridSize/iprocs etc.

In this example, flags are set to start the simulation from scratch and to set the grid geometry, boundary conditions, and refinement. Parameters are also set for the density, pressure and velocity values on either side of the shock, and also the angles and point of intersection of the shock with the “x” axis. Additional parameters specify details of the run, such as the number of timesteps between various output files, and the initial, minimum and final values of the timestep. The comments and alternate values at the end of the file are provided to help configure uniform grid and variably-sized array situations.

When creating the `flash.par` file, another very helpful source of information is the `setup_params` file which gets written by the `setup` script each time a problem is setup. This file lists all possible runtime parameters and their default values from the `Config` files, as well as a brief description of the parameters. It is located in the `object/` directory created at setup time.

Figure 3.1 shows the initial distribution of density for the 2-d Sod problem as setup by the example described in this chapter.

### 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

```bash
mpirun -np N flash4
```
or

```bash
mpiexec -n N flash4
```
to start a run on `N` processing entities. On many systems, you can also simply use
Figure 3.1: Image of the initial distribution of density in example setup.
3.7. RUNNING YOUR SIMULATION

./flash4

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.
CHAPTER 4. OVERVIEW OF FLASH ARCHITECTURE

FLASH Transition

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-alpha’s units, which are too complex to be described by a single file. Instead, FLASH4-alpha 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 Grid_getCellCoords, Grid_getBlkData, and Grid_putBlkData, while Grid_fillGuardCells and Grid_updateRefinement 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 they 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 an example of a stub implementation for the routine Grid_getListOfBlocks.

```
subroutine Grid_getListOfBlocks(blockType, listOfBlocks, count)

  implicit none

  integer, intent(in) :: blockType
  integer,dimension(1),intent(out) :: listOfBlocks
  integer, intent(out) :: count

  count=0
  listOfBlocks=0

  return
end subroutine Grid_updateRefinement
```

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-alpha 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/website/codesupport/.
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 Unit_init file to initialize it. The Unit_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.

subroutine Driver_evolveFlash()

    use Driver_data, ONLY: dr_myPE, dr_numProcs, dr_nb, &
    dr_nend, dr_dt, dr_wallClockTimeLimit, &


4.2 UNIT ARCHITECTURE

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 13.1.2). 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-alpha 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 14.6. 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-alpha 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. 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.

setup 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.

1if a machine has multiple hostnames, setup tries them all
FLASH Transition

In FLASH2, the Units file was located in the FLASH root directory. In FLASH4-alpha, 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-alpha 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 Basic Setup Options

The various setup options are given below. The basic options are enough to make use of the full functionality of the setup script. The advanced options of Section 5.2 often help save time while compiling/debugging your code.

-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.
5.1. BASIC SETUP OPTIONS

-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 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.

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. Resetup 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.

\footnote{Formerly, (in FLASH2) it was located in the FLASH root directory}
# 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

[-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:

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-alpha distribution directory.

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

Use the specified <unit> in setting up the problem.
### 5.2 Advanced Setup Options

This section deals with some less commonly used setup options.

- **-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 it. 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>
setup 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 for compatibility with alternative grids with FLASH4-alpha. This is supported by both the Uniform Grid as well as PARAMESH.

-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.

-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
5.2. ADVANCED SETUP OPTIONS

-particlemethods=TYPE=<particle type>[,INIT=<init method>][,MAP=<init 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 implies inclusions of appropriate implementations of ParticlesMapping and ParticlesInitialization subunits, 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 18, especially Section 18.3.

-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.

### Dependencies among libraries

If you have some libraries which depend on other libraries, create a lib/<libname>/Config which declares the dependencies. Libraries can have their own Config files, but the format is a little different. For details see Library-HOWTO.

### 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 +ug
```

equivalent to

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

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>
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

The syntax of the configuration files 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
5.5. CONFIG FILES

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 REAL variable which is allowed to take values between 0.1 and 9.9 or above 25.0. Similarly \texttt{coords} is a string variable which can take one of the four specified values.

- **D parameter-name 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,

  \begin{verbatim}
  D SOME_PARAMETER The purpose of this parameter is whatever
  D & This is a second line of description
  \end{verbatim}

  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.

- **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:

  - **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).

  - **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).

  - **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{convertToConsvdForMeshCalls} or \texttt{convertToConsvdInMeshInterp} 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 `EosWrapped` 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 `EosWrapped`, 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 `EosWrapped`, 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 `EosWrapped` 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 `EosWrapped` 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.

- **GRIDVAR name [eosmap-spec]**
  This keyword is used in connection with the grid scope scratch space supported by FLASH3. 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.

  For `eosmap-spec`, see above under `VARIABLE`. An `eosmap-spec` for `GRIDVAR` is only used when `EosWrapped` 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
  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 18.2), and initialization-method describes the method used to distribute the particles at initialization (see Section 18.3). This keyword has been introduced to facilitate inclusion of multiple particle types in the same simulation. As such it imposes certain restrictions on the use of ParticlesMapping and ParticlesInitialization subunits. None of the Particles subunits have any defaults, a PARTICLETYPE directive in a Config file (or an equivalent -particlemethods= setup option, see Table 5.3) is the only way to specify the appropriate implementations of the subunits to be used. The declaration should be accompanied by appropriate “REQUESTS” or “REQUIRES” directives to specify the paths of the 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 and initialization-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 and Lattice initialization, the following code segment should appear in the Config file of the Simulations directory.

  PARTICLETYPE passive INITMETHOD lattice MAPMETHOD quadratic
  REQUIRES Particles/ParticlesMain
  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 GRIDVAR. 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 18.3.2.

- **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.

- **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:

  ```plaintext
  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:

```
  globalparam \leq meshCopyCount \ast 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:

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

flash.par file:

```
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-wraper (`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.

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

5.7.2 Code generated by the setup call

These routines are generated by the setup call and provide simulation-specific code.
CHAPTER 5. THE FLASH CONFIGURATION SCRIPT (SETUP)

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 FLASH applications that use Chombo library

5.8.1 Overview

In FLASH4-alpha 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.

5.8.2 Build procedure

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

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

<table>
<thead>
<tr>
<th>Fortran array element:</th>
<th>Expected</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 2 1 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 2 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1 2 2 1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1 1 3 1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1 2 3 1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1 1 1 2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>1 2 1 2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>1 1 2 2</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>1 2 2 2</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>1 1 3 2</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>1 2 3 2</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>1 1 1 3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>1 2 1 3</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>1 1 2 3</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>1 2 2 3</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>1 1 3 3</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>1 2 3 3</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>1 1 1 4</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>1 2 1 4</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>1 1 2 4</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>1 2 2 4</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td>1 1 3 4</td>
<td>22</td>
<td>13</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>23</td>
<td>14</td>
</tr>
</tbody>
</table>

FAILURE! Picked up unexpected C++ assigned values

Figure 5.3: Unit test stdout for gfortran versions affected by GCC bug 40962.
5.8. SETUP FLASH APPLICATIONS THAT USE CHOMBO LIBRARY

| 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.

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 that make use of Chombo. The Chombo source code can be downloaded from https://seesar.lbl.gov/anag/chombo/. You should download Chombo-3.0-0ct2009.tar.gz because it is the only version that has been tested with FLASH. Please refer to their user guide named Chombo-3.0.pdf which is available from the same download page. This 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. Only after you have successfully built and run the Chombo unit tests described in Chombo-3.0.pdf 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 in Chombo-3.0-Oct2009.tar.gz needs to be modified in order to compile Chombo on IBM BG/P. Change line 51 from

```c
#elif defined(_POWER) || defined(_POWERPC) || defined(__POWERPC__)
```

to

```c
#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 command "./compile.sh "compiler name"", where "compiler name" is gnu, intel or ibm. 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 -DMPI_SKIP_MPCXX
-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
-lamrtimedependent2d.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-alpha 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 \n    -lamrtimedependent\$_LIB \n    -lboxtools\$_LIB \n    -lbasetools\$_LIB \n    -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-alpha 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)
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. However, in FLASH4-alpha 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 (13.9) 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 Grid Vars

In FLASH4-alpha the user is allowed to declare ‘scratch’ space for grid scope variables with the keyword GRIDVAR in the Config files. These variables are not advected or transformed in by the usual evolution steps. A user for example could declare a scratch variable to store the temperature change on the grid from one timestep to another. The scratch data structure is a 4-dimensional array per block, where the first three dimensions are the spatial dimensions, and the last dimension is for the variables. Please note that the order of the scratch data structure indices is reversed from that used in unk. Scratch variables are indexed by postpending _SCRATCH_GRID_VAR to the capitalized four letter variable defined in the Config file. Similarly to property variables, NSCRATCH_GRID_VARS, SCRATCH_GRID_VARS_BEGIN, and SCRATCH_GRID_VARS_END are defined to hold the number and endpoints of the scratch variable indices.

### 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

DEFAULT PPMKernel

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

GRIDVAR otmp

The Flash.h files would declare the property variables, fluxes and gridvar 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
#define UT_FLUX 6
#define UTT_FLUX 7

#define OTMP_SCRATCH_GRID_VAR 1
6.6 Particles

6.6.1 Particles Types

FLASH4-alpha 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

REQUESTS Particles/ParticlesMain
REQUESTS Particles/ParticlesMapping/Quadratic
REQUESTS Particles/ParticlesInitialization/Lattice
REQUESTS IO/IOMain/
REQUESTS IO/IOParticles

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

#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.

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

- **globs**: Length of the global array. This should be the value read from the array’s runtime parameter $FOO$\_NONREP\_RPCOUNT.

- **mesh**: 0-based index of the mesh in question. For a processor, this is its rank in the MESH\_ACROSS\_COMM communicator.
6.8. OTHER PREPROCESSOR SYMBOLS

- 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:

```plaintext
NONREP MASS_SCALAR FOO 3 numFoo foo?
```

`flash.par`:

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

Fortran 90:

```fortran
#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

call Driver_getMype(MESH_ACROSS_COMM, mesh)
call Driver_getNumProcs(MESH_ACROSS_COMM, nmesh)
call RuntimeParameters_get(FOO_NONREP_RPCOUNT, nfooglob)
nfooloc = NONREP_NLOCS(mesh, nmesh, nfooglob)\n! iterate over the local subset for this mesh
do fooloc=1, nfooloc
  ! get the location in UNK
  unk = FOO_NONREP_LOC2UNK(fooloc)
  ! get the global index
  fooglob = NONREP_LOC2GLOB(fooloc, mesh, nmesh)\n  ! you have what you need, do your worst...
end do
```

6.8 Other Preprocessor Symbols

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

Each `Config` file can include the `PPDEFINE` keyword to define additional preprocessor symbols. Each “`PPDEFINE symbol [value]`” gets translated to a “`#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

Figure 7.1: The Driver unit directory tree.

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:

```
program Flash
    implicit none
    call Driver_initParallel()
    call Driver_initFlash()
    call Driver_evolveFlash()
    call Driver_finalizeFlash()
```
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-alpha 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 13.3.4) and the unsplit gas hydrodynamics solver (Section 13.1.3). 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.
7.1.2.3 Super-Time-Stepping (STS)

A new timestepping method implemented in FLASH4-alpha 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 recently (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

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

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 \(\nu \rightarrow 0\), 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

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

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 described 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 logfile 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.

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

In FLASH4-alpha 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 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-alpha 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.
In FLASH4-alpha 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
Chapter 8

Grid Unit

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

Figure 8.3: The Grid unit: structure of GridSolvers 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 (\(n_{xb}\) in the \(x\)-direction, \(n_{yb}\) in the \(y\)-direction, and \(n_{zb}\) in the \(z\)-direction). A perimeter of guard cells, 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-alpha 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

---

**Figure 8.4**: A single 2-D block showing the interior cells (shaded) and the perimeter of guard cells.
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 21 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
eight-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-alpha, 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.

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, nz
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
replicating, 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 data structure for storing auxiliary fluid variables. The data structure Scratch 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 Scratch is dimensioned as `array(SCRATCH_GRID_VARS,nxb+1,nyb+1,nzb+1,blocks)`. For instance, the unsplit MHD solver StaggeredMesh discussed in Section 13.3.4 gives an example of GRIDVAR use. 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

---

\(^1\)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.
is effective because the resultant resolution in any section is related to the demands of the initial conditions there.

### FLASH Transition

FLASH2 supported only an AMR grid with paramesh 2. At initialization, it created the coarsest level initial blocks covering the domain using an algorithm called “sequential” divide domain. A uniform grid of blocks on processor zero was created, and until the first refinement, all blocks were on processor zero. FLASH3 onwards the paramesh based implementation of the Grid uses a “parallel” domain creation algorithm that attempts to create the initial domain in blocks that are distributed amongst all processors according to the same Morton ordering used by PARAMESH.

First, the parallel algorithm computes a Morton number for each block in the coarsest level uniform grid, producing a sorted list of Morton numbers for all blocks to be created. Each processor will create the blocks from a section of this list, and each processor determines how big its section will be. After that, each processor loops over all the blocks on the top level, computing Morton numbers for each, finding them in the sorted list, and determining if this block is in its own section. If it is, the processor creates the block. Parallel divide domain is especially useful in three-dimensional problems where memory constraints can sometimes force the initial domain to be unrealistically coarse with a sequential divide domain algorithm.

## 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 x Nblocky x 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. Boundary Conditions

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`.

<table>
<thead>
<tr>
<th>$ab_{\text{boundary}_type}$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>periodic</td>
<td>Periodic (‘wrap-around’)</td>
</tr>
<tr>
<td>reflect</td>
<td>Non-penetrating boundaries; zero-gradient with normal velocity reflected</td>
</tr>
<tr>
<td>outflow</td>
<td>Zero-gradient boundary conditions; allows shocks to leave the domain</td>
</tr>
<tr>
<td>diode</td>
<td>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</td>
</tr>
<tr>
<td>hydrostatic-f2</td>
<td>Hydrostatic boundary handling as in FLASH2. See remark in text.</td>
</tr>
<tr>
<td>hydrostatic-f2+nvrefl</td>
<td>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.</td>
</tr>
<tr>
<td>hydrostatic-f2+nvout</td>
<td></td>
</tr>
<tr>
<td>hydrostatic-f2+nvdiode</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>The user must implement the desired boundary behavior; see text.</td>
</tr>
</tbody>
</table>

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

`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 $ab_{\text{boundary}\_type}$ 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 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.
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.

<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>GRID_BC_MG_EXTRAPOLATE</td>
<td>for use by multigrid solver</td>
</tr>
<tr>
<td>—</td>
<td>NEUMANN</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>

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 where 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.applyBCEdgeAll1UnkVars (which calls Grid.applyBCEdge) each. Another implementation of Grid.applyBCEdgeAll1UnkVars 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.applyBCEdgeAll1UnkVars, 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 \texttt{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 \texttt{Grid} implementation chosen. At a lower level, the various implementations of \texttt{Grid\_Main} have different ways of requesting that boundary guard cells be filled. The \texttt{Grid\_BoundaryConditions} subunit collaborates with \texttt{Grid\_Main} 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 \texttt{Grid} unit. For \texttt{PARAMESH} 4, the most relevant routine is \texttt{amr\_1blk\_bcset.F90}, for \texttt{PARAMESH} 2 it is \texttt{tot\_bnd.F90}, and for uniform grid UG it is \texttt{gr\_bcApplyToAllBlks.F90}.

### 8.4.3.1 Additional Concerns with \texttt{PARAMESH} 4

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

1. In other \texttt{Grid} implementations, filling of domain boundary guard cells is under control of the “user” (in this context, the user of the grid implementation, \textit{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 \texttt{Grid\_fillGuardCells}, only. With \texttt{PARAMESH} 4, the user-provided \texttt{amr\_1blk\_bcset} routine can be called from within the depths of \texttt{PARAMESH} on individual blocks (and cell regions, see below) during guard cell filling and at other times when the user has called a \texttt{PARAMESH} routine. It is not easy to predict when and in which sequence this will happen.

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

3. \texttt{PARAMESH} 4 does not let the user routine \texttt{amr\_1blk\_bcset} operate on permanent storage (\texttt{unk} \textit{etc.}) directly, but on (regions of) one-block buffers.

4. \texttt{PARAMESH} 4 occasionally invokes \texttt{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 \texttt{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 \texttt{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 \texttt{unk} \textit{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 \texttt{PARAMESH} 4 calls while a call to \texttt{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 \texttt{blockID}. The code will be passed a “block handle” which can be used in some ways, but not all, like a valid \texttt{blockID}.

#### Caveat Block Handles

See the README file in \texttt{Grid/Grid\_BoundaryConditions} 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:
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). 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 \(i = 1 \ldots nxb\), \(j = 1 \ldots nyb\), and \(k = 1 \ldots nzb\) 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

```plaintext
./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:

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

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

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 \(nvar \times (2 \times nguard + nx) \times (2 \times nguard + ny) \times (2 \times nguard + nz)\) 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.

```plaintext
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.

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

In this case, each processor will now have blocks of size \(16 \times 32 \times 64\).
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 $n_{xb} \times n_{yb} \times n_{zb}$ 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 differencing 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 `MAXBLOCKS`, which can be overridden at setup time with the `setup ...--maxblocks=#` argument. The amount of memory consumed by the Grid data structure of code when running with `PARAMESH` is $NUNK_VARS \times (2 \times n_{guard} + n_{xb}) \times (2 \times n_{guard} + n_{yb}) \times (2 \times n_{guard} + n_{zb}) \times MAXBLOCKS$. `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 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.
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).

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 act 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 parts of this metadata on a block by block basis, like Grid.getBlkBoundaryBox, Grid.getBlkCenterCoords, Grid.getBlkPhysicalSize, Grid.getBlkRefineLevel, and Grid.getBlkType.

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” 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 \( 2^{\text{refine\_max}-n} \) where \( 1 \leq n \leq \text{refine\_max} \).

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-alpha 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-alpha supply their own default interpolation scheme, which is used when FLASH4-alpha is configured with the -gridinterpolation=native setup option (see Section 5.2). The default schemes are only appropriate for Cartesian coordinates. If FLASH4-alpha 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-monotonicity in the data. The order of “monotonic” interpolation can be chosen with the interp_order 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-alpha provides three ways to deal with this:

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 convertToConsvdInMeshInterp 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

VARIABLE dens TYPE: PER_VOLUME

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.
This behavior is available in both PARAMESH 2 and PARAMESH 4. It is enabled by setting the `convertToConsrvdForMeshCalls` 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-alpha, this is the default behavior whenever available. It can be enabled explicitly (only necessary in setups that change the default) by setting the `convertToConsrvdInMeshInterp` 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-alpha 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

\[ E_i = \frac{|u_{i+1} - 2u_i + u_{i-1}|}{|u_{i+1} - u_i| + |u_i - u_{i-1}| + \epsilon} \left(\frac{\partial^2 u}{\partial x^2} \Delta x \right)^2, \]  

(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

\[ E_{i_1i_2i_3} = \left\{ \frac{\sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right)^2}{\sum_{pq} \left[ \left( \frac{\partial u}{\partial x_p} \right)^2_{|i_p+1/2|} + \left( \frac{\partial u}{\partial x_p} \right)^2_{|i_p-1/2|} \Delta x_p + \epsilon \frac{\partial^2 u}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right]^2} \right\}^{1/2}, \]  

(8.2)

where the sums are carried out over coordinate directions, and where, unless otherwise noted, partial derivatives are evaluated at the center of the \( i_1i_2i_3 \)-th cell.
The estimator actually used in FLASH4-alpha’s default refinement criterion is a modification of the above, as follows:

$$E_i = \frac{|u_{i+2} - 2u_{i} + u_{i-2}|}{|u_{i+2} - u_{i}| + |u_{i} - u_{i-2}| + \epsilon\sum |u_{i+2}| + 2 |u_{i}| + |u_{i-2}|}$$  \hspace{1cm} (8.3)

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

$$E_{iX_{iY_{iZ}}} = \left\{ \frac{\sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \right)^2}{\sum_{pq} \left[ \frac{1}{2 \Delta x_q} \left( \frac{\partial u}{\partial x_p} \big|_{i_p+1} + \frac{\partial u}{\partial x_p} \big|_{i_p-1} \right) + \epsilon \left| u_{pq} \right| \Delta x_p \Delta x_q \right)^2} \right\}^{1/2} \hspace{1cm} (8.4)$$

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 $iX_{iY_{iZ}}$-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 `refine_filter.#` runtime parameters. Blocks are marked for refinement when the value of $E_{iX_{iY_{iZ}}}$ for any of the block’s cells exceeds a threshold given by the runtime parameters `refine_cutoff.#`, where the number # matching the number of the `refine_var.#` runtime parameter selecting the refinement variable. Similarly, blocks are marked for derefinement when the values of $E_{iX_{iY_{iZ}}}$ for all of the block’s cells lie below another threshold given by the runtime parameters `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 `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 `refine_on_particle_count` to true and setting `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 derefinition 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 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 \texttt{Grid\_markRefineSpecialized}. It is intended to be called from the \texttt{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 \texttt{specs} before making the call. A copy of the file \texttt{Grid\_markRefineDerefine.F90} should be placed in the \texttt{Simulation} directory, and the interface file \texttt{Grid\_interface.F90} should be used in the header of the function.

\textbf{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 \texttt{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 \texttt{lrefine\_max}. See the runtime parameters \texttt{gr\_lrefineMaxRedDoByLogR}, \texttt{gr\_lrefineMaxRedRadiusFact}, \texttt{gr\_lrefineMaxRedDoByTime}, \texttt{gr\_lrefineMaxRedTimeScale}, \texttt{gr\_lrefineMaxRedTRef}, and \texttt{gr\_lrefineMaxRedLogBase}.

\section*{8.7 Chombo}

We have included an experimental \texttt{Grid} implementation which makes use of Chombo library in FLASH4-alpha. 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 \texttt{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 \texttt{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 5.8. Please also see the restrictions in Hydro Section 13.1.5 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 fastest 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 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:
• \texttt{lrefine\_max} Used in an identical way to Paramesh. It is still a unit-based parameter and not zero-based. Defaults to 1.

• \texttt{igridsize, jgridsize, kgridsize} The global number of cells on the coarsest level of the adaptive mesh. Varying \texttt{igridsize} gives the same level of control as the unused Paramesh parameters \texttt{lrefine\_min} and \texttt{nblockx}. Defaults to 16.

• \texttt{maxBlockSize} The maximum number of cells along a single dimension of a block. Defaults to 16.

• \texttt{BRMeshRefineBlockFactor} The minimum number of cells along a single dimension of a block. See BRMeshRefine class in \\texttt{Chombo} user-guide. Defaults to 8.

• \texttt{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 \\texttt{Chombo} user-guide. Defaults to 0.75.

• \texttt{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 \\texttt{Chombo} user-guide. Defaults to 1.

• \texttt{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 \\texttt{Chombo} objects. Defaults to 0.

• \texttt{refRatio} The refinement jump between fine-coarse levels, typically 2, 4 or 8. Defaults to 2.

• \texttt{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.

• \texttt{tagRadius} The radius of cells around a tagged cell that should also be tagged. Defaults to 2.

Set \texttt{convertToConsvdForMeshCalls} runtime parameter to \texttt{.true.} to interpolate mass specific solution variables (see Section 8.6.2.1). We have not yet added code for \texttt{convertToConsvdInMeshInterp} runtime parameter and so it should not be used.

Setup a simulation using:

```
./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 \texttt{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 \texttt{Grid\_init} routine, like all other \texttt{Unit\_init} routines, collects the runtime parameters needed by the unit and stores values in the data module. If using UG, the \texttt{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 \texttt{Grid\_initDomain} routine, which also makes a call to mesh package’s own initialization routine. The physical variables are all owned by the \texttt{Grid} unit, and it initializes them by calling the \texttt{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 \texttt{Eos} function to bring all simulation variables to thermodynamic equilibrium. Even though the mesh-based variables are under \texttt{Grid}’s control, all the physics units can operate on and modify them.

A suite of \texttt{get/put} accessor/mutator functions allows the calling unit to fetch or send data by the block. One option is to get a pointer \texttt{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-alpha 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 \texttt{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 \texttt{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 \texttt{OPTIONAL} arguments, see \texttt{Grid\_fillGuardCells} for documentation. When these optional arguments are absent or when a \texttt{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 \texttt{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.

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

As mentioned in Chapter 4, FLASH allows every unit to identify scalar variables for checkpointing. In the \texttt{Grid} unit, the function that takes care of consolidating user specified checkpoint variable is \texttt{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.

\section*{8.9 GridParticles}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{grid_particles_diagram.png}
\caption{The \texttt{Grid} unit: structure of \texttt{GridParticles} subunit.}
\end{figure}
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 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 algorithms 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 particle 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 18. 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 18.2.

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:

```bash
-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.

![Figure 8.10: A single 2-D block showing how guard cells are divided into regions.](image)

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 onward 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

![Diagram]

Figure 8.11: The Grid unit: structure of GridSolvers subunit.

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 8.10.2.3 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:

$$N_1 \times N_2 \times N_3 / P,$$

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

\[ N_1 \times N_2 / P_1 \times N_3 / P_2 \]

\[ P = P_1 \times P_2. \]  

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 \textsc{mpi} operation \texttt{alltoall}. In a slabwise distribution, the upper limit on the number of processors is determined by the smallest of \(< N_1, N_2, N_3 \rangle\), where as in our distribution, the upper limit on the number of processors is the smallest of \(< N_1 \ast N_2, N_2 \ast N_3, N_1 \ast N_3 \rangle\).

8.10.1.1 Using \texttt{Pfft}

\texttt{Pfft} can only be used with a pencil grid, with the constraint that the number of processors along the \texttt{IAXIS} must be 1. This is because all one dimensional transforms are computed locally within a processor. However, \texttt{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, \texttt{Pfft} can be used in two different modes. In the first mode, every instance of \texttt{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 \texttt{pfft\_setupOnce} to true, which enables the \texttt{FLASH} initialization process to also create and initialize all the data structures of \texttt{Pfft}. The finalization of the \texttt{Pfft} subunit is also done automatically by the \texttt{FLASH} shutdown process in this mode. However, if a simulation needs to use \texttt{Pfft} in different configurations at different instances of its use, then each set of calls to \texttt{Grid\_pfft} for computing the transforms must be preceded by a call to \texttt{Grid\_pfftInit} and followed by a call to \texttt{Grid\_pfftFinalize}. In addition, the runtime parameter \texttt{pfft\_setupOnce} should be set to false. A few other helper routines are available in the subunit to allow the user to query \texttt{Pfft} about the dimensioning of the domain, and also to map the Mesh variables from the \texttt{unk} array to and from \texttt{Pfft} compatible (single dimensional) arrays. \texttt{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 \texttt{Pfft} data movement subroutines

Mesh reconfiguration subroutines are available to generate a pencil grid for the \texttt{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 \texttt{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 \texttt{Grid\_pfftInit}. In UG mode they are invoked when the number of processors in the \texttt{IAXIS} of the \texttt{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 \textsc{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 \texttt{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 \texttt{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 \texttt{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 this 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

\[ F = \cos 2x \times \sin 3y \]  

(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 \(x_{cm}\)

\[
x_{cm} = \frac{\int d^3x x \rho(x)}{\int d^3x \rho(x)} .
\]  

(8.10)

We will take \(x_{cm}\) as our origin. In integral form, Poisson’s ((8.9)) is

\[
\phi(x) = -\frac{\alpha}{4\pi} \int d^3x' \frac{\rho(x')}{|x - x'|} .
\]

(8.11)

The Green’s function for this equation satisfies the relationship

\[
\frac{1}{|x - x'|} = 4\pi \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell + 1} \frac{r_<}{r_<+1} Y_{\ell m}(\theta', \phi') Y_{\ell m}(\theta, \varphi) ,
\]

(8.12)

where the components of \(x\) and \(x'\) are expressed in spherical coordinates \((r, \theta, \phi)\) about \(x_{cm}\), and

\[
r_< = \min\{|x|, |x'|\}
\]

(8.13)

\[
r_> = \max\{|x|, |x'|\} .
\]

Here \(Y_{\ell m}(\theta, \varphi)\) are the spherical harmonic functions

\[
Y_{\ell m}(\theta, \varphi) \equiv (-1)^m \sqrt{\frac{2\ell + 1}{4\pi} \frac{(\ell - m)!}{(\ell + m)!}} P_{\ell m}(\cos \theta)e^{im\varphi} .
\]

(8.14)

\(P_{\ell m}(x)\) are Legendre polynomials. Substituting (8.12) into (8.11), we obtain

\[
\phi(x) = -\alpha \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell + 1} \left\{ Y_{\ell m}(\theta, \varphi) \times \left[ \frac{r_<}{r_<+1} \int_{r_<}^{r_>} d^3x' \rho(x') Y_{\ell m}^*(\theta', \phi') + \frac{1}{r_<+1} \int_{r_>}^{r_<} d^3x' \rho(x') Y_{\ell m}^*(\theta', \phi') r'<r< \right] \right\} .
\]

(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

\[
\sum_{m=-\ell}^{\ell} Y_{\ell m}^*(\theta', \phi') Y_{\ell m}(\theta, \varphi) = \frac{2\ell + 1}{4\pi} \left[ P_{00}(\cos \theta)P_{00}(\cos \theta') + 2 \sum_{m=1}^{\ell} \frac{(\ell - m)!}{(\ell + m)!} \left( P_{\ell m}(\cos \theta)P_{\ell m}(\cos \theta') \cos (m(\varphi - \varphi')) \right) \right] .
\]

(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

\[
\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \sum_{\ell=0}^{\infty} P_{\ell 0}(\cos \theta) \left[ r^\ell \mu_{\ell 0}^e(r) + \frac{1}{r^{\ell+1}} \mu_{\ell 0}^o(r) \right] - \frac{\alpha}{2\pi} \sum_{\ell=1}^{\infty} \sum_{m=1}^{\ell} P_{\ell m}(\cos \theta) \left[ (r^\ell \cos m\phi) \mu_{\ell m}^e(r) + (r^\ell \sin m\phi) \mu_{\ell m}^o(r) + \frac{\cos m\phi}{r^{\ell+1}} \mu_{\ell m}^o(r) + \frac{\sin m\phi}{r^{\ell+1}} \mu_{\ell m}^o(r) \right].
\]

(8.17)

The even (e)/odd (o), inner (i)/outer (o) source moments in this expression are defined to be

\[
\mu_{\ell m}^e(r) = \frac{(\ell - m)!}{(\ell + m)!} \int_{r>r'} d^3\mathbf{x}' r^\ell \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \cos m\phi', \quad \mu_{\ell m}^o(r) = \frac{(\ell - m)!}{(\ell + m)!} \int_{r>r'} d^3\mathbf{x}' r^\ell \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \sin m\phi'.
\]

(8.18, 8.19)

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 \( \mathbf{x}_{\ell m} \), with an implied discretization. The solver allocates storage for moment samples spaced a distance \( \Delta \) apart in radius

\[
\mu_{\ell m, q}^e = \mu_{\ell m}^e(q\Delta), \quad \mu_{\ell m, q}^o = \mu_{\ell m}^o((q-1)\Delta),
\]

(8.22)

\[
\bar{\mu}_{\ell m, q}^e = \bar{\mu}_{\ell m}^e(q\Delta), \quad \bar{\mu}_{\ell m, q}^o = \bar{\mu}_{\ell m}^o((q-1)\Delta).
\]

(8.23)

The sample index \( q \) varies from 0 to \( N_q \) (\( \mu_{\ell m, 0}^e \) and \( \mu_{\ell m, 0}^o \) 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 establish (see mpole\_subSample) consisting of \( N' \) points at the locations \( \mathbf{x}_{i'j'k'} \), where

\[
x_{i'} = x_i + (i' - 0.5(N' - 1)) \frac{\Delta x_i}{N'}, \quad i' = 0 \ldots N' - 1
\]

(8.24)

\[
y_{j'} = y_j + (j' - 0.5(N' - 1)) \frac{\Delta y_j}{N'}, \quad j' = 0 \ldots N' - 1
\]

(8.25)

\[
z_{k'} = z_k + (k' - 0.5(N' - 1)) \frac{\Delta z_k}{N'}, \quad k' = 0 \ldots N' - 1
\]

(8.26)

and where \( \mathbf{x}_{ijk} \) is the center of cell \( ijk \). (For clarity, we have omitted \( ijk \) indices on \( \mathbf{x}' \) as well as all block...
indices.) For each subcell, we assume $\rho(x'_{ij'k'}) \approx \rho_{ijk}$ and then apply

$$
\mu^e_{\ell m,q \ge q'} \leftarrow \mu^e_{\ell m,q \ge q'} + \frac{(-m)!}{(\ell + m)!} \Delta x_i \Delta y_j \Delta z_k \rho(x'_{ij'k'}) P_{\ell m}(\cos \theta'_{ij'k'}) \cos m\varphi'_{ij'k'} \tag{8.27}
$$

$$
\mu^o_{\ell m,q \ge q'} \leftarrow \mu^o_{\ell m,q \ge q'} + \frac{(-m)!}{(\ell + m)!} \Delta x_i \Delta y_j \Delta z_k r'_{ij'k'} \rho(x'_{ij'k'}) P_{\ell m}(\cos \theta'_{ij'k'}) \sin m\varphi'_{ij'k'} \tag{8.28}
$$

$$
\mu^{oo}_{\ell m,q \le q'} \leftarrow \mu^{oo}_{\ell m,q \le q'} + \frac{(-m)!}{(\ell + m)!} \Delta x_i \Delta y_j \Delta z_k \rho(x'_{ij'k'}) P_{\ell m}(\cos \theta'_{ij'k'}) \cos m\varphi'_{ij'k'} \tag{8.29}
$$

$$
\mu^{oo}_{\ell m,q \le q'} \leftarrow \mu^{oo}_{\ell m,q \le q'} + \frac{(-m)!}{(\ell + m)!} \Delta x_i \Delta y_j \Delta z_k r'_{ij'k'} \rho(x'_{ij'k'}) P_{\ell m}(\cos \theta'_{ij'k'}) \sin m\varphi'_{ij'k'} \tag{8.30}
$$

where

$$
q' = \left\lfloor \frac{|x'_{ij'k'}|}{\Delta} \right\rfloor + 1 \tag{8.31}
$$

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 ($\mu^{oi}_{\ell m,N_q}$) 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 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

$$
x'_i = x_i + (i' - 0.5N') \frac{\Delta x_i}{N'} \quad , \quad i' = 0 \ldots N' \tag{8.32}
$$

$$
y'_j = y_j + (j' - 0.5N') \frac{\Delta y_j}{N'} \quad , \quad j' = 0 \ldots N' \tag{8.33}
$$

$$
z'_k = z_k + (k' - 0.5N') \frac{\Delta z_k}{N'} \quad , \quad k' = 0 \ldots N' \tag{8.34}
$$

The cell-averaged potential in cell $ijk$ is then

$$
\phi_{ijk} = \frac{1}{N'^3} \sum_{i'j'k'} \phi(x'_{i'j'k'}) \tag{8.35}
$$

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 new implementation of multipole Poisson solver is located in the directory

source/Grid/GridSolvers/Multipole_experimental.

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.

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 and 3D Cartesian. Symmetries can be specified for the 2D spherical and cylindrical cases (a horizontal symmetry plane along the radial axis) and the 3D Cartesian case (imposed axisymmetric property). Because of the imposed symmetries 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 mass $x_{cm}$ is determined via

$$x_{cm} = \frac{\int x \rho(x) \, dx}{\int \rho(x) \, dx}.$$  (8.36)

We will take $r_{cm}$ as our origin. In integral form, Poisson’s equation (8.9) becomes

$$\phi(x) = -\frac{\alpha}{4\pi} \int \frac{\rho(x')}{|x - x'|} \, dx'.$$  (8.37)

The inverse radial distance part can be expanded in terms of Legendre polynomials

$$\frac{1}{|x - x'|} = \sum_{\ell=0}^{\infty} \frac{x_<^\ell}{x_>^{\ell+1}} P_\ell(\cos \gamma),$$  (8.38)

where $x_<$ ($x_>$) indicate the smaller (larger) of the magnitudes and $\gamma$ denotes the angle between $x$ and $x'$.

Expansion of the Legendre polynomials in terms of spherical harmonics gives

$$P_\ell(\cos \gamma) = \frac{4\pi}{2\ell + 1} \sum_{m=-\ell}^{+\ell} Y^*_{\ell m}(\theta', \phi') Y_{\ell m}(\theta, \phi),$$  (8.39)

where $\theta, \phi$ and $\theta', \phi'$ are the spherical angular components of $x$ and $x'$ about $x_{cm}$. Defining now the regular $R_{\ell m}$ and irregular $I_{\ell m}$ solid harmonic functions

$$R_{\ell m}(x_<) = \sqrt{\frac{4\pi}{2\ell + 1}} x_<^\ell Y_{\ell m}(\theta, \phi),$$  (8.40)

$$I_{\ell m}(x_> ) = \sqrt{\frac{4\pi}{2\ell + 1}} \frac{Y_{\ell m}(\theta, \phi)}{x_>^{\ell+1}},$$  (8.41)

we can rewrite Eq.(8.37) in the form

$$\phi(x) = -\frac{\alpha}{4\pi} \sum_{\ell m} R_{\ell m}(x_<) I^*_ {\ell m}(x_> ) \rho(x') \, dx',$$  (8.42)

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

$$\phi(q) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}(q_<) I^*_ {\ell m}(q_> ) m(q'),$$  (8.43)

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 center of mass 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 is accounted for
by considering the cell’s self-potential $\phi_{Self}(q)$ and the correct form becomes, after splitting the summation over cells in two parts

$$
\phi(q) = -\frac{\alpha}{4\pi} \left\{ \sum_{q' < q} \sum_{\ell m} [R_{\ell m}(q')^m(q')] I_{\ell m}(q) + \sum_{q' > q} \sum_{\ell m} R_{\ell m}(q') [I_{\ell m}(q')^m(q')] + \phi_{Self}(q) \right\}.
$$

(8.44)

Defining the two moments

$$
M_{\ell m}^R(q) = \sum_{q' < q} R_{\ell m}(q')^m(q')
$$

(8.45)

$$
M_{\ell m}^I(q) = \sum_{q' > q} I_{\ell m}(q')^m(q'),
$$

(8.46)

we obtain

$$
\phi(q) = -\frac{\alpha}{4\pi} \left[ \sum_{\ell m} M_{\ell m}^R(q) I_{\ell m}(q) + \sum_{\ell m} M_{\ell m}^I(q) R_{\ell m}(q) + \phi_{Self}(q) \right]
$$

(8.47)

and using vector notation

$$
\phi(q) = -\frac{\alpha}{4\pi} \left[ M^R(q) \cdot \mathbf{I}(q) + M^I(q) \cdot \mathbf{R}(q) + \phi_{Self}(q) \right].
$$

(8.48)

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

$$
R_{\ell m} = R_{\ell m}^c + i R_{\ell m}^s.
$$

(8.49)

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 (8.49) into (8.48) all cosine and sine mixed terms of the scalar products cancel out. Also, due to the symmetry relations

$$
R_{\ell,-m}^c = (-1)^m R_{\ell m}^c \quad \text{and} \quad R_{\ell,-m}^s = (-1)^m R_{\ell m}^s,
$$

(8.50)

(8.51)

we can restrict ourselves to the following polar angle number ranges

$$
c : \ell \geq 0, \quad \ell \geq m \geq 0 \quad \text{and} \quad s : \ell \geq 1, \quad \ell \geq m \geq 1.
$$

(8.52)

(8.53)

The real formulation of (8.48) becomes then

$$
\phi(q) = -\frac{\alpha}{4\pi} \left[ M^R \cdot \Delta \mathbf{I} + M^I \cdot \Delta \mathbf{R} + M^R \cdot \Delta \mathbf{R} + M^I \cdot \Delta \mathbf{I} + \phi_{Self} \right],
$$

(8.54)

where it is understood that all rhs terms are those corresponding to the $q$-th cell. The $\Delta$ matrix is a diagonal matrix whose elements are equal to 2 for $m \neq 0$ and 1 otherwise

$$
\Delta = \text{diag}(2 - \delta_{m0}).
$$

(8.55)

The recursion relations for calculating the solid harmonic vectors are

$$
R_{00}^c = 1
$$

(8.56)

$$
R_{\ell\ell}^c = -\frac{x R_{\ell-1,\ell-1}^c - y R_{\ell-1,\ell-1}^s}{2\ell}
$$

(8.57)

$$
R_{\ell\ell}^s = -\frac{y R_{\ell-1,\ell-1}^c + x R_{\ell-1,\ell-1}^s}{2\ell}
$$

(8.58)

$$
R_{\ell m}^{c/s} = \frac{(2\ell - 1)z R_{\ell-1,m}^{c/s} - r^2 R_{\ell-2,m}^{c/s}}{(\ell + m)(\ell - m)}, \quad 0 \leq m < \ell
$$

(8.59)
\[ I^c_{00} = \frac{1}{r} \]  
\[ I^c_{\ell\ell} = -(2\ell - 1) \frac{x I^c_{\ell-1,\ell-1} - y I^c_{\ell-1,\ell-1}}{r^2} \]  
\[ I^s_{\ell\ell} = -(2\ell - 1) \frac{y I^s_{\ell-1,\ell-1} + x I^s_{\ell-1,\ell-1}}{r^2} \]  
\[ I^{c/s}_{\ell m} = \frac{(2\ell - 1) z I^{c/s}_{\ell-1,m} - [(\ell - 1)^2 - m^2] I^{c/s}_{\ell-2,m}}{r^2}, \quad 0 \leq m < \ell \]  

in which \( x, y, z \) are the cartesian location coordinates of the \( q \)-th cell 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 \( z \to -z \), resulting in the symmetry relations \( R^{c/s}_{\ell0} \to -R^{c/s}_{\ell0} \) for even \( \ell \) and \( R^{c/s}_{\ell0} \to -R^{c/s}_{\ell0} \) 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 (generators of the symmetry group 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 (8.56-8.63), 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 \( D_x, D_y, D_z \) for the 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:

\[ R^{c/s}_{L\ell m} \approx 1 \]  
\[ I^{c/s}_{L\ell m} \approx 1. \]  

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.(8.54). 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 with \( x, y = 0 \) and \( z \neq 0 \). Then \( r^2 = z^2, |z| = r \) and only the \( m = 0 \) components are different from zero. An explicit form can be stated for the absolute values of these components in terms of \( r \):

\[ |R_{\ell0}| = \frac{r^\ell}{\ell!} \]  
\[ |I_{\ell0}| = \frac{\ell!}{r^{\ell+1}}. \]
Since $r = \sqrt{x^2 + y^2 + z^2}$, damping of the coordinates with $D$ results in a damped radial cell distance $Dr$. Inserting this result into (8.66) and (8.67) and imposing conditions (8.64) and (8.65) results in

$$D_R = \frac{1}{r} \sqrt{L!} \approx \frac{1}{r} \left( \frac{L}{e} \right)^{\frac{L}{2} \pi L} \quad (8.68)$$

$$D_I = \frac{1}{r} L^\frac{1}{2} \sqrt{L!} \approx \frac{1}{r} \left( \frac{L}{e} \right)^{\frac{1}{2} \pi e^2} \frac{L}{L}. \quad (8.69)$$

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.

Ideally, the moments in Eq. (8.54) should be calculated and stored for each possible cell $q$. For high refinement levels and/or 3D simulations this would result in an unmanageable request for computer memory. Several $q$’s have to be bundled into radial bins $Q$ defined by lower and upper radial bounds. Once a solid harmonic vector pair $\mathbf{R}(q)$ and $\mathbf{I}(q)$ for a particular cell has been calculated, its radial bin location $q \rightarrow Q$ is determined and its contribution is added to the moments $\mathbf{M}^R(Q)$ and $\mathbf{M}^I(Q)$. The computational definition of the radial bin moments is

$$\mathbf{M}^R(Q) = \sum_{q' \leq Q} \mathbf{R}(q') m(q') \quad (8.70)$$

$$\mathbf{M}^I(Q) = \sum_{q' \geq Q} \mathbf{I}(q') m(q') \quad (8.71)$$

where $q' \leq Q$ means including all cells belonging to $Q$ and all radial bins with lower radial boundaries than $Q$. Note the difference when compared to the definitions in (8.45) and (8.46) due to the equality in the summation index. 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. (8.45) 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 the cell moments $\mathbf{M}^R(q)$ and $\mathbf{M}^I(q)$, which are ultimately needed for the potential evaluation? Consider the three consecutive radial bins $Q - 1$, $Q$ and $Q + 1$, together with their calculated moments:

$$\begin{bmatrix}
\mathbf{M}^R(Q - 1) \\
\mathbf{M}^I(Q - 1)
\end{bmatrix}
\begin{bmatrix}
\mathbf{M}^R(Q) \\
\mathbf{M}^I(Q)
\end{bmatrix}
\begin{bmatrix}
\mathbf{M}^R(Q + 1) \\
\mathbf{M}^I(Q + 1)
\end{bmatrix} \quad (8.72)$$

Let us concentrate on the $Q$ bin, whose lower and upper radial limits are shown as solid vertical lines. The radial distance corresponding to cell $q$ splits the $Q$ bin into two parts: the left fractional part, denoted $R_{frac}$, and the right fractional part, denoted $I_{frac}$. Since both $\mathbf{M}^R(Q - 1)$ and $\mathbf{M}^I(Q + 1)$ are completely contained respectively in $\mathbf{M}^R(Q)$ and $\mathbf{M}^I(Q)$, the cell moments can be evaluated as:

$$\mathbf{M}^R(q) = \mathbf{M}^R(Q - 1) + R_{frac} [\mathbf{M}^R(Q) - \mathbf{M}^R(Q - 1) - m \mathbf{R}(q)] \quad (8.73)$$

$$\mathbf{M}^I(q) = \mathbf{M}^I(Q + 1) + I_{frac} [\mathbf{M}^I(Q) - \mathbf{M}^I(Q + 1) - m \mathbf{I}(q)], \quad (8.74)$$

where we have to remove the moment contributions coming from the cell itself. Ignoring the removal of the cell’s moment contribution leads to the error

$$\epsilon = -m \left[ R_{frac} \mathbf{R}(q) \cdot \mathbf{I}^*(q) + I_{frac} \mathbf{I}^*(q) \cdot \mathbf{R}(q) \right]$$

$$= -m \left[ \frac{L + 1}{r_q} \right], \quad (8.75)$$

where $m$ is the cell’s mass, $L$ the highest angular number and $r_q$ the radial distance of the cell. This error is of opposite sign to the cell’s self potential contribution and thus some cancellation occurs. When compared to the old multipole solver, where both contributions are ignored, we see that at large $r_q$ ($\epsilon \rightarrow 0$) and high refinement ($\phi_{Self} \rightarrow 0$) everything is fine, but for those cells close to the origin ($r_q \rightarrow 0$) the overall error can be substantial, particularly for high $L$ values.
The extraction of the cell’s individual moments via (8.73) and (8.74) is of course an approximation that relies on the statistically dense distribution of the individual cell moments inside each radial bin. For bins which are reasonably far away from the computational origin this statistical approximation is valid but for those close to the origin the statistical distribution does not hold and calculating the cell’s moments via the above scheme results in a so-called statistical error. The way out of this problem is to move from a statistical radial bin description to a more discrete one. By this we mean the construction of very narrow isolated radial bins near the multipolar origin. The code is thus forced to analyze the detailed structure of the geometrical domain grid surrounding the multipolar origin 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:

\[
\begin{align*}
\text{exponential bin size upper radial limit} & = s \cdot \Delta r \cdot Q^t, \quad (8.76) \\
\text{logarithmic bin size upper radial limit} & = s \cdot \Delta r \cdot \frac{e^{tQ} - 1}{e^t - 1}, \quad (8.77)
\end{align*}
\]

In these definitions, \( \Delta r \) is a small distance ‘atomic’ (basic unit) radial measure, defined as half the geometric mean of the cell dimensions \( \Delta x, \Delta y, \Delta z \) at highest refinement level:

<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 r \Delta y \Delta z} )</td>
</tr>
<tr>
<td>2D cylindrical</td>
<td>( \frac{1}{2} \sqrt{\Delta x \Delta y} )</td>
</tr>
<tr>
<td>2D spherical</td>
<td>( \frac{1}{2} \Delta x )</td>
</tr>
<tr>
<td>1D spherical</td>
<td>( \frac{1}{2} \Delta x )</td>
</tr>
</tbody>
</table>

\( s \) is a scalar factor to optionally increase the basic unit radial measure and \( Q = 1, 2, \ldots \) is a local bin index counter for each outer zone. Note that definitions (8.76) and (8.77) 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:

\[
\begin{align*}
\text{Q-th exponential bin size} & = s \cdot \Delta r \cdot [Q^t - (Q-1)^t] \quad (8.79) \\
\text{Q-th logarithmic bin size} & = s \cdot \Delta r \cdot e^{t(Q-1)}. \quad (8.80)
\end{align*}
\]

In principle the user can specify as many outer zone types as he/she likes, each having its own exponential or logarithmic parameter pair \( \{s,t\} \).

We next state the evaluation of the cell’s self potential \( \phi_{\text{Self}} \) in FLASH. For a rectangular cell of dimensions \( x, y, z \) an analytical formula can be given for the self potential per unit mass at its midpoint \( (x/2, y/2, z/2) \) in terms of inverse tangent and hyperbolic tangent functions, simplifying considerably for a cubic cell \( (T \) being a shorthand notation for \( 1/\sqrt{x^2 + y^2 + z^2} \)):

\[
\begin{align*}
\phi_{\text{Self}}(xyz)/m & = 2xy \text{artanh}(Tz) + 2xz \text{artanh}(Ty) + 2yz \text{artanh}(Tx) \\
& - x^2 \text{artanh}(yzT/x) - y^2 \text{artanh}(xzT/y) - z^2 \text{artanh}(xyT/z) \quad (8.81)
\end{align*}
\]

\[
\begin{align*}
\phi_{\text{Self}}(\text{cube})/m & = 3x^2 \left[ 2 \text{artanh}(1/\sqrt{3}) - \text{artanh}(1/\sqrt{3}) \right] \\
& \approx 2.3800772 \times x^2. \quad (8.82)
\end{align*}
\]

Here \( T = 1/\sqrt{x^2 + y^2 + z^2} \). For curvilinear shaped cells occurring in cylindrical and spherical geometries no such closed analytical form exists. The self potentials can be stated as very complicated expressions involving elliptic integrals of all three kinds and are prohibitively expensive to be calculated on the fly inside the multipole solver. The way out of this is to rectify the curvilinear shapes into rectangular form and apply the above given formula. For toroidal cell shapes with rectangular cross sections (2D cylindrical domain) the torus is simply cut open at the opposite end from the midpoint location, leading to an elongated rectangular cell with dimensions \( x, y, 2R\pi \) with \( R \) being the radial distance of the cell from the 2D cylindrical domain.
origin. For toroidal cells with curvilinear cross sections (2D spherical domain) the same rectifying procedure is applied on the toroidal part, while the curvilinear cross section is rectified by forming equal rectangular opposite sides of total length equalling the sum of the corresponding curvilinear sides. Several tests were performed to assess the validity of the rectification process. For the 2D cylindrical geometries a comparison between the exact self potentials (obtained through numerical integration) and the rectified ones showed a maximum error of $\approx 1\%$ occurring for completely closed tori. For the 2D spherical geometries only the curvilinear cross section rectification was tested leading to an almost identical match in two-dimensional potentials. From these tests we concluded that the rectification process is a viable option for calculating the cell self potentials in curvilinear cases.

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, 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 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 (Figure 8.12). This is overcome by the exchange of guardcells from coarse to fine using monotonic interpolation (Section 8.6.2) and external boundary extrapolation for the calculation of the residual.

![Figure 8.12: Contrast between jumps of refinement in meshes used in the original paper (left) and the oct-tree adapted method (right).](image)

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:

\[
R(x) = 4\pi G\rho(x) - \nabla^2 \tilde{\phi}(x) . \tag{8.84}
\]

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 \( \Delta z_\ell \).

\[
\mathcal{D}_\ell \tilde{\phi}^{bl}_{ijk} = \frac{1}{\Delta x_\ell^2} \left( \tilde{\phi}_{i+1,j,k}^{bl} - 2\tilde{\phi}_{ijk}^{bl} + \tilde{\phi}_{i-1,j,k}^{bl} \right) + \frac{1}{\Delta y_\ell^2} \left( \tilde{\phi}_{i,j+1,k}^{bl} - 2\tilde{\phi}_{ijk}^{bl} + \tilde{\phi}_{i,j-1,k}^{bl} \right) + \frac{1}{\Delta z_\ell^2} \left( \tilde{\phi}_{i,j,k+1}^{bl} - 2\tilde{\phi}_{ijk}^{bl} + \tilde{\phi}_{i,j,k-1}^{bl} \right). \tag{8.85}
\]

The restriction operator \( \mathcal{R}_\ell \) for block interior zones \( (i,j,k) \) is:

\[
(\mathcal{R}_\ell \tilde{\phi}^{P(c),\ell}_{ijk}) = \frac{1}{2^d} \sum_{i'j'k'} \tilde{\phi}^{c,\ell+1}_{i'j'k'}, \tag{8.87}
\]

where the indices \( (i',j',k') \) refer to the zones in block \( c \) that lie within zone \( (i,j,k) \) of block \( P(c) \). We apply the restriction operator throughout the interiors of blocks, but its opposite, the prolongation operator \( \mathcal{I}_\ell \), 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:

\[
(\mathcal{I}_\ell \tilde{\phi}^{c,\ell+1}_{i'j'k'}) = \sum_{p,q,r=-2}^{2} \alpha_{i'j'k'pqr} \tilde{\phi}^{P(c),\ell}_{i+j+p,j+q,k+r}. \tag{8.88}
\]

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,

\[
\alpha_{1/2,j'k'pqr} = \beta_p \gamma_{j'q} \gamma_{k'r} \tag{8.89}
\]

\[
(\beta_p) = \begin{pmatrix}
\frac{1}{12} & 7 & \frac{7}{12} & -\frac{1}{12} & 0
\end{pmatrix}
\]

\[
(\gamma_{j'q}) = \begin{cases}
\begin{pmatrix}
-3 \\
\frac{3}{128} \\
\frac{11}{64} \\
-\frac{11}{128} \\
\frac{3}{64}
\end{pmatrix} & j' \text{ odd} \\
\begin{pmatrix}
3 \\
\frac{3}{128} \\
\frac{11}{64} \\
-\frac{11}{128} \\
\frac{3}{64}
\end{pmatrix} & j' \text{ even}
\end{cases}
\]

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.

### 8.10.2.4 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:

\[
G_{ijk}^{\ell} = -16\pi G \left[ \frac{1}{\Delta x_\ell^2} \sin^2 \left( \frac{i\pi}{2n_x} \right) + \frac{1}{\Delta y_\ell^2} \sin^2 \left( \frac{j\pi}{2n_y} \right) + \frac{1}{\Delta z_\ell^2} \sin^2 \left( \frac{k\pi}{2n_z} \right) \right]^{-1} . \tag{8.90}
\]
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:

\[
\rho_{1jk} \to \rho_{1jk} - \frac{2}{\Delta x^2} \phi(x_{1/2}, y_j, z_k). \tag{8.91}
\]

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:

\[
G_{i,j,k}^\ell = \begin{cases} 
-16\pi G \left[ \frac{1}{\Delta x^2} \sin^2 \left( \frac{(i-1)\pi}{n_x} \right) + \frac{1}{\Delta y^2} \sin^2 \left( \frac{(j-1)\pi}{n_y} \right) + \frac{1}{\Delta z^2} \sin^2 \left( \frac{(k-1)\pi}{n_z} \right) \right]^{-1} \\
0 
\end{cases} \quad i, j, k \neq 1
\tag{8.92}
\]

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 \(D_\ell \tilde{\phi}_{ijk}^\ell = 4\pi G \rho_{ijk}^{\text{bf}}\) for all blocks \(b\) on level \(\ell\).
   - (c) Compute the residual \(R_{ijk}^\ell = 4\pi G \rho_{ijk}^{\text{bf}} - D_\ell \tilde{\phi}_{ijk}^\ell\)
   - (d) For each block \(b\) on level \(\ell\) that has children, prolong face values for \(\tilde{\phi}_{ijk}^\ell\) onto each child block.

3. **Residual propagation step:** Restrict the residual \(R_{ijk}^\ell\) 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 \(D_\ell C_{ijk}^{\text{bf}} = R_{ijk}^{\text{bf}}\) for all blocks \(b\) on level \(\ell\).
   - (c) Overwrite \(R_{ijk}^{\text{bf}}\) with the new residual \(R_{ijk}^{\text{bf}} - D_\ell C_{ijk}^{\text{bf}}\) for all blocks \(b\) on level \(\ell\).
   - (d) Correct the solution on all leaf-node blocks \(b\) on level \(\ell\): \(\tilde{\phi}_{ijk}^{\text{bf}} \to \tilde{\phi}_{ijk}^{\text{bf}} + C_{ijk}^{\text{bf}}\).
   - (e) For each block \(b\) on level \(\ell\) that has children, interpolate face boundary values of \(C_{ijk}^{\text{bf}}\) 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 \(4n_xn_yn_z\) values per block. Global communication is required in computing the tolerance-based stopping criterion.
8.10.2.5 Interfacing PFFT with Multigrid

We demonstrate a technique to improve the performance of the Multigrid solver 8.10.2.3. This involves replacing single block FFTs with a parallel FFT at a specified coarse level. The coarse level may be any level that is fully refined (i.e. containing blocks that completely cover the computational domain). Currently, we automatically select the maximum level that is fully refined.

The motive for interfacing the two units is to improve the poor load balance of the Multigrid unit (see Section 8.10.2.3). Poor load balance occurs because we use single block FFTs (by whichever processor owns the block) to obtain a solution at each level. At the coarse levels there are relatively few blocks compared to available processors. This means many processors are effectively idle during the coarse level solves. Our intention is to distribute work more evenly by performing a parallel FFT over all processors.

The hybrid solver can be used in place of the Multigrid solver for all-periodic, [2 periodic, 1 neumann], or [1 periodic, 2 neumann] boundary condition problems only. It is included in the application when the following sub-unit is selected: Grid/GridSolvers/Multigrid/PfftTopLevelSolve. Please note that this sub-unit automatically includes the necessary Pfft routines.

The hybrid solver reaches solution faster than the pure Multigrid solver during our 3-periodic Pfft (3D) unit test. We measured the time taken to reach a Poisson solution on 128 and 256 processors (lrefine_min=4, lrefine_max=5 configuration), and 256 and 512 processors (lrefine_min=4, lrefine_max=6 configuration) using the Collective mesh reconfiguration. For the lrefine_max=5 configuration we reached a solution in 43%(128 processors) less time and 35%(256 processors) less time than a pure Multigrid solution. For the lrefine_max=6 configuration we reached a solution in 22%(256 processors) less time and 15%(512 processors) less time than a pure Multigrid solution. Performance profiles indicate that the actual FFTs and parallel transposes are very fast, and that most of the Pfft time is actually spent moving data from a FLASH grid to a pencil grid, and then back again (The PtToPt mesh reconfiguration implementation has not been profiled yet). However, the time spent in the Pfft unit is relatively small (about a fifth of total Poisson solve time during the 512 processor run) compared to the Multigrid unit during a hybrid solver run.

In these tests we performed the parallel FFT at level 4 in the AMR grid, and this increased the overall heap memory usage by just over 4MB in each of the configurations (128,256,512 processors). This is acceptable, but at finer solve levels the memory required increases rapidly. This is mostly due to the large communication buffers, which allow us to use collective MPI calls to simplify the communication. It may be necessary to change the solve level to a more coarse level by editing Grid/GridSolvers/Multigrid/-PfftTopLevelSolve/gr_hgSolve.F90 if you are running in a limited memory environment.

Finally, we note that a large portion of the total time is actually spent in the routine gr_hgGuardCell. This routine is invoked by Multigrid routines when data must be communicated to neighboring blocks (and thus off-processor in some situations). Therefore, even though we attempt to overcome the load balance issue by interfacing with Pfft, there is still an issue with the time spent exchanging guard cells. As such, the hybrid solver performance is still limited by the communication cost associated with guard cell exchanges within Multigrid.

Please note that a lot of this work is still at an experimental stage. We still need to perform further validation checks, and also to profile the PtToPt mesh reconfiguration implementation.

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

\[
\text{call Grid\_solvePoisson (iSoln, iSrc, bcTypes(6), bcValues(2,6), poisfact)},
\]
8.10. GRIDSOLVERS

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 \(iSoln\) 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 \texttt{Grid\_solvePoisson}. It should be noted that if \texttt{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 you use the quadratic\_cartesian/cylindrical/spherical (quadratic) interpolants supplied by FLASH. If the interpolants supplied by the mesh are not of at least the same order as the differencing scheme used, unphysical forces will be produced at refinement boundaries. Also, using constant or linear interpolants may cause the multigrid solver to fail to converge.

8.10.3.1 Multipole (original version)

The \texttt{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 \texttt{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 \texttt{+newMpole} at setup command line, effectively replacing the following setup options:

- \texttt{with-unit=Grid/GridSolvers/Multipole\_experimental}
- \texttt{with-unit=physics/Gravity/GravityMain/Poisson/Multipole}
- \texttt{without-unit=Grid/GridSolvers/Multipole}

The improved multipole solver takes several runtime parameters, whose functions are explained in detail below, together with comments about expected time and memory scaling.

- \texttt{mpole\_Lmax}: The maximum angular moment \(L\) to be used for the multipole Poisson solver. Depending on the domain geometry, the memory and time scalings due to this variable alone are: i) 3D cartesian \(\rightarrow (L+1)(L+1)\), 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\_2D\_symmetry\_plane**: 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 reduces the computational domain size by one half. The code internally computes the multipole moments as if the other symmetric part is present.

- **mpole\_3D\_axisymmetry**: 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 $(L + 1)$.

- **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 '<basename>_dumpMoments.txt', where <basename> 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 '<basename>_printRadialInfo.txt'.

- **mpole\_inner\_zone\_size**: The size defining the discrete inner zone. The size is given in terms of the inner zone smallest radius, which is determined at each time step by analyzing the domain grid structure around the multipolar origin (center of mass). 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 center of mass) this might be necessary. This value needs to be an integer, as it is used by the code to define dimensions of certain arrays.

- **mpole\_inner\_zone\_grid**: The grid spacing (and, in a sense, the inner zone radial bin size) for the inner zone in terms of the inner zone smallest radius. Two inner zone radii will be considered different, if they are more than the grid spacing apart. A very tiny number or a value of zero will result in a complete separation of all inner zone radii into separate bins. If a very large number is chosen, the inner zone reverts to a statistical zone containing only one radial bin.

- **mpole\_ignore\_inner\_zone**: 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 center of mass.

- **mpole\_max\_radial\_zones**: 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 cartesian runs. In these cases the user can specify several outer radial zones each having their own radial bin size determination rule.

- **mpole\_zone\_radius\_fraction\_n**: The fraction of the maximum domain radius defining the n-th outer zone radial limit. 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\_zone\_type\_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_zone_scalar_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_zone_exponent_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_2D_symmetry_plane</td>
<td>logical</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>mpole_3D_axisymmetry</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_inner_zone_size</td>
<td>integer</td>
<td>16</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>mpole_inner_zone_grid</td>
<td>real</td>
<td>0.1</td>
<td>&gt; = 0.0</td>
</tr>
<tr>
<td>mpole_ignore_inner_zone</td>
<td>logical</td>
<td>false</td>
<td>&gt; = 0.0</td>
</tr>
<tr>
<td>mpole_max_radial_zones</td>
<td>integer</td>
<td>1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>mpole_zone_radius_fraction_n</td>
<td>real</td>
<td>1.0</td>
<td>between 0.0 and 1.0</td>
</tr>
<tr>
<td>mpole_zone_type_n</td>
<td>string</td>
<td>“exponential”</td>
<td>“logarithmic”</td>
</tr>
<tr>
<td>mpole_zone_scalar_n</td>
<td>real</td>
<td>1.0</td>
<td>&gt; 0.0 (exponential)</td>
</tr>
<tr>
<td>mpole_zone_exponent_n</td>
<td>real</td>
<td>-</td>
<td>any ≠ 0 (logarithmic)</td>
</tr>
</tbody>
</table>

8.10.3.3 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 17). It may be included by setup or Config by including:

```plaintext
physics/Gravity/GravityMain/Poisson/Multigrid
```

The multigrid solver may also be included stand-alone using:

```plaintext
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>
<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>

### 8.10.4 HYPRE

As a part of implicit time advancement we end up with a system of equations that needs to be 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 it’s extents. In FLASH4 this information can be computed easily using a combination of `Grid_getBkCornerID` and `Grid_getBkIndexLimits`. All leaf blocks at a given refinement level form a HYPRE part. So number of parts in a typical FLASH4 grid would be give 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 the 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 16 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 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 HYPER 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 HYPER 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 HYPER SPLIT solver in FLASH4 is purely experimental.

Customizing solvers: HYPER 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 HYPER parameters might require direct manipulation of FLASH4 code.

The default settings use PCG as the solver with AMG as preconditioner. The following parameters can be tweaked at run time,

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gr_hyprePType</td>
<td>string</td>
<td>&quot;hypre_amg&quot;</td>
<td>Algebraic Multigrid as Preconditioner</td>
</tr>
<tr>
<td>gr_hypreMaxIter</td>
<td>integer</td>
<td>10000</td>
<td>Maximum number of iterations</td>
</tr>
<tr>
<td>gr_hypreRelTol</td>
<td>real</td>
<td>$1 \times 10^{-12}$</td>
<td>Maximum ratio of the norm of the residual to that of the initial residue</td>
</tr>
<tr>
<td>gr_hypreSolverType</td>
<td>string</td>
<td>&quot;hypre_pcg&quot;</td>
<td>Type of linear solver, Preconditioned Conjugate gradient</td>
</tr>
</tbody>
</table>

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-alpha also currently take geometry into account, see Section 8.6.3 above.
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>(cylindrical with a different ordering of coordinates)</td>
<td>3</td>
<td>—</td>
<td>n</td>
<td>r</td>
<td>φ</td>
</tr>
</tbody>
</table>

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 IAXIS, JAXIS, and KAXIS dimensions in FLASH4-alpha, 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 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 $\theta$ and $\phi$ 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 $\phi$ in these geometries implies the assumption of axial symmetry, and this is noted in Table 8.7. Similarly, omitting both $\phi$ and $\theta$ in spherical coordinates implies an assumption of complete spherical symmetry. When $\phi$ 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 user can therefore determine flow-control based on the geometry type (see Figure 8.13). 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_getBkCenterCoords and Grid_getBkPhysicalSize.

The volume of a single cell can be obtained via the Grid_getSingleCellVol or the Grid_getPointData routine. Use the Grid_getBkData, 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
#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 8.13: Branching based on geometry type

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 13), 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 \(\Delta r = r_r - r_l\); 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

```plaintext
geometry = "cartesian"
```

in the runtime parameter file.
Cell Volume in Cartesian Coordinates

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>$\Delta x$</td>
</tr>
<tr>
<td>2-d</td>
<td>$\Delta x\Delta y$</td>
</tr>
<tr>
<td>3-d</td>
<td>$\Delta x\Delta y\Delta z$</td>
</tr>
</tbody>
</table>

8.11.4.2 Cylindrical geometry

To run FLASH with cylindrical coordinates, the **geometry** parameter must be set thus:

```plaintext
geometry = "cylindrical"
```

Cell Volume in Cylindrical Coordinates

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>$\pi(r_r^2 - r_l^2)$</td>
</tr>
<tr>
<td>2-d</td>
<td>$\pi(r_r^2 - r_l^2)\Delta z$</td>
</tr>
<tr>
<td>3-d</td>
<td>$\frac{1}{2}(r_r^2 - r_l^2)\Delta z\Delta\phi$</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 8.14 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

$$A = \pi(r_r^2 - r_l^2) .$$

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

$$f_3 = \frac{A_1f_1 + A_2f_2}{A_3} .$$

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

```plaintext
geometry = "spherical"
```

in the runtime parameter file.

Cell Volume in Spherical Coordinates

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>$\frac{4}{3}\pi(r_r^3 - r_l^3)$</td>
</tr>
<tr>
<td>2-d</td>
<td>$\frac{2}{3}\pi(r_r^3 - r_l^3)\cos(\theta_l) - \cos(\theta_r))$</td>
</tr>
<tr>
<td>3-d</td>
<td>$\frac{1}{3}(r_r^3 - r_l^3)\cos(\theta_l) - \cos(\theta_r))\Delta\phi$</td>
</tr>
</tbody>
</table>

If the minimum radius is chosen to be zero ($x_{\text{min}} = 0.$), the left-hand boundary type should be reflecting.
Figure 8.14: 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.

### 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

```
geometry = "polar"
```

in the runtime parameter file.

<table>
<thead>
<tr>
<th>Cell Volume in Polar Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d $\pi(r_2^2 - r_1^2)$</td>
</tr>
<tr>
<td>2-d $\frac{1}{2}(r_2^2 - r_1^2)\Delta\phi$</td>
</tr>
<tr>
<td>3-d $\frac{1}{2}(r_2^2 - r_1^2)\Delta\phi\Delta z$ (not supported)</td>
</tr>
</tbody>
</table>

As in other non-cartesian geometries, if the minimum radius is chosen to be zero ($xmin = 0.$), 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 automatically added by the setup script when the -gridinterpolation=monotonic option is in effect (which is the case by default, unless -gridinterpolation=native was specified). The “monotonic” interpolation scheme used in both cases is taking geometry into consideration and is appropriate for all supported geometries.

FLASH Transition

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

\[
\langle f \rangle_{i,j} = \frac{V_{ic,jc} \langle f \rangle_{ic,jc} + V_{ic+1,jc} \langle f \rangle_{ic+1,jc} + V_{ic,jc+1} \langle f \rangle_{ic,jc+1} + V_{ic+1,jc+1} \langle f \rangle_{ic+1,jc+1}}{V_{i,j}},
\]

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 Custom Regions

FLASH4-alpha implements a two options for custom regions that may be placed in the domain. The custom regions currently implemented are QuietStart and Piston.

8.12.1 Quiet Start

A quiet start region is a region in the domain that is not allowed to undergo hydrodynamic evolution until certain conditions are met. In FLASH4-alpha, quiet start regions must be rectangular and the criterion for release of a quiet region is based on temperature. To use quiet start, set the runtime parameter useQuietStart to true and specify the initial boundaries of the quiet start region. The region is specified with the parameters quietStartXmin, quietStartXmax, quietStartYmin, quietStartYmax, quietStartZmin, and quietStartZmax. The temperature at which the quiet start region will release and begin to evolve is set by quietStartTemp.
8.12.2 Piston

A rectangular “piston” can be input into the domain in FLASH4-alpha. To use a piston, set the runtime parameter `usePiston` to true. The initial location of the piston is set by the parameters `pistonXmin`, `pistonXmax`, `pistonYmin`, `pistonYmax`, `pistonZmin`, and `pistonZmax`. The piston velocity is set by `pistonVel{ x, y, z }` where the last letter of the velocity specifies the direction of the piston. Note that the piston has only been tested for movement in one direction only.

8.13 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 27) and sfocu (Chapter 26) 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-alpha supports multiple IO implementations: direct, serial and parallel implementations as well as support for different parallel libraries. In addition, FLASH4-alpha 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-alpha 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-alpha 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-alpha 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

```
./setup Sod -2d -auto +ug
```
9.2. OUTPUT FILES

./setup Sod -2d -auto +ug +parallelio

If you do not want to use IO, you need to explicitly specify on the setup line that it should not be included, as in this example:

./setup Sod -2d -auto +noio

To setup a problem using the Parallel-NetCDF library the user should include either

-\texttt{unit=IO/IOMain/pnetcdf/PM} or \texttt{-unit=IO/IOMain/pnetcdf/UG}

to the setup line. The predefined shortcut for including the Parallel-NetCDF library is

+\texttt{pnetcdf}

Note that Parallel-NetCDF IO unit does not have a serial implementation.

If you are using non-fixed blocksize the shortcut

+nofbs

will bring in both Uniform Grid, set the mode to nonfixed blocksize, and choose the appropriate IO.

\textbf{Note:}

Presently, nonfixed blocksize is only supported by HDF5 parallel IO.

In keeping with the FLASH code architecture, the F90 module \texttt{IO.data} stores all the data with IO unit scope. The routine \texttt{IO.init} is called once by \texttt{Driver_initFlash} and initializes IO data and stores any runtime parameters. See Chapter 10.

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 22 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 \texttt{IO.writeCheckpoint}. Users usually will not need to call this routine directly because the FLASH IO unit calls \texttt{IO.writeCheckpoint} from the routine \texttt{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 `.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 `.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 `.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 single 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 `.dump_checkpoint` or `.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 <code>rolling_checkpoint</code>, then the checkpoint number is reset to 0. There will be at most <code>rolling_checkpoint</code> 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 <code>wall_clock_checkpoint</code> 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>
9.2. OUTPUT FILES

Table 9.2: Checkpoint IO parameters (continued).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>A logical variable indicating whether the simulation is restarting from a checkpoint file (.true.) or starting from scratch (.false.).</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-alpha 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:

```plaintext
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.

<table>
<thead>
<tr>
<th>FLASH Transition</th>
</tr>
</thead>
</table>

In FLASH2 a few variables like density and pressure were output to the plotfiles by default. Because FLASH4-alpha supports a wider range of simulations, it makes no assumptions that density or pressure variables are even included in the simulation. In FLASH4-alpha 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-alpha 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 \texttt{nend}, \texttt{zFinal}, or \texttt{tmax}. 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>
<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>HUGE(1.)</td>
<td>The change in cosmological redshift desired between subsequent plotfiles.</td>
</tr>
<tr>
<td>plot_var_1, \ldots, 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 \texttt{IO\_writeParticles}. Again the user will not usually call this function directly because the routine \texttt{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 \texttt{.dump\_particle\_file} in the output directory. The following runtime parameters pertain to controlling particle files:

<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.</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>

\textbf{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 18 for more information about the particles data structure in FLASH4-alpha. Additionally, filtered particles are not implemented in FLASH4-alpha.

All the code necessary to output particle data is contained in the \texttt{IO} subunit called \texttt{IOParticles}. Whenever the \texttt{Particles} unit is included in a simulation the correct \texttt{IOParticles} subunit will also be included. For example as setup:

\texttt{.setup IsentropicVortex -2d -auto -unit=Particles +ug}

will include the \texttt{IO} unit \texttt{IO/IOMain/hdf5/serial/UG} and the correct \texttt{IOParticles} subunit \texttt{IO/IOParticles/hdf5/serial/UG}. The shortcuts \texttt{+parallelio}, \texttt{+pnetcdf}, \texttt{+ug} will also cause the setup script to pick up the correct \texttt{IOParticles} subunit as long as a \texttt{Particles} unit is included in the simulation.

\textbf{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 \texttt{flash.dat}. A default routine \texttt{IO\_writeIntegralQuantities} is provided to output standard measures for hydrodynamic simulations. The user should copy and modify the routine
IO_writeIntegralQuantities into a given simulation directory to store any quantities other than the default values. Two runtime parameters pertaining to the flash.dat file are listed in the table below.

Table 9.5: flash.dat runtime parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>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 (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 9.6: General IO runtime parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>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. output_directory can be an absolute or relative path.</td>
</tr>
<tr>
<td>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 (flash.log)</td>
</tr>
<tr>
<td>useCollectiveHDF5</td>
<td>BOOLEAN</td>
<td>.false.</td>
<td>When using the parallel HDF5 implementation of IO, will enable collective mode for HDF5. By default, independent mode is used.</td>
</tr>
</tbody>
</table>
9.3 Restarts and Runtime Parameters

FLASH4-alpha 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-alpha 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-alpha, 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, nx, ny, and nz. 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-alpha 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

```c
IO_setScalar("nx", NXB)
IO_setScalar("ny", NYB)
IO_setScalar("nz", 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-alpha, 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 Grid Variables

In FLASH4-alpha a user can allocate space for a scratch or temporary variable with grid scope using the Config keyword GRIDVAR (see Section 5.5.1). To output these scratch variables, the user only needs to
set the values of the runtime parameters \texttt{plot\_grid\_var\_1}, \texttt{plot\_grid\_var\_2}, etc., by setting them in the \texttt{flash.par} file. For example to output the magnitude of vorticity with a declaration in a Config file of GRIDVAR \texttt{mvrt}:

\begin{verbatim}
plot_grid_var_1 = "mvrt"
\end{verbatim}

Note that the post-processing routines like \texttt{fidelr} 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 \texttt{basenm}) 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:

\begin{verbatim}
basename_s0000.{ h5 ncmi }_chk_0000 ,
\end{verbatim}

where \texttt{h5} or \texttt{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:

\begin{verbatim}
basename_s0000.{ h5 ncmi }_plt_{ crn cnt }_0000 ,
\end{verbatim}

where \texttt{h5} or \texttt{ncmi} is picked depending on the IO implementation used, \texttt{crn} and \texttt{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 \texttt{plotfileNumber}. \texttt{crn} is reserved, even though corner data output is not presently supported by FLASH4-alpha’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 fidelr visualization tool. In FLASH4-alpha the name of the file is irrelevant to fidelr3.0 (see Chapter 27). 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-alpha 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, fidelr 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. OUTPUT FORMATS

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-alpha 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 fidl3.0 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.

```bash
./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-alpha 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-alpha.

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-alpha can make use of this mode by setting the runtime parameter useCollectiveHDF5 to true.
CHAPTER 9. IO UNIT

FLASH Transition

We recommend that HDF5 version 1.6 or later be used with the HDF5 IO implementations with FLASH4-alpha. 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 $\geq 1.8$ 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-alpha 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 $m_{dim}$, or the maximum dimensions supported by FLASH’s grids, which is three, rather than $n_{dim}$. PARAMESH 4.0 and PARAMESH 4dev, however, do requires 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-alpha HDF5 format is summarized in Table 9.7.

<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>
Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct sim_info_t {</td>
<td></td>
</tr>
<tr>
<td>int file_format_version;</td>
<td></td>
</tr>
<tr>
<td>char setup_call[400];</td>
<td></td>
</tr>
<tr>
<td>char file_creation_time[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char flash_version[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char build_date[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char build_dir[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char build_machine[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char cflags[400];</td>
<td></td>
</tr>
<tr>
<td>char fflags[400];</td>
<td></td>
</tr>
<tr>
<td>char setup_time_stamp[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>char build_time_stamp[MAX_STRING_LENGTH];</td>
<td></td>
</tr>
<tr>
<td>} sim_info_t;</td>
<td></td>
</tr>
<tr>
<td>sim_info_t sim_info;</td>
<td></td>
</tr>
<tr>
<td>sim_info.file_format_version:</td>
<td>An integer giving the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file.</td>
</tr>
<tr>
<td>sim_info.setup_call:</td>
<td>The complete syntax of the setup command used when creating the current FLASH executable.</td>
</tr>
<tr>
<td>sim_info.file_creation_time:</td>
<td>The time and date that the file was created.</td>
</tr>
<tr>
<td>sim_info.flash_version:</td>
<td>The version of FLASH used for the current simulation. This is returned by routine setup_flashVersion.</td>
</tr>
<tr>
<td>sim_info.build_date:</td>
<td>The date and time that the FLASH executable was compiled.</td>
</tr>
<tr>
<td>sim_info.build_dir:</td>
<td>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.</td>
</tr>
<tr>
<td>sim_info.build_machine:</td>
<td>The name of the machine (and anything else returned from uname -a) on which FLASH was compiled.</td>
</tr>
<tr>
<td>sim_info.cflags:</td>
<td>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.</td>
</tr>
<tr>
<td>sim_info.fflags:</td>
<td>The f compiler flags used in the given simulation.</td>
</tr>
<tr>
<td>sim_info.setup_time_stamp:</td>
<td>The date and time the given simulation was setup. The routine setup_buildstamp is created by the setup script at compile time.</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.build_time_stamp:</td>
<td>The date and time the given simulation was built. The routine setup_buildstamp is created by the setup 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:

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       int_list_t int_list(numIntParams)
A linked list holding the names and values of all the integer runtime parameters.

real runtime parameters          real_list_t real_list(numRealParams)
A linked list holding the names and values of all the real runtime parameters.

string runtime parameters        str_list_t str_list(numStrParams)
A linked list holding the names and values of all the string runtime parameters.

logical runtime parameters       log_list_t log_list(numLogParams)
A linked list holding the names and values of all the logical runtime parameters.
Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<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`  
| block size        | `real size(mdim,globalNumBlocks)` |
|                  | This array stores the dimensions of the current block. |
Table 9.7: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(1,blockID)</td>
<td>$x$ size</td>
</tr>
<tr>
<td>size(2,blockID)</td>
<td>$y$ size</td>
</tr>
<tr>
<td>size(3,blockID)</td>
<td>$z$ 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 (Paramesh4.0 and Paramesh4dev only!)</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(nx,nynz,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. 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></td>
</tr>
<tr>
<td>localnp</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 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 is sized as the total number of particles in the simulation. Note that all the particle properties are real values.</td>
</tr>
</tbody>
</table>

9.9.1.4 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/](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.

```bash
./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:

```bash
-unit=IO/IOMain/direct/UG or -unit=IO/IOMain/direct/PM
```

9.9.4 Output Side Effects

In FLASH4-alpha 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 27). In addition, ViSit version 10.0 and higher (see Chapter 25) can natively read FLASH4-alpha 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-alpha 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:

```bash
#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:

```bash
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 \texttt{lrefine\_min} to a value \(\geq 1\) to create more blocks. The user could also set the runtime parameters \texttt{nblockx}, \texttt{nblocky}, \texttt{nblockz} to make a bigger problem.

Just like any other simulation, the user can change the number of zones in a simulation using \texttt{-nxb=numZones} on the setup line.

9.11.1 Online tips for working with the IO Unit

Please see the website for further hints on efficient usage of the IO unit.

9.12 Chombo

In FLASH4-alpha 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 \texttt{+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.

At the present time we have only created an interface to Chombo write routines and this is restricted to writing data owned by the Grid unit; data owned by other units is not written to the output file. We do not have an interface to Chombo read routines and so restarts are not yet supported.
Chapter 10

Runtime Parameters Unit

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
Figure 10.2: Section of output log showing runtime parameters values

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-alpha 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

FLASH has the ability to track multiple fluids, each of which can have its own properties. The Multispecies unit handles setting, querying, and operating on the properties of fluids. The advection and normalization of species is described in Chapter 13.

11.1 Defining Species

The names and properties of fluids are accessed by using their constant integer values defined in the header files. The species names are defined in the Config file of the Simulation. The name of the specie, for example AIR, is listed in the Config file with keyword SPECIES. The setup procedure transforms those names into an accessor integer 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 bracketed by SPECIES_BEGIN and SPECIES_END. To access the species in your code, use the index listed in Flash.h, for example AIR_SPEC. 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 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.
CHAPTER 11. MULTISPECIES UNIT

# Portion of a "Config" file for a Simulation
SPECIES AIR
SPECIES SF6

Figure 11.2: 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: Header file Flash.h showing integer definition of fluid species.

```c
FLASH Transition

In FLASH2, you found the integer index of a species by using find_fluid_index. In FLASH4-alpha, 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 specie 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Property Name</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>“num total”</td>
<td>A</td>
<td>Number of protons and neutrons in nucleus</td>
<td>real</td>
</tr>
<tr>
<td>“num positive”</td>
<td>Z</td>
<td>Atomic number</td>
<td>real</td>
</tr>
<tr>
<td>“num neutral”</td>
<td>N</td>
<td>Number of neutrons</td>
<td>real</td>
</tr>
<tr>
<td>“num negative”</td>
<td>E</td>
<td>Number of electrons</td>
<td>real</td>
</tr>
<tr>
<td>“binding energy”</td>
<td>BE</td>
<td>Binding Energy</td>
<td>real</td>
</tr>
<tr>
<td>“adiabatic index”</td>
<td>GAMMA</td>
<td>Ratio of heat capacities</td>
<td>real</td>
</tr>
</tbody>
</table>

Table 11.1: Properties available through the Multispecies unit.
11.3. ROUTINE DESCRIPTIONS

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

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.

11.3 Routine Descriptions

We now briefly discuss the various interfaces to the multifluid database that are of interest to the user. Many of these functions 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).

FLASH Transition
In FLASH2, you could set multiple properties at once in a call to add_fluid_to_db. In FLASH4-alpha, 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 $A_i$, $Z_i$, ..., $\gamma_i$) 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 $A_i$, $Z_i$, ..., $\gamma_i$) 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 $A_i$, $Z_i$, ..., $\gamma_i$) 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
\[ \frac{1}{A} = \sum_i X_i \frac{A_i}{A_i} \]
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
\begin{verbatim}
call Multispecies_getSumInv(A, abarinv, xn(:))
abar = 1.e0 / abarinv
\end{verbatim}
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 $A_i$, $Z_i$, ..., $\gamma_i$) is denoted $P_i$, the sum calculated is
\[ \sum_i \frac{X_i}{A_i} P_i \]

- **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 $A_i$, $Z_i$, ..., $\gamma_i$) 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.
11.4 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

...`

11.5 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... is defined in the header file 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

```
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).
Table 12.1: Available Physical Constants

<table>
<thead>
<tr>
<th>String Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>Gravitational constant G</td>
</tr>
<tr>
<td>speed of light</td>
<td>Speed of light</td>
</tr>
<tr>
<td>Planck</td>
<td>Planck’s constant</td>
</tr>
<tr>
<td>electron charge</td>
<td>charge of an electron</td>
</tr>
<tr>
<td>electron mass</td>
<td>mass of an electron</td>
</tr>
<tr>
<td>proton mass</td>
<td>Mass of a proton</td>
</tr>
<tr>
<td>fine-structure</td>
<td>fine-structure constant</td>
</tr>
<tr>
<td>Avogadro</td>
<td>Avogadro’s Mole Fraction</td>
</tr>
<tr>
<td>Boltzmann</td>
<td>Boltzmann’s constant</td>
</tr>
<tr>
<td>ideal gas constant</td>
<td>ideal gas constant</td>
</tr>
<tr>
<td>Wien</td>
<td>Wien displacement law constant</td>
</tr>
<tr>
<td>Stefan-Boltzmann</td>
<td>Stefan-Boltzmann constant</td>
</tr>
<tr>
<td>pi</td>
<td>Pi</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>Euler</td>
<td>Euler-Mascheroni constant</td>
</tr>
</tbody>
</table>

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 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: 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.

- **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.

- **PhysicalConstants_init** Initializes the Physical Constants Unit by loading all constants. This routine is called by Driver_initFlash and must be called before the first invocation of PhysicalConstants_get. In general, the user does not need to invoke this call.

- **PhysicalConstants_list** Lists the available physical constants in a snappy table.

- **PhysicalConstants_listUnits** Lists all the units available for optional conversion.

- **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 $H_0 = 100 \text{ km/s/Mpc}$:

<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>LFLY</td>
<td>3.0856775807E24</td>
<td>1 Mpc</td>
</tr>
<tr>
<td>time</td>
<td>TFLY</td>
<td>2.05759E17</td>
<td>$\frac{2}{H_0}$</td>
</tr>
<tr>
<td>mass</td>
<td>MFLY</td>
<td>9.8847E45</td>
<td>5.23e12 Msun</td>
</tr>
</tbody>
</table>

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

Hydrodynamics Units

Figure 13.1: The Hydro unit directory tree.

The Hydro unit solves Euler’s equations for compressible gas dynamics in one, two, or three spatial
dimensions. These equations can be written in conservative form as

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \tag{13.1} \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P &= \rho \mathbf{g} \tag{13.2} \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] &= \rho \mathbf{v} \cdot \mathbf{g} , \tag{13.3}
\end{align*}
\]

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,

\[
E = \epsilon + \frac{1}{2} |\mathbf{v}|^2 , \tag{13.4}
\]

\( \mathbf{g} \) 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

\[
P = (\gamma - 1) \rho \epsilon , \tag{13.5}
\]

where \( \gamma \) is the ratio of specific heats. More general equations of state are discussed in Section 14.2 and Section 14.3.

In regions where the kinetic energy greatly dominates the total energy, computing the internal energy using

\[
\epsilon = E - \frac{1}{2} |\mathbf{v}|^2 \tag{13.6}
\]

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

\[
\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + P) \mathbf{v}] - \mathbf{v} \cdot \nabla P = 0 . \tag{13.7}
\]

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 (13.7) and the velocities from the momentum equation. Numerical experiments using the PPM solver included with FLASH showed that using (13.7) 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

\[
\frac{\partial \rho X_\ell}{\partial t} + \nabla \cdot (\rho X_\ell \mathbf{v}) = 0 , \tag{13.8}
\]

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,

\[
\frac{\partial \rho \phi_\ell}{\partial t} + \nabla \cdot (\rho \phi_\ell \mathbf{v}) = 0 , \tag{13.9}
\]

where \( \phi_\ell \) is the \( \ell \)th mass scalar. Mass scalars are restricted between 0 and 1, but are not renormalized in order to sum to 1. 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. See Section 5.5.1 for more details.
### 13.1. Gas hydrodynamics

#### 13.1.1 Usage

Two gas hydrodynamic solvers supplied in the release of FLASH4-alpha 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 13.1.2 and Section 13.1.3 and implemented in the directory tree physics/Hydro/HydroMain/split/PPM and physics/Hydro/HydroMain/unsplit/Hydro_Unsplit. Extensions for multitemperature applications are described in Section 13.1.4.

Current and future implementations of Hydro uses the runtime parameters and solution variables described in Table 13.1 and Table 13.2. Additional runtime parameters used solely by the PPM method are described in PPM, and more runtime parameters specific for the new unsplit hydro solver are further described in Hydro_Unsplit.

#### 13.1.2 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

---

**Table 13.1: Runtime parameters used with the hydrodynamics (Hydro) unit.**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eintSwitch</td>
<td>real</td>
<td>0</td>
<td>If $\epsilon &lt; eintSwitch \cdot \frac{1}{2}</td>
</tr>
<tr>
<td>irenorm</td>
<td>integer</td>
<td>0</td>
<td>If equal to one, renormalize multifluid abundances following a hydro update; else restrict their values to lie between smallx and 1.</td>
</tr>
<tr>
<td>cfl</td>
<td>real</td>
<td>0.8</td>
<td>Courant-Friedrichs-Lewy (CFL) factor; must be less than 1 for stability in explicit schemes</td>
</tr>
</tbody>
</table>

**Table 13.2: Solution variables used with the hydrodynamics (Hydro) unit.**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dens</td>
<td>PER_VOLUME</td>
<td>density</td>
</tr>
<tr>
<td>velx</td>
<td>PER_MASS</td>
<td>$x$-component of velocity</td>
</tr>
<tr>
<td>vely</td>
<td>PER_MASS</td>
<td>$y$-component of velocity</td>
</tr>
<tr>
<td>velz</td>
<td>PER_MASS</td>
<td>$z$-component of velocity</td>
</tr>
<tr>
<td>pres</td>
<td>GENERIC</td>
<td>pressure</td>
</tr>
<tr>
<td>ener</td>
<td>PER_MASS</td>
<td>specific total energy ($T + U$)</td>
</tr>
<tr>
<td>temp</td>
<td>GENERIC</td>
<td>temperature</td>
</tr>
</tbody>
</table>
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.`.

### 13.1.3 The unsplit hydro solver

A new directionally unsplit pure hydrodynamic solver (unsplit hydro) is provided in the FLASH4-alpha release. The method basically adopts a MUSCL-Hancock type formulation (zone-edge data-extrapolated method) that provides second-order solution accuracy in both space and time. Recently, the order of spatial accuracy has been extended to implement the 3rd order PPM and 5th order Weighted ENO (WENO)
### Table 13.3: Additional runtime parameters for Interpolation Schemes in the unsplit hydro solver

<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 NGUARD=6, and should include +supportWENO in setup line</td>
</tr>
<tr>
<td>transOrder</td>
<td>integer</td>
<td>4</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 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>&quot;vanLeer&quot;</td>
<td>Slope limiter: &quot;MINMOD&quot;, &quot;MC&quot;, &quot;VANLEER&quot;, &quot;HYBRID&quot;, &quot;LIMITED&quot;</td>
</tr>
<tr>
<td>LimitedSlopeBeta</td>
<td>real</td>
<td>1.0</td>
<td>Slope parameter specific for the &quot;LIMITED&quot; 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_steeplening</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 NGUARD=6</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 use_gravConsv=.false.</td>
</tr>
</tbody>
</table>

methods. This unsplit hydro solver is a reduced version of the Unsplit Staggered Mesh (USM) MHD solver (see details in Section 13.3.4) that has been available in previous FLASH3 releases.

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 order Godunov methods; a strong shock/rarefaction detection algorithm and handling of grid-aligned shock instabilities such as carbuncle and odd-even decoupling phenomena 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 upon preserving flow symmetries as compared to splitting formulations. Also, the scheme used in this unsplit algorithm can take a wide range of CFL stability limits (e.g., 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).

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 they use them.
Table 13.4: Additional runtime parameters for Riemann Solvers in the unsplit hydro solver

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RiemannSolver</td>
<td>string</td>
<td>&quot;Roe&quot;</td>
<td>Different choices for Riemann solver. &quot;LLF (local Lax-Friedrichs)&quot; &quot;HLL&quot;, &quot;HLLC&quot;, &quot;HYBRID&quot;, &quot;ROE&quot;, and &quot;Marquina&quot;</td>
</tr>
<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 the Roe solver</td>
</tr>
<tr>
<td>entropyFixMethod</td>
<td>string</td>
<td>&quot;HARTENHYMAN&quot;</td>
<td>Entropy fix method for the Roe solver. &quot;HARTEN&quot;, &quot;HARTENHYMAN&quot;</td>
</tr>
</tbody>
</table>

- 5th order WENO: The 5th order method WENO requires NGUARD=6 in FLASH's implementation. To use WENO, users should include +supportWENO in the setup for both unsplit hydro and MHD solvers. Users should expect that this larger number of guard cells will require increased number of inter-processor data exchanges, resulting in slower performance during parallel computations.

- Extended stencil: When NGUARD=6 is used, users should also use nxb, nyb, nzb larger than 2*NGUARD. For example, specifying -nxb=16 in the setup works well for 1D cases. Once setting up NGUARD=6, users still can use FOG, MH, or PPM without changing NGUARD back to 4.

- transOrder: The first order method transOrder=1 is somewhat diffusive but can be applicable in most cases; the 2nd order method sometimes generates inaccurate solutions; the 3rd order method (although known to have dispersive errors) is the least diffusive and most accurate, providing extra stabilities using larger stencils for its upwinding formulation. transOrder=4 is default and uses a slope limiter between the 1st and 3rd order accurate methods to minimize oscillations in taking upwinded numerical derivatives at discontinuities.

- EOSforRiemann: EOSforRiemann = .true. will call (expensive) EOS routines to compute consistent adiabatic indices (e.g., gmc, 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 .false. for computational efficiency and performance. On the other hand, for a degenerate gas, one can enable this switch to compute thermodynamically consistent gmc, game, which in turn are used to compute sound speed and internal energy in Riemann flux calculations. When disabled, interpolations will be used instead to get approximations of gmc, 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.

- 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., use_gravHalfUpdate=.true.). This will provide a second-order accuracy with respect to coupling gravitational accelerations to hydrodynamics. With use_gravHalfUpdate=.true., users can choose use_gravConsv=.true. to adopt conservative forms (expensive); although, a primitive counterpart update should be efficient and accurate enough for most cases. Otherwise, if 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 a first order accuracy. We also have included a new approach to update gravity accelerations at n + 1/2 time step (i.e., use_gravPotUpdate=.true.). In this new approach, the gravity potentials are calculated by calling Poisson solver, giving more accurate values at cell-interfaces. The option use_gravPotUpdate=.true. only works with the primitive update, use_gravConsv=.false..
13.1. GAS HYDRODYNAMICS

Unsplit Hydro Solver vs. Unsplit Staggered MHD Mesh Solver

One major difference between the new 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 conductivity, can be included in the unsplit hydro solver for simulating diffusive processes. Please see related descriptions in Section 13.3.5.

13.1.4 Multitemperature extension for Hydro

The multitemperature hydro extensions are implemented in several implementation directories named multiTemp. 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.

As of FLASH4-alpha, this is implemented for the split PPM solver. This subsection thus extends Section 13.1.2.

The basic idea of the multitemperature extensions is to consider a fluid with three components, namely, ions, electrons, and radiation. The three components may in general not be in local thermodynamic equilibrium with each other and can be described by different temperatures. The Hydro unit describes the motion and evolution of this three-component fluid, with the help of appropriate equations of state.

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
Table 13.5: 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>ion</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></td>
<td>PER_MASS (mass scalar)</td>
<td>specific electron entropy</td>
</tr>
</tbody>
</table>

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 13.5 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.

13.1.5 Chombo compatible Hydro

This is a slightly modified version of the split Hydro solver (Section Section 13.1.2) 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 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 >= 8, BRMeshRefineBlockFactor >= 8, refRatio = 2 and...
13.2 Relativistic hydrodynamics (RHD)

13.2.1 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 13.2.2, 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.

13.2.2 Equations

The motion of an ideal fluid in special relativity is described by the system of conservation laws

$$\frac{\partial}{\partial t}\begin{pmatrix} D \\ m \\ E \end{pmatrix} + \nabla \cdot \begin{pmatrix} Dv \\ mv + pI \\ m \end{pmatrix} = 0,$$

(13.10)

where $D$, $m$, $E$, $v$ and $p$ define, respectively, the fluid density, momentum density, total energy density, three-velocity and pressure of the fluid. (13.10) 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 ((13.10)). The current version of FLASH supports only the ideal equation of state, for which the specific enthalpy $h$ may be expressed as

$$h = 1 + \Gamma \frac{p}{\Gamma - 1 \rho},$$

(13.11)

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 ("gamma").
As in classical hydrodynamics, relativistic fluids may be described in terms of a state vector of conservative, \( \mathbf{U} = (D, \mathbf{m}, E) \), or primitive, \( \mathbf{V} = (\rho, \mathbf{v}, p) \), variables. The connection between the two sets is given by

\[
D = \gamma \rho, \quad m = \rho \gamma \mathbf{v}, \quad E = \rho \gamma^2 - p, \quad (13.12)
\]

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 \( \mathbf{V} \) in terms of \( \mathbf{U} \), is

\[
\rho = \frac{D}{\gamma}, \quad \mathbf{v} = \frac{m}{E + p}, \quad p = Dh \gamma - E. \quad (13.13)
\]

This inverse map is not trivial due to the non-linearity introduced by the Lorentz factor \( \gamma \); it can be shown, in fact, that (13.13) can be combined together to obtain the following implicit expression for \( p \):

\[
p = Dh(p, \tau(p))\gamma(p) - E. \quad (13.14)
\]

(13.14) has to be solved at least once per time step in order to recover the pressure from a set of conservative variables \( \mathbf{U} \). 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 \texttt{rhd_conserveToPrimitive.F90} and \texttt{rhd_primitiveToConserve.F90} source files.

### 13.2.3 Relativistic Equation of State

A variant version of the ideal gamma law \texttt{Eos_wrapped.F90} routine is required by the \texttt{RHD} unit and is provided in \texttt{Eos/EosMain/Gamma/RHD}. In order to do so the unit requires a typical \texttt{Config} file which should look like this:

```
REQUIRES physics/Eos/EosMain/Gamma/RHD
```

For this specific purpose, the current \texttt{RHD} implementation supports \texttt{MODE_DENS_EI} (a default mode) and \texttt{MODE_DENS_PRES} only (but not \texttt{MODE_DENS_TEMP}) in making a \texttt{Eos_wrapped} call.

### 13.2.4 Additional Runtime Parameter

One additional runtime parameter used with the \texttt{RHD} unit is

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{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>

### 13.3 Magnetohydrodynamics (MHD)

#### 13.3.1 Description

The FLASH4-alpha code includes two magnetohydrodynamic (MHD) units that represent two different algorithms. The first is the eight-wave model (\texttt{8Wave}) by Powell \textit{et al.} (1999) that is already present in FLASH2. The second is a newly implemented unsplit staggered mesh algorithm (USM or \texttt{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
13.3. MAGNETOHYDRODYNAMICS (MHD)

strict physical law. To improve such unphysical effects in simulations, the new 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:

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v} - \mathbf{B} \mathbf{B}) + \nabla p_\ast &= \rho \mathbf{g} + \nabla \cdot \tau \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho E \mathbf{v} - \mathbf{B} \cdot \mathbf{v}) &= \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot (\mathbf{v} \cdot \tau + \sigma \nabla T) + \nabla \cdot (\mathbf{B} \times (\eta \nabla \times \mathbf{B})) \\
\frac{\partial \mathbf{B}}{\partial t} + \nabla \times (\mathbf{v} \mathbf{B} - \mathbf{B} \mathbf{v}) &= -\nabla \times (\eta \nabla \times \mathbf{B})
\end{align*}
\]

where

\[
\begin{align*}
p_\ast &= p + \frac{B^2}{2}, \\
E &= \frac{1}{2} \mathbf{v}^2 + \epsilon + \frac{1}{2} \frac{B^2}{\rho}, \\
\tau &= \mu \left( (\nabla \mathbf{v}) + (\nabla \mathbf{v})^T - \frac{2}{3} \nabla \cdot \mathbf{v} \mathbf{I} \right)
\end{align*}
\]

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. \( \tau \) 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 13). In addition, both the eight-wave and the unsplit staggered mesh units declare additional plasma variables and runtime parameters, which are listed in Table 13.7 and Table 13.8.

13.3.2 Usage

In the current release of FLASH4-alpha, 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.
Table 13.7: 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 13.8: 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>&quot;none&quot;</td>
<td>System of units in which MHD calculations are to be performed. Acceptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>values are &quot;none&quot;, &quot;CGS&quot; and &quot;SI&quot;</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>

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 -gridinterpolation=native is automatically adopted if +8wave is specified in setup, otherwise, -gridinterpolation=native should be explicitly included in order to use the eight-wave solver without specifying +8wave. For instance, running a script ./setup magnetohD/BrioWu -1d -auto +8wave will properly setup the BrioWu problem for the 8Wave solver, and ./setup magnetohD/BrioWu -1d -auto +usm for the StaggeredMesh solver.

Supported configurations

Both MHD units currently support the uniform grid with FIXEDBLOCKSIZE and NONFIXEDBLOCKSIZE modes, and the adaptive grid with PARAMESH on Cartesian geometries. 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.

13.3.3 Algorithm: The Eight-wave Solver

The eight-wave magnetohydrodynamic (MHD) unit in the FLASH4-alpha code 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-alpha 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 FLASH4-alpha code, 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 13.3.4 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

\[
\frac{\partial B}{\partial t} + \nabla \cdot (vB - Bv) = -v\nabla \cdot B + \eta_a \nabla \nabla \cdot B ,
\]

with the artificial diffusion coefficient \( \eta_a \) chosen to match that of grid numerical diffusion. In the FLASH code, \( \eta_a = \frac{\lambda}{2} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2} \right)^{-1} \), 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-alpha and will be supported in a later release.

13.3.4 Algorithm: The Unsplit Staggered Mesh Solver

A directionally unsplit staggered mesh algorithm (USM), which solves ideal and non-ideal MHD governing equations (13.15) \sim (13.18) in multiple dimensions, is a new MHD solver. In FLASH4-alpha, a full 3D implementation has been included as a new official release (Lee, D., in preparation, 2011). 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, D., in preparation, 2011):

- **Quasi-linearization**: This step replaces the nonlinear system (13.15) \sim (13.18) 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 $\nabla \cdot B$ 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.
13.3. MAGNETOHYDRODYNAMICS (MHD)

- **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 13.9) 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 waves. 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):

\[
\begin{align*}
  b_{x,i+1/2,j}^{n+1} &= b_{x,i+1/2,j}^n - \frac{\Delta t}{\Delta y} \left\{ E_{z,i+1/2,j+1/2}^{n+1/2} - E_{z,i+1/2,j-1/2}^{n+1/2} \right\}, \quad (13.23) \\
  b_{y,i,j+1/2}^{n+1} &= b_{y,i,j+1/2}^n - \frac{\Delta t}{\Delta x} \left\{ -E_{z,i+1/2,j+1/2}^{n+1/2} + E_{z,i-1/2,j+1/2}^{n+1/2} \right\}. \quad (13.24)
\end{align*}
\]

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 13.2.

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

\[
(\nabla \cdot \mathbf{B})_{i,j}^{n+1} = \frac{b_{x,i+1/2,j}^{n+1} - b_{x,i-1/2,j}^{n+1}}{\Delta x} + \frac{b_{y,i,j+1/2}^{n+1} - b_{y,i,j-1/2}^{n+1}}{\Delta y} \quad (13.25)
\]

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 13.9 below).

To solve the above induction equations (13.23) and (13.24) 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:

\[
E_{x,i+1/2,j+1/2}^{n+1/2} = \frac{1}{4} \left( -F_{B_x,i+1/2,j}^{n+1/2} - F_{B_x,i+1/2,j+1}^{n+1/2} + G_{B_x,i,j+1/2}^{n+1/2} + G_{B_x,i+1,j+1/2}^{n+1/2} \right)
\]

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 \( B_z \), respectively (see details in Balsara and Spicer, 1999).

A high-order accurate version is also available by turning on a logical switch `modification` in the unsplit staggered mesh scheme, which takes Taylor series expansions of the cell-cornered electric field \( E_{x,i+1/2,j+1/2}^{n+1/2} \) in all directions, followed by taking an arithmetic average of them (Lee, 2006; Lee and Deane, 2008).

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 HLL (Einfeldt 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.

### 13.3.4.1 Slowly moving shock handling in PPM

A new dissipative mechanism introduced in FLASH4-alpha 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; Lee, D., 2011 in preparation). 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 24.2.1.)

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 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 FLASH4-alpha.

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 13.1.3.
13.3. MAGNETOHYDRODYNAMICS (MHD)

Figure 13.2: 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 \( \int \partial F_n \mathbf{E} \cdot \mathbf{T} \, dl \) 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 \( F_x, i + 1/2, j, k \)).

Table 13.9: 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>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;BAL-SARA_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 NOT available for MHD.</td>
</tr>
</tbody>
</table>
CHAPTER 13. HYDRODYNAMICS UNITS

Stability limit

As described in the unsplit hydro solver unit (physics/Hydro/HydroMain/unsplit/Hydro_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. 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/.

13.3.5 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:

```
# Magnetic Resistivity ---------
useMagneticResistivity = .true.
resistivity = 1.0E-0

# Viscosity ---------------------
useViscosity = .false.
diff_visc_nu = 1.0E-2

# Conductivity ------------------
useConductivity = .false.
diff_constant = 1.0E-2
```

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:

```
# Magnetic Resistivity
REQUIRES physics/materialProperties/MagneticResistivity/MagneticResistivityMain

# Viscosity
```
New changes in including diffusive terms

The treatments of including these non-ideal diffusive terms in the USM-MHD solver have been changed in the new FLASH4-alpha release. First, in previous releases, 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. Second, the diffusion time step calculation has been changed in that it was done internally within the USM-MHD scope previously, however, the new way of calculating diffusion time step now uses the Diffuse_computeDt.F90 routine in Driver_computeDt.F90, which provides a more consistent way of computing a non-ideal time step.
Chapter 14

Equation of State Unit

14.1 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-alpha: **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 ($\gamma$); 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.

### 14.2 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

$$
\gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial \rho}.
$$

The second relates the pressure to the energy and is given by

$$
\gamma_4 = 1 + \frac{P}{\rho \epsilon}.
$$

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 (14.2).

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

$$
P = (\gamma - 1) \rho \epsilon.
$$

We also have an expression relating pressure to the temperature $T$

$$
P = \frac{N_a k}{\bar{A}} \rho T,
$$

where $N_a$ is the Avogadro number, $k$ is the Boltzmann constant, and $\bar{A}$ is the average atomic mass, defined as

$$
\frac{1}{\bar{A}} = \sum_i X_i \frac{A_i}{A_i},
$$
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

\[
\epsilon = \frac{1}{\gamma - 1} \frac{N_a k}{A} T .
\]  

(14.6)

The relativistic variant of the ideal gas equation is explained in more detail in Section 13.2.

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

\[
\frac{1}{(\gamma - 1)} = \bar{A} \sum_i \frac{1}{(\gamma_i - 1) \bar{A}_i} .
\]  

(14.7)

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.

14.2.1 Ideal Gamma Law for Relativistic Hydrodynamics

The relativistic variant of the ideal gas equation is explained in more detail in Section 13.2.

14.3 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

\[
P_{\text{tot}} = P_{\text{rad}} + P_{\text{ion}} + P_{\text{ele}} + P_{\text{pos}} + P_{\text{coul}}
\]

(14.8)

\[
\epsilon_{\text{tot}} = \epsilon_{\text{rad}} + \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{pos}} + \epsilon_{\text{coul}} .
\]  

(14.9)

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

\[
P_{\text{rad}} = \frac{a T^4}{3}
\]

(14.10)

\[
\epsilon_{\text{rad}} = \frac{3 P_{\text{rad}} \rho}{\rho}
\]

(14.11)

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 14.3 - 14.4) 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

\[
N_{\text{ele}} = \frac{8\pi \sqrt{2}}{h^3} m_e^3 \beta^{3/2} \left[ F_{1/2}(\eta, \beta) + F_{3/2}(\eta, \beta) \right]
\]

(14.12)
\[ N_{\text{pos}} = \frac{8\pi \sqrt{2}}{h^3} m_e^3 \, c^3 \, \beta^{3/2} \left[ F_{1/2}(-\eta - 2/\beta, \beta) + \beta \, F_{3/2}(-\eta - 2/\beta, \beta) \right] , \]  

(14.13)

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

\[ F_k(\eta, \beta) = \int_0^\infty \frac{x^k \left( 1 + 0.5 \, \beta \, x \right)^{1/2}}{\exp(x - \eta) + 1} \, dx . \]  

(14.14)

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

\[ N_{\text{ele, matter}} = \frac{Z}{A} \, N_a \, \rho = \bar{Z} \, N_{\text{ion}} , \]  

(14.15)

and charge neutrality requires

\[ N_{\text{ele, matter}} = N_{\text{ele}} - N_{\text{pos}} . \]  

(14.16)

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 \( \gamma_1 \) and \( \gamma_4 \) 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,

\[ F = \epsilon - T \, S \]  

(14.17)

\[ dF = -S \, dT + \frac{P}{\rho^2} \, d\rho , \]  

(14.18)

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 (14.14) 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 (14.16). 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 \( (T, \rho \bar{Z}/\bar{A}) \) 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 \( \bar{A} = \bar{Z} = 1 \) (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 14.10 – 14.11), 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 \( \text{any} \) 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.
### 14.4 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 13.1.4, 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 14.1.

#### 14.4.1 Gamma Law and Multigamma

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 14.3). 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.

---

<table>
<thead>
<tr>
<th>MODE</th>
<th>Inputs</th>
<th>Outputs</th>
<th>1-T</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENS</td>
<td>T, ϵ, s</td>
<td>T, ϵ, s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>GATHER</td>
<td>T, ϵ, s</td>
<td>T, ϵ, s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>EI</td>
<td>T, ϵ, s</td>
<td>T, ϵ, s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>SCATTER</td>
<td>T, ϵ, s</td>
<td>T, ϵ, s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>GATHER</td>
<td>T, ϵ, s</td>
<td>T, ϵ, s</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>pres</td>
<td>P</td>
<td>T, ϵ</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.
In particular, the values of $\gamma$, $\gamma_{\text{Ion}}$, $\gamma_{\text{Ele}}$, and $\gamma_{\text{Rad}}$ 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 $\text{eos\_singleSpeciesZ}$ runtime parameter. The $\text{eos\_singleSpeciesA}$ 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 $\text{eos\_singleSpeciesA} = 12.0$ and $\text{eos\_singleSpeciesA} = 6.0$.

The combined pressure and internal energy are calculated as the sum over the components

$$P_{\text{tot}} = P_{\text{ion}} + P_{\text{ele}} + P_{\text{rad}}$$

$$\epsilon_{\text{tot}} = \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{rad}}.$$  \hspace{1cm} (14.19)

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

$$P_{\text{rad}} = \frac{aT_{\text{rad}}^4}{3}$$

$$\epsilon_{\text{rad}} = \frac{3P_{\text{rad}}}{\rho}$$

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 $\text{Eos}$ must return $\gamma = \gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial \rho}$, and $\gamma_4$ is calculated from (14.2). These two generalizations of adiabatic indices are usually stored as the mesh-based variables $\text{GAMC\_VAR}$ and $\text{GAME\_VAR}$ as a result of calling $\text{Eos\_wrapped}$.

### 14.4.1.1 Gamma/Ye

The $\text{Ye}$ variant of $\text{Gamma}$ implementation is like the $\text{Gamma}$ 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 $\text{Simulation\_initBlock}$ that initializes mass scalar variables $\text{sumy}$ and $\text{ye}$ as desired. The initial values then get advected with the motion of the fluid. $\text{Eos}$ interprets them as

$$\text{sumy} = \sum Y_i$$

and

$$\text{ye} = Y_e$$

and computes

$$\bar{A} = \bar{A} = \frac{1}{\sum Y_i}$$

and

$$\bar{Z} = \bar{Z} = \frac{\bar{A}}{Y_e}$$

from them.
14.4.2 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 $\bar{A}$ and $\bar{Z}$, 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 $\bar{A}$ and $\bar{Z}$, all else goes as described for Gamma, above.

14.5 Usage

14.5.1 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.

14.5.2 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.

### 14.5.3 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 electron pressure, degeneracy parameter, and thermodynamic derivatives can be calculated by the `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 `Eos` are provided in the API description, see `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, `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 `Eos_wrapped` routine translates the block section thus described into the vector format of the `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 `Eos` routine directly with the optional mask set to true and space allocated for the returned quantities.

### 14.6 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:

```
./setup unitTest/Eos/implementation -auto -3d
```

where `implementation` is one of `Gamma`, `Multigamma`, `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 `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 `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 x axis and gradients for temperature and pressure along the y axis. If the test is being run for the Multigamma or Helmholtz implementations, then the species are initialized to have gradients along the z axis.
Chapter 15

Local Source Terms

Figure 15.1: The organizational structure of physics source terms, which include units such as Burn and Stir. Shaded units include only stub implementations.
The physics/sourceTerms organizational directory contains several units that implement forcing terms. The Burn, Stir, Ionize, and Diffuse units contain implementations in FLASH4-alpha. Two other units, Cool and Heat, contain only stub level routines in their API.

15.1 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.

15.1.1 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.

15.1.2 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

$$X_i = \rho_i/\rho = n_i A_i/(\rho N_A) \ , \quad (15.1)$$

where $N_A$ is Avogadro’s number. Let the molar abundance of the $i$th isotope be

$$Y_i = X_i/A_i = n_i/(\rho N_A) \ . \quad (15.2)$$

Mass conservation is then expressed by

$$\sum_{i=1}^{N} X_i = 1 \ . \quad (15.3)$$

At the end of each timestep, FLASH checks that the stored abundances satisfy (15.3) 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

$$\frac{dY_i}{dt} + \nabla \cdot (Y_i \mathbf{V}_i) = \dot{R}_i \ . \quad (15.4)$$

In this equation $\dot{R}_i$ is the total reaction rate due to all binary reactions of the form $i(j,k)l$,

$$\dot{R}_i = \sum_{j,k} Y_j Y_k \lambda_{kj}(l) - Y_i Y_j \lambda_{jk}(i) \ , \quad (15.5)$$

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 (15.5). The mass diffusion velocities $\mathbf{V}_i$ in (15.4) 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 $\mathbf{V}_i \equiv 0$ 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 $\mathbf{V}_i \equiv 0$ transforms (15.4) into

$$\frac{dY_i}{dt} = \dot{R}_i,$$

which may be written in the more compact, standard form

$$\dot{y} = f(y).$$

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 (15.6) and (15.7) 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 $\mathbf{J} \equiv \partial f / \partial y$ 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 (15.6) and the associated Jacobian matrix are formed. Consider the $^{12}\text{C}(\alpha,\gamma)^{16}\text{O}$ 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 (15.7) through the terms

$$
\dot{Y}(^4\text{He}) = -Y(^4\text{He}) Y(^{12}\text{C}) R + \ldots
$$

$$
\dot{Y}(^{12}\text{C}) = -Y(^4\text{He}) Y(^{12}\text{C}) R + \ldots
$$

$$
\dot{Y}(^{16}\text{O}) = +Y(^4\text{He}) Y(^{12}\text{C}) R + \ldots,
$$

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 $\mathbf{J} = \partial f / \partial y$

$$
J(^4\text{He},^4\text{He}) = -Y(^{12}\text{C}) R + \ldots
$$

$$
J(^{12}\text{C},^4\text{He}) = -Y(^{12}\text{C}) R + \ldots
$$

$$
J(^{16}\text{O},^4\text{He}) = +Y(^{12}\text{C}) R + \ldots
$$

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, and the magnitudes of the entry changes as the abundances, temperature, or density change with time.

This release of FLASH4 contains three reaction networks. A seven-isotope alpha-chain (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 (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 (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.
15.1.2.1 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.

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 \texttt{algebra} controls which linear algebra package is used in the simulation. \texttt{algebra = 1} is the default choice and invokes the sparse matrix MA28 package. \texttt{algebra = 2} invokes the GIFT linear algebra routines.

15.1.2.2 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
where the four vectors $\Delta^i$ are found from successively solving the four matrix equations

\begin{align}
(\bar{1}/\gamma h - \tilde{J}) \cdot \Delta_1 &= f(y^n) \\
(\bar{1}/\gamma h - \tilde{J}) \cdot \Delta_2 &= f(y^n + a_{21} \Delta_1) + c_{21} \Delta_1/h \\
(\bar{1}/\gamma h - \tilde{J}) \cdot \Delta_3 &= f(y^n + a_{31} \Delta_1 + a_{32} \Delta_2) + (c_{31} \Delta_1 + c_{32} \Delta_2)/h \\
(\bar{1}^* h - \tilde{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 .
\end{align}

\( b_i, \gamma, a_{ij}, \text{ and } c_{ij} \) 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}
\Delta_0 &= h f(y^n) \\
y_1 &= y^n + \Delta_0 .
\end{align}

Then from $k = 1, 2, \ldots, m - 1$

\begin{align}
(\bar{1} - \tilde{J}) \cdot x &= h f(y_k) - \Delta_{k-1} \\
\Delta_k &= \Delta_{k-1} + 2x \\
y_{k+1} &= y_k + \Delta_k ,
\end{align}

and closure is obtained by the last stage

\begin{align}
(\bar{1} - \tilde{J}) \cdot \Delta_m &= h [f(y_m) - \Delta_{m-1}] \\
y^{n+1} &= y_m + \Delta_m .
\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 15.15 – 15.17, 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 `odeStepper` controls which integration method is used in the simulation. The
choice `odeStepper = 1` is the default and invokes the variable order Bader-Deuflhard scheme. The choice `odeStepper = 2` invokes the fourth order Kaps-Rentrop / Rosenbrock scheme.

### 15.1.3 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 `useShockBurn` parameter is set to `.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 (i.e. `useShockBurning = .false.`).

### 15.1.4 Energy generation rates and reaction rates

The instantaneous energy generation rate is given by the sum

\[
\dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{dY_i}{dt}.
\]  

(15.18)

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

\[
\dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{\Delta Y_i}{\Delta t}.
\]  

(15.19)

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 `useBurnTable` controls which reaction rate evaluation method is used in the simulation. The choice `useBurnTable = 0` is the default and evaluates the reaction rates from analytical expressions. The choice `useBurnTable = 1` evaluates the reaction 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 $\epsilon_{\text{nuc}}$ due to neutrino losses as given by Itoh *et al.* (1996) are included in FLASH.

### 15.1.5 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} (15.20)

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.

### 15.2 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} (15.21)

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P = \rho \mathbf{g}$$  \hspace{1cm} (15.22)

$$\frac{\partial (\rho E)}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} \left[ + S \right]$$  \hspace{1cm} (15.23)

$$\frac{\partial n^Z_i}{\partial t} + \nabla \cdot n^Z_i \mathbf{v} = R^Z_i \left( i = 1, \ldots, N_{\text{spec}} \right),$$  \hspace{1cm} (15.24)

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, $n^Z_i$ is the number density of the ion $i$ of the element $Z$, $N_{\text{spec}}$ is the total number of species, and

$$R^Z_i = N_e \left[ n^Z_{i+1} \alpha^Z_{i+1} + n^Z_{i-1} S^Z_{i-1} - n^Z_i \left( \alpha^Z_i + S^Z_i \right) \right],$$  \hspace{1cm} (15.25)

where $N_e$ is the electron number density. $\alpha^Z_i \equiv \alpha(N_e, T)$ are the collisional and dielectronic recombination coefficients, and $S^Z_i \equiv S(N_e, T)$ are the collisional ionization coefficients of Summers(1974).
Table 15.1: 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</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>

15.2.1 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 (15.21) 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

$$\frac{\partial n_i^Z}{\partial t} = R_i^Z \quad (i = 1, \ldots, N_{spec})$$

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 (15.23)) 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.

15.2.2 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.1e8.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 15.1. There are two implementations of `physics/sourceTerms/Ionize/IonizeMain`: the default implementation, Nei (tested using Neitest (see Section 24.7.1)), and Eqi (untested in FLASH4-alpha). 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.

15.3 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 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. 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 \textit{st_energy}) divided by the decay time \( \tau \) (\textit{st_decay}). In the limit that the timestep \( \Delta t \rightarrow 0 \), 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. The runtime parameters associated with the \texttt{Stir} unit are described in the online [[rpi reference]] page.

15.4 Energy Deposition Unit

The Energy Deposition unit calculates the energy deposited by one or more laser beams incident upon the simulation domain. 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 indices. The laser power deposited in a zone is calculated based on the inverse Bremsstrahlung power in the zone and depends on the local electron number density gradient and local electron temperature gradient. The laser energy deposition implementation in flash follows closely that described by Kaiser (2000).

15.4.1 Ray Tracing in the Geometric Optics Limit

In the geometric optics approach, the path of a ray can be described as the motion of a unit-mass particle through the potential field \( V = (c^2/2)(n_e/n_c) \), where \( c \) is the speed of light in vacuum, \( n_e \) is the electron number density, and \( n_c \) is the critical density at the wavelength of the ray. The ray equation of motion is then

\[
\frac{d^2 \vec{x}}{dt^2} = \vec{\nabla} \left( -\frac{c^2 n_e}{2 n_c} \right).
\]

The ray tracing algorithm employed by the Energy Deposition unit assumes that the gradients of the electron number density and electron temperature are constant within a zone. The electron number density at any point within a zone may be calculated via

\[
n_e(x) = \langle n_e \rangle - \langle \nabla n_e \rangle \cdot (x - \langle x \rangle).
\]

This equation introduces an error in \( n_e(x) \sim O(e^2) \).

Integration of the ray equation of motion leads to equations for the ray velocity and position and time \( t_0 + \Delta t \),

\[
\vec{v}(\Delta t) = \vec{v}_0 - \frac{c^2}{2n_c} (\nabla n_e)(\Delta t),
\]

\[
\vec{x}(\Delta t) = \vec{x}_0 + \vec{v}_0 \Delta t - \frac{c^2}{4n_c} (\nabla n_e)(\Delta t)^2,
\]

where \( \vec{x}_0 \) and \( \vec{v}_0 \) are the ray position and velocity upon entry into the zone under consideration. The ray exit point from a zone is determined by solving equation 15.30 for the minimum zone exit time of the ray. The zone face through which the ray passes upon exit determines the next zone the ray enters.

The ray will experience a discontinuous change in \( n_e \) at zone interfaces. To account for this, the ray undergoes refraction according to Snell’s law. Refraction at zone interfaces causes a changes in the ray’s
velocity normal to the interface surface while preserving the ray velocity component transverse to the interface normal. The change in the ray normal velocity component due to Snell’s law is

\[
\Delta v_\perp = -\Delta \left( \frac{n_e}{n_c} c^2 \right).
\]  

(15.31)

If \( \Delta v_\perp^2 < 0 \), the ray reflects at the interface.

The current implementation of laser energy deposition assumes that rays transit the entire domain in a single time step. The Energy Deposition unit moves the rays across all blocks until every ray has reached the edge of its current block. The Energy Deposition unit then utilizes the infrastructure in the GridParticles unit to move rays between blocks and processors and this process is then repeated.

### 15.4.2 Laser Power Deposition

The energy in the transiting rays is deposited via inverse Bremsstrahlung (IB) radiation. The IB rate is

\[
\nu_{ei} = \frac{n_e}{n_c} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_e Z e^4 \ln \Lambda}{T_e^{3/2}}.
\]

(15.32)

where \( n_e \) is the electron mass, \( Z \) is the charge of the plasma, \( e \) is the electron charge, \( \ln \Lambda \) is the Coulomb logarithm, and \( T_e \) is the electron temperature. The ray’s power is attenuated as it moves through a zone according to

\[
P(\Delta t) = P(0) \exp \left\{ -\int_0^{\Delta t} \nu_{ib}[\vec{x}(t)] \, dt \right\},
\]

(15.33)

where \( P(0) \) is the ray’s power as it enters the zone under consideration. The evaluation of the integral in equation 15.33 employs second order Gaussian quadrature and follows closely the method described in Kaiser (2000). The rate of energy deposition in a zone is then \( P(0) - P(\Delta t) \).

### 15.5 Usage

The Energy Deposition unit reads information about the laser beams from a file whose name can be specified at runtime with the parameter ed_inputFileName. The format of this file is explained in the example file “laser.beams” in the EnergyDepositionMain/Laser/ directory. In this file, the user specifies the number of laser beams, and for each beam the domain face on which the beam is incident, the coordinates of the beam within the incident face, the radius of the beam spot on the incident face, the three-velocity of the beam (normalized to 1), the beam irradiance, laser frequency, and the number of rays in the beam. The current implementation of EnergyDeposition assumes circular beams with a constant irradiance profile across the beam and that all rays enter the domain parallel with the beam incidence direction (specified by the beam three-velocity). These assumptions can be easily relaxed.

The runtime parameters associated with the EnergyDeposition unit are the following. The use of EnergyDeposition is toggled with useEnergyDeposition. The EnergyDeposition can run in “ray-trace only” mode by setting the parameter ed_doDeposition to false. The maximum number of rays per processor is specified with ed_maxRayCount.

### 15.6 Heatexchange

The heatexchange unit implements coupling among the different components (Ion, Electron, Radiation) as a simple relaxation law.

\[
\frac{\partial \rho E_{Ele}}{\partial t} = \kappa_{i,e}(T_{Ion} - T_{Ele}) + \kappa_{e,r}(T_{Rad} - T_{Ele})
\]

(15.34)

\[
\frac{\partial E_{Rad}}{\partial t} = -\kappa_{e,r}(T_{Rad} - T_{Ele})
\]

(15.35)

\[
\frac{\partial \rho E_{Ion}}{\partial t} = -\kappa_{i,e}(T_{Ion} - T_{Ele})
\]

(15.36)
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.

`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 15.3: 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.
Chapter 16

Diffusive Terms

The physics/Diffuse unit implements diffusive effects, such as heat conduction, viscosity, and mass diffusivity.

16.1 Diffuse Unit

The Diffuse code unit implements diffusive effects. Currently supported effects are heat conduction and viscosity. Support for mass diffusivity (species mixing) is not provided in the FLASH4-alpha release but can be added by users along the same lines. We provide an add-on package for this as a starting point. See http://flash.uchicago.edu/website/codesupport/ for more information.

A simulation that models diffusive effects (whether using the Diffuse unit or by some other means) should use the appropriate materialProperties code units for defining corresponding material properties that control the size of the diffusive effects. Here is an overview over the major Diffuse routines and their purposes:

<table>
<thead>
<tr>
<th>Physical effect</th>
<th>Usage</th>
<th>Public Diffuse interface</th>
<th>associated materialProperties unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat conduction</td>
<td>standalone</td>
<td>Diffuse</td>
<td>Conductivity</td>
</tr>
<tr>
<td>Heat conduction</td>
<td>flux-based</td>
<td>Diffuse_therm</td>
<td>Conductivity</td>
</tr>
<tr>
<td>Viscosity</td>
<td>flux-based</td>
<td>Diffuse_visc</td>
<td>Viscosity</td>
</tr>
<tr>
<td>Species mixing</td>
<td>flux-based</td>
<td>Diffuse_species</td>
<td>MassDiffusivity</td>
</tr>
</tbody>
</table>
16.1.1 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-alpha, 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 FLASH4-alpha simulation, the runtime parameters useDiffuse and useDiffuseTherm must be set to .true.; and to use the Diffuse unit for viscosity effects in a FLASH4-alpha simulation, the runtime parameters useDiffuse and useDiffuseVisc must be set to .true..

16.1.2 Diffuse Split/Unsplit implementation

The Diffuse interface in DiffuseMain provides a “standalone” way for invoking diffusive effects: it is called from Driver.evolveFlash, and does not require a co-operating Hydro implementation.
In FLASH4-alpha the Diffuse unit is used to solve thermal conduction in single temperature mode and electron conduction, multigroup diffusion in multi temperature mode. Each of the above physical operators is advanced implicitly. The need for implicit time advancement arises naturally in radiation transport as the time scales are much smaller compared to the hydrodynamic time scales.

The entire Diffusion operator is spread across two units in FLASH4-alpha. The actual physics related aspects such as boundary conditions, *dt* computation etc resides under Diffuse and the linear solver (HYPRE for unsplit and Thomas direct solve for split) resides under GridSolvers.

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 typical diffusion equation that can be solved by this unit takes the form,

\[ A \frac{\partial f}{\partial t} + Cf = \nabla \cdot B \nabla f + D, \tag{16.1} \]

where, 
\( f \) is the variable to be diffused in time,
\( A, B, C, D \) are parameters with spatial variation.

While solving the thermal conduction problem the above equation simplifies to,

\[ \rho C_v \frac{\partial T}{\partial t} = \nabla \cdot \kappa \nabla T, \tag{16.2} \]

where \( A \) is replaced by \( \rho C_v \) and \( B \) by \( \kappa \). \( C \) and \( D \) are not used and set to \(-1\).

Similarly for a simple radiation diffusion problem, the above equation takes the form,

\[ \frac{\partial E}{\partial t} + c\sigma_a E = \nabla \cdot c \frac{c}{3\sigma_t} \nabla E + c\sigma_a T^4 \tag{16.3} \]

The infrastructure is provided for a general diffusion equation and can be customized to solve specific equations. The API function \texttt{Diffuse\_solveScalar} provides a comprehensive interface to solve different types of equations.

In the unsplit solver the equation is implicitly advanced as follows,

\[ A^n \frac{f^{n+1} - f^n}{\delta t} + C^n f^{n+1} = \theta(\nabla \cdot B^n \nabla f^{n+1}) + (1 - \theta)(\nabla \cdot B^n \nabla f^n) + D, \tag{16.4} \]

In the split solver, the solution is advanced along each direction separately. i.e,

\[ A^n \frac{f^{n+\frac{1}{2}} - f^n}{\delta t} + \frac{1}{3} C^n f^{n+\frac{1}{2}} = \theta \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^{n+\frac{1}{2}}) + (1 - \theta) \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^n) + \frac{1}{3} D \tag{16.5} \]

\[ A^n \frac{f^{n+\frac{1}{2}} - f^{n+\frac{1}{2}}}{\delta t} + \frac{1}{3} C^n f^{n+\frac{1}{2}} = \theta \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{1}{2}}) + (1 - \theta) \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{1}{2}}) + \frac{1}{3} D \tag{16.6} \]

\[ A^n \frac{f^{n+1} - f^{n+\frac{1}{2}}}{\delta t} + \frac{1}{3} C^n f^{n+1} = \theta \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+1}) + (1 - \theta) \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+\frac{1}{2}}) + \frac{1}{3} D \tag{16.7} \]

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>Scheme</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,

\[
\nabla \cdot B \nabla f = \frac{B_{i+\frac{1}{2},j,k}(f_{i+1,j,k} - f_{i,j,k}) - B_{i-\frac{1}{2},j,k}(f_{i,j,k} - f_{i-1,j,k})}{\delta x^2} \\
+ \frac{B_{i,j+\frac{1}{2},k}(f_{i,j+1,k} - f_{i,j,k}) - B_{i,j-\frac{1}{2},k}(f_{i,j,k} - f_{i,j-1,k})}{\delta y^2} \\
+ \frac{B_{i,j,k+\frac{1}{2}}(f_{i,j,k+1} - f_{i,j,k}) - B_{i,j,k-\frac{1}{2}}(f_{i,j,k} - f_{i,j,k-\frac{1}{2}})}{\delta z^2}
\]

(16.8)

Where, $f$ and $B$ are cell centered variables, and

\[
B_{i+\frac{1}{2},j,k} = \frac{B_{i+1,j,k} + B_{i,j,k}}{2}, \\
B_{i-\frac{1}{2},j,k} = \frac{B_{i-1,j,k} + B_{i,j,k}}{2}, \\
B_{i,j+\frac{1}{2},k} = \frac{B_{i,j+1,k} + B_{i,j,k}}{2}, \\
B_{i,j-\frac{1}{2},k} = \frac{B_{i,j-1,k} + B_{i,j,k}}{2}, \\
B_{i,j,k+\frac{1}{2}} = \frac{B_{i,j,k+1} + B_{i,j,k}}{2}, \\
B_{i,j,k-\frac{1}{2}} = \frac{B_{i,j,k-1} + B_{i,j,k}}{2}
\]

(16.9) – (16.14)

**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_consvervefluxes` 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 conversation 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,

\[
A \frac{\partial f}{\partial t} = \nabla \cdot B \nabla f,
\]

(16.15)

We introduce a new variable $\vec{F}$ defined as,

\[
\vec{F} = B \nabla f, and
\]

(16.16)

We discretize the equation as follows,

\[
A \frac{\partial F}{\partial t} = \nabla \cdot \vec{F},
\]

(16.17)

We then 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,

\[
A^n \frac{f^{n+1} - f^n}{\delta t} \partial s = \theta(\vec{F}^{n+1} \cdot \partial s) + (1 - \theta)(\vec{F}^n \cdot \partial s)
\]

(16.18)
\[
(F^L)_{i+\frac{1}{2}} = (F^R)_{i+\frac{1}{2}} = B_i + \frac{1}{2} \frac{f_{i+1} - f_i}{\delta x_{i+\frac{1}{2}}}, \quad (16.20)
\]
\[
(F^L)_{i-\frac{1}{2}} = (F^R)_{i-\frac{1}{2}} = B_i - \frac{1}{2} \frac{f_i - f_{i-1}}{\delta x_{i-\frac{1}{2}}}, \quad (16.21)
\]
\[
(16.22)
\]

The idea is similarly extended to 2D where coarse cell flux is set to sum of computed fine cell fluxes of its neighbors.

\[
F_c ds_c = F_1 ds_1 + F_2 ds_2 \quad (16.23)
\]

Where,

\[
F_1 = B_{face} \frac{f_1 - f_c}{\delta x_{1+\frac{1}{2}}}, \quad (16.24)
\]
\[
F_2 = B_{face} \frac{f_2 - f_c}{\delta x_{2+\frac{1}{2}}}, \quad (16.25)
\]

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>
<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>
</tbody>
</table>

Table 16.1: comparison of capabilities of Split and Unsplit diffusion.
There are two switches to the diffusion mode, the run time parameter `useDiffuse` is to be used in normal mode and `diff_useEleCond` is to be used in multiTemp mode.

```
useDiffuse = .true.
diff_useEleCond = .true.
```

### Table 16.2: Support for boundary conditions

<table>
<thead>
<tr>
<th>Boundary Type</th>
<th>Split</th>
<th>Unsplit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Outflow/Zero Flux</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Vacuum</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Dirichlet (homogeneous)</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

The following are run time parameters to set boundary conditions,

```
diff_eleXlBoundaryType = "outflow"
diff_eleXrBoundaryType = "outflow"
diff_eleYlBoundaryType = "outflow"
diff_eleYrBoundaryType = "outflow"
diff_eleZlBoundaryType = "outflow"
diff_eleZrBoundaryType = "outflow"
```

**Note:**

- It uses an implicit scheme, can use larger time steps compared to flux based scheme.
- It is applied as a separate operator and is not a part of Hydro, called from `Driver.evolveFlash`
- It has its own run time parameters for boundary conditions and does not use the grid boundaries.

**Mesh Replication:** In Multigroup diffusion the diffusion operation is performed on multiple groups and hence can quite easily become a bottle neck. An efficient alternate is to replicate the mesh across processor groups.

**Operability:** The Split Hydro code has been tested with diffusion using Rad-Hydro problems and is known to work. The Unsplit Hydro has been run with diffusion code but not extensively tested. The diffusion solver has been tested on intel, absoft, nag and gnu compilers and is known to have no issues.

**Simulations:** A number of simulations are available in FLASH4-alpha which use this unit. MGDStep, MGDInfinite and Shafranov use the solver in multiTemp mode and ConductionDeltaSaDiff uses this solver in single temperature mode.
Chapter 17

Gravity Unit

17.1 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 $\mathbf{g}(\mathbf{x})$,

$$\mathbf{g}(\mathbf{x}) = -\nabla \phi(\mathbf{x}) .$$  \hfill (17.1)

The gravitational field can be externally imposed or self-consistently computed from the gas density via the Poisson equation,

$$\nabla^2 \phi(\mathbf{x}) = 4\pi G \rho(\mathbf{x}) ,$$  \hfill (17.2)
where $G$ is Newton’s gravitational constant. In the latter case, either periodic or isolated boundary conditions can be applied.

### 17.2 Externally Applied Fields

The FLASH distribution includes three externally applied gravitational fields. Each provides the acceleration vector $\mathbf{g}(\mathbf{x})$ directly, without using the gravitational potential $\phi(\mathbf{x})$.

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.

#### 17.2.1 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`.

#### 17.2.2 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.

#### 17.2.3 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.

### 17.3 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 (17.2). This unit’s implementation can also return the acceleration field $\mathbf{g}(\mathbf{x})$ computed by finite-differencing the potential using the expressions

$$
g_{x;ijk} = \frac{1}{2\Delta x} \left( \phi_{i-1,j,k} - \phi_{i+1,j,k} \right) + \mathcal{O}(\Delta x^2)
$$

$$
g_{y;ijk} = \frac{1}{2\Delta y} \left( \phi_{i,j-1,k} - \phi_{i,j+1,k} \right) + \mathcal{O}(\Delta y^2)
$$

$$
g_{z;ijk} = \frac{1}{2\Delta z} \left( \phi_{i,j,k-1} - \phi_{i,j,k+1} \right) + \mathcal{O}(\Delta z^2).
$$

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 8.10.2.3 is appropriate for general mass distributions and can solve problems with more general boundary conditions.

### 17.3.1 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

$$\rho E \equiv \frac{1}{2} \rho v^2 + \rho \epsilon,$$  \hspace{1cm} (17.4)

where $\epsilon$ is the specific internal energy, then the gravitational source terms for the momentum and energy equations are $\rho g$ and $\rho v \cdot g$, 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 $g$, 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 $v$ and $g$.

For example, the PPM algorithm supplied with FLASH uses the following update steps to obtain the momentum and energy in cell $i$ at timestep $n + 1$

$$\begin{align*}
(pv)^n_{i+1} & = (pv)^n_i + \frac{\Delta t}{2} g_i^n + (\rho_i^n + \rho_i^{n+1}) \\
(pE)^n_{i+1} & = (pE)^n_i + \frac{\Delta t}{4} g_i^n + (\rho_i^n + \rho_i^{n+1}) (v_i^n + v_i^{n+1}) .
\end{align*}$$  \hspace{1cm} (17.5)

Here $g_i^n$ 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, $g$ 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 $g$ 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

$$\begin{align*}
v_{L,i+1/2} & \rightarrow v_{L,i+1/2} + \frac{\Delta t}{4} \left(g_{L,i+1/2}^+ + g_{L,i+1/2}^-\right) \\
v_{R,i+1/2} & \rightarrow v_{R,i+1/2} + \frac{\Delta t}{4} \left(g_{R,i+1/2}^+ + g_{R,i+1/2}^-\right). \hspace{1cm} (17.6)
\end{align*}$$

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.

### 17.4 Usage

To include the effects of gravity in your FLASH executable, include the option

```bash
-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 Section 17.2 explicitly compute the potential, hence they inherit the null implementation from the API for accessing potential. The gravitation 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 Poisson 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.

17.5 Unit Tests

There are two unit tests for the gravity unit. Poisson3 is essentially the Maclaurin spheroid problem described in Section 24.3.4. 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 24.4.3. PoisParticles is based on the Huang-Greengard Poisson gravity test described in Section 24.3.3.
Chapter 18

Particles Unit

Figure 18.1: The Particles unit main subunit.
Figure 18.2: The *Particles* unit with ParticlesInitialization and ParticlesMapping subunits.

The support for particles in FLASH4-alpha 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.

$$\frac{dx_i}{dt} = v_i \quad (18.1)$$

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

$$v_{i}^{n+1} = v_{i}^{n} + a_{i}^{n} \Delta t^{n} \quad (18.2)$$

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 (18.1) and another describing the effects of forces.

$$m_i \frac{dv_i}{dt} = F_{lr,i} + F_{sr,i} \quad (18.3)$$

Here, $F_{lr,i}$ represents the sum of all long-range forces (coupling all particles) 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 (18.1) forward through time. Many alternative integration methods are described in Section 18.1 below. Additional information about the mesh to particle mapping is described in Section 18.2. 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).
18.1 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-alpha has support for including both active and passive particles in a single simulation, `Particles_advance` calls helper routines `pt_advanceActive` and `pt_advancePassive`. Only one type of passive and one type of active time integration scheme can be selected for any simulation, no matter how many types of active particles exist. In all implementations, particle velocities are obtained by mapping grid-based velocities onto particle positions as described in Section 18.2.

18.1.1 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 18.1 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/Leapfrog`, or by specifying something like `REQUIRES Particles/ParticlesMain/active/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 18.3 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:

\[
\begin{align*}
x_i^{n+1} &= x_i^n + v_i^n \Delta t^n \\
v_i^{n+1} &= v_i^n + a_i^n \Delta t^n.
\end{align*}
\]

(18.4)

Here \( a_i \) is the particle acceleration. Within FLASH4-alpha, this scheme is called `Particles/ParticlesMain/- active/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

\[
\begin{align*}
x_i^1 &= x_i^0 + v_i^0 \Delta t^0 \\
v_i^{1/2} &= v_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
v_i^{n+1/2} &= v_i^{n-1/2} + C_n a_i^n + D_n a_i^{n-1} \\
x_i^{n+1} &= x_i^n + v_i^{n+1/2} \Delta t^n.
\end{align*}
\]

(18.5)
The coefficients $C_n$ and $D_n$ are given by

$$C_n = \frac{1}{2} \Delta t^n + \frac{1}{3} \Delta t^{n-1} + \frac{1}{6} \left( \frac{\Delta t^n}{\Delta t^{n-1}} \right)$$

$$D_n = \frac{1}{6} \left( \Delta t^{n-1} - \frac{\Delta t^n}{\Delta t^{n-1}} \right).$$  (18.6)

By using time-centered velocities and stored accelerations, this method achieves second-order time accuracy even with variable timesteps. Within FLASH4-alpha, this scheme is called `Particles/ParticlesMain/active/Leapfrog`.

- **Cosmological variable-timestep leapfrog.** (`Particles/ParticlesMain/active/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 ($v = \dot{x}$). The resulting update steps are

  \[
  x_i^1 = x_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
  v_i^{1/2} = v_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
  v_i^{n+1/2} = v_i^{n-1/2} \left[ 1 - \frac{A^n}{2} \Delta t^n + \frac{1}{3!} \Delta t^{n-1} \left( A^{n-1} - \dot{A} \right) \right] \left[ 1 - \Delta t^{n-1} A^n + \Delta t^{n-2} A^{n^2} + 2 \dot{A} A^n \right] \\
  + a_i^n \left[ \frac{\Delta t^{n-1}}{2} + \frac{\Delta t^n}{6} \Delta t^{n-1} \right] \\
  + a_i^{n-1} \left[ \frac{\Delta t^{n-1} - \Delta t^n}{6} \right] \\
  + \frac{\Delta t^{n-1} - \Delta t^n}{12} (\Delta t^n + \Delta t^{n-1}) \\
  x_i^{n+1} = x_i^n + v_i^{n+1/2} \Delta t^n.\
  \]

  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 `Particles/ParticlesMain/passive/LeapfrogCosmo` time integration scheme only makes sense if the Cosmology module is also used, since otherwise $a \equiv 1$ and $\dot{a} \equiv 0$.

The leapfrog-based integrators supply the additional particle attributes listed in Table 18.2.

<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>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>-----------------------------------------------</td>
</tr>
</tbody>
</table>

Table 18.1: Particle attributes provided by active particles.
18.1.2 Charged Particles - Hybrid PIC

Modeling of collisionless plasmas are often done using fluid magnetohydrodynamic models (MHD). The MHD fluid approximation is however questionable when the gyro radius of the ions are 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 very computational expensive. For problems where the ion time- and spatial scales 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 mean that the electron time- and spatial scales 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).

18.1.2.1 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 $r_i(t)$ [m] with velocities $v_i(t)$ [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 $\rho_I(r,t) \text{[Cm}^{-3}\text{]}$ of the ions, their average velocity $u_I(r,t)$ [m/s], and the corresponding current density $J_I(r,t) = \rho_I u_I \text{[Cm}^{-2}\text{s}^{-1}\text{]}$.

Electrons are modelled as a fluid with charge density $\rho_e(r,t)$, average velocity $u_e(r,t)$, and current density $J_e(r,t) = \rho_e u_e$. 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 $p_e = n_e k T_e$, 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 $\mathbf{E} = \mathbf{E}(r,t)$ is the electric field, and $\mathbf{B} = \mathbf{B}(r,t)$ 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 denote reference values (here the initial values at $t = 0$). Then Faraday’s law is used to advance the magnetic field in time,

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}.$$
18.1.2.2 A cell centered finite difference hybrid 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 includes 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.

18.1.2.3 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 straightforward. 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 specie, named $pt\_pic\_Pname\_1$ consists of particles of mass $pt\_pic\_Pmass\_1$ and charge $pt\_pic\_Pcharge\_1$. The initial (at $t = 0$) uniform number density is $pt\_pic\_Pdensity\_1$, and the velocity distribution is Maxwellian with a drift velocity of $(pt\_pic\_Pvelx\_1, pt\_pic\_Pvely\_1, pt\_pic\_Pvelz\_1)$, and a temperature of $pt\_pic\_Ptemp\_1$. Each model macro-particle represents $pt\_pic\_Pweight\_1$ real particles. So this parameter will control the total number of macro-particles of specie 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 specie, e.g., $pt\_pic\_Pmass\_2$, which is initialized in the same way as the first specie. 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 ($gbx1$, $gby1$, $gbz1$). We then solve for the deviation from this background field. Care must be taken 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 ($grex$, $grey$, $grez$), the current density in ($grjx$, $grjy$, $grjz$), 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 specie 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, $\Delta t = \text{dtinit} = \text{dtmin} = \text{dtmax}$, since the leap frog solver requires this. For the solution to be stable the time step, $\Delta t$, must be small enough. We
18.1. TIME INTEGRATION

Table 18.3: Runtime parameters for the hybrid solver. Initial values are at $t = 0$. For each parameter for specie 1, there is a corresponding parameter for specie 2 (named with 2 instead of 1), e.g., $pt_{\text{picPvelx}}_2$.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pt_{\text{picPname}}_1$</td>
<td>STRING</td>
<td>&quot;H+&quot;</td>
<td>Specie 1 name</td>
</tr>
<tr>
<td>$pt_{\text{picPmass}}_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Specie 1 mass, $m_i$ [amu]</td>
</tr>
<tr>
<td>$pt_{\text{picPcharge}}_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Specie 1 charge, $q_i$ [e]</td>
</tr>
<tr>
<td>$pt_{\text{picPdensity}}_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Initial $n_i$ specie 1 [$m^{-3}$]</td>
</tr>
<tr>
<td>$pt_{\text{picPtemp}}_1$</td>
<td>REAL</td>
<td>1.5e5</td>
<td>Initial $T_i$ specie 1 [K]</td>
</tr>
<tr>
<td>$pt_{\text{picPvelx}}_1$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $u_i$ specie 1 [m/s]</td>
</tr>
<tr>
<td>$pt_{\text{picPvely}}_1$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$pt_{\text{picPvelz}}_1$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$pt_{\text{picPweight}}_1$</td>
<td>REAL</td>
<td>1.0</td>
<td>Real particles per macro-particle of specie 1</td>
</tr>
<tr>
<td>$sim_{\text{bx}}$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $B$ (at $t = 0$) [T]</td>
</tr>
<tr>
<td>$sim_{\text{by}}$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$sim_{\text{bz}}$</td>
<td>REAL</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$sim_{\text{te}}$</td>
<td>REAL</td>
<td>0.0</td>
<td>Initial $T_e$ [K]</td>
</tr>
<tr>
<td>$sim_{\text{gam}}$</td>
<td>REAL</td>
<td>-1.0</td>
<td>Adiabatic exponent for electrons</td>
</tr>
<tr>
<td>$sim_{\text{nsub}}$</td>
<td>INTEGER</td>
<td>3</td>
<td>Number of CL B-field update subcycles (must be odd)</td>
</tr>
<tr>
<td>$sim_{\text{rng_seed}}$</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 specie 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. $\Delta t \frac{\max_{i}(|v_i|)}{\Delta x} < 1$. This time step is printed to standard output by FLASH (as $dt_{\text{Part}}$), 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 sub-cycling 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 $n_1/B$, e.g., if $n_1$ 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.

18.1.3 Passive Particles

Passive particles may be moved using one of several different methods available in FLASH4-alpha. 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.

Numerically solving Equation (18.1) 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 18.4: The grid variables for the hybrid solver that are most important for a user. For each property for specie 1, there is a corresponding variable for specie 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>cdenv</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, $\mathbf{B}$ [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, $\mathbf{E}$ [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, $\mathbf{J}$ [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, $\mathbf{J}_I$ [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>Specie 1 charge density, $\rho_I$ [C/m$^3$]</td>
</tr>
<tr>
<td>jix1</td>
<td>PER_VOLUME</td>
<td>Specie 1 ion current density, $\mathbf{J}_I$ [A/m$^2$]</td>
</tr>
<tr>
<td>ji1</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 $\mathbf{v}_i$ could be performed at arbitrary times (and with unlimited accuracy). Note that FLASH4-alpha 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 Particles_advance is called, the fluid fields have already been updated to $t^{n+1}$, as is the case with the Driver_evolveFlash implementations provided with FLASH4-alpha. A specific implementation of the passive portion of Particles_advance is selected by a setup option such as `\-with-unit=Particles/-ParticlesMain/passive/Euler`, or by specifying something like `REQUIRES Particle/ParticlesMain/passive/Euler` 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 18.3 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:

\[
x_i^{n+1} = x_i^n + \mathbf{v}_i^n \Delta t^n .
\]  

Here $\mathbf{v}_i^n$ 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 $\mathbf{v}_i^n$ 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 Particles_computeDt, one does not in general wish to let particles integration dictate the time step of the simulation.
18.1. TIME INTEGRATION

Table 18.5: 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(r_i) ) [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(r_i) ) [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-alpha. 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:

\[
x_i^{n+1} = x_i^n + \frac{\Delta t^n}{2} \left[ v_i^n + v_i^{*,n+1} \right], \quad (18.8)
\]

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 \(^2\); 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

\[
x_i^{n+1} = x_i^{n-1} + v_i^n (\Delta t^{n-1} + \Delta t^n). \quad (18.9)
\]

The scheme is second order if \( \Delta t^n = \Delta t^{n-1} \).

To get the scheme started, a Euler step (as described for active/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

\[ x_{i}^{n+1} = x_{i}^{n} + \Delta t^{n} v_{i}^{\text{comb}}, \tag{18.10} \]

where

\[ v_{i}^{\text{comb}} = c_{1} v(x_{i}^{n}, t_{n}) + c_{2} v(x_{i}^{n+\frac{1}{2}}, t_{n}) + c_{3} v(x_{i}^{n}, t_{n+1}) + c_{4} v(x_{i}^{n}, t_{n+1}) \tag{18.11} \]

is a combination of four evaluations (two each at the previous and the current time),

\[ x_{i}^{n+\frac{1}{2}} = x_{i}^{n} + \frac{1}{2} \Delta t^{n-1} v_{i}^{n} \]

are estimated midpoint positions as before in the Estimated Midpoint scheme, and the coefficients

\[
\begin{align*}
    c_{1} &= c_{1}(\Delta t^{n-1}, \Delta t^{n}) , \\
    c_{2} &= c_{2}(\Delta t^{n-1}, \Delta t^{n}) , \\
    c_{3} &= c_{3}(\Delta t^{n-1}, \Delta t^{n}) , \\
    c_{4} &= c_{4}(\Delta t^{n-1}, \Delta t^{n})
\end{align*}
\]

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 \( \Delta t^{*} = \frac{1}{2} \Delta t^{n-1} \) the estimated half time step used in the scheme, let \( t_{*}^{n+\frac{1}{2}} = t_{n} + \Delta t^{*} \) 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 \( x_{i}^{E,n+\frac{1}{2}} = x_{i}^{n} + \frac{1}{2} \Delta t^{n} v_{i}^{n} \) for first-order (Euler) approximate positions at the actual midpoint time \( t^{n+\frac{1}{2}} \), and we continue to denote with \( x_{i}^{*,n+\frac{1}{2}} \) the estimated positions reached at the estimated midpoint time \( t_{*}^{n+\frac{1}{2}} \).

Assuming reasonably smooth functions \( v(x, t) \), we can then write for the exact value of the velocity field at the approximate positions evaluated at the actual midpoint time

\[ v(x_{i}^{E,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) = v(x_{i}^{n}, t_{n}) + v_{t}(x_{i}^{n}, t_{n}) \frac{1}{2} \Delta t^{n} + (v_{i}^{n} \cdot \frac{\partial}{\partial x}) v(x_{i}^{n}, t_{n}) \frac{1}{2} \Delta t^{n} + O((\frac{1}{2} \Delta t^{n+1})^{2}) \tag{18.12} \]

by Taylor expansion. It is known that the propagation scheme \( x_{i}^{n+1} = x_{i}^{n} + v(x_{i}^{E,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) \Delta t \) using these velocities is second order (this is known as the modified Euler method).

On the other hand, expansion of (18.11) gives

\[
\begin{align*}
    v_{i}^{\text{comb}} &= (c_{1} + c_{2} + c_{3} + c_{4}) v(x_{i}^{n}, t_{n}) \\
    &+ (c_{2} + c_{4}) v_{t}(x_{i}^{n}, t_{n}) \Delta t + (c_{1} + c_{2}) (v_{i}^{n} \cdot \frac{\partial}{\partial x}) v(x_{i}^{n}, t_{n}) \Delta t^{n} \\
    &+ \text{higher order terms in } \Delta t \text{ and } \Delta t^{n}.
\end{align*}
\]

After introducing a time step factor \( f \) defined by \( \Delta t^{n} = f \Delta t^{n} \), this becomes

\[ v_{i}^{\text{comb}} = (c_{1} + c_{2} + c_{3} + c_{4}) v(x_{i}^{n}, t_{n}) \\
+ (c_{2} + c_{4}) v_{t}(x_{i}^{n}, t_{n}) \Delta t + (c_{1} + c_{2}) (v_{i}^{n} \cdot \frac{\partial}{\partial x}) v(x_{i}^{n}, t_{n}) f \Delta t \\
+ O((\Delta t)^2) . \tag{18.13} \]

One can derive conditions for second order accuracy by comparing (18.13) with (18.12) and requiring that

\[ v_{i}^{\text{comb}} = v(x_{i}^{E,n+\frac{1}{2}}, t^{n+\frac{1}{2}}) + O((\Delta t)^2) . \tag{18.14} \]
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 $O(\Delta t^n)$ 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 (18.10) simplifies significantly.

An Euler step, as described for active/Euler in (18.7), 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:

\[
\begin{align*}
  \Delta t^n > \Delta t^{n-1} \quad &\text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq pt_{dtChangeToleranceUp} \times \Delta t^{n-1} \\
  \text{or} \quad \Delta t^n < \Delta t^{n-1} \quad &\text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq pt_{dtChangeToleranceDown} \times \Delta t^{n-1},
\end{align*}
\]

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 $x_{i}^{t_n+\frac{1}{2}}$ and $v_{i}^{t_n+\frac{1}{2}}$ between the Particles_advance calls at $t_n$ and $t_{n+1}$:

\begin{verbatim}
PARTICLEPROP velPredX REAL
PARTICLEPROP velPredY REAL
PARTICLEPROP velPredZ REAL
PARTICLEPROP posPredX REAL
PARTICLEPROP posPredY REAL
PARTICLEPROP posPredZ REAL
\end{verbatim}

The time integration of passive particles is tested in the ParticlesAdvance unit test, which can be used to examine the convergence behavior, see Section 18.3.4.

18.2 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-alpha, 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

\[ f_i(x) = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} f(x') \, dx' \]  

(18.15)

where \( i \) is the cell index and \( \Delta x \) is the spatial resolution. The mapping back and forth from the mesh to the particle properties are 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 \texttt{Config} file for each type of particle. See Section 6.6.1 for more details.

18.2.1 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

\[ f(x) = A + B(x-x_i) + C(x-x_i)^2 . \]  

(18.16)

Then integrating gives

\[
\begin{align*}
f_{i-1} &= \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x-x_i)^2 \right]^{x_{i-1/2}}_{x_{i-3/2}} + \frac{1}{3} C (x-x_i)^3 \bigg|^{x_{i-1/2}}_{x_{i-3/2}} \\
&= A - B \Delta x + \frac{13}{12} C \Delta x^2 , \tag{18.17}
\end{align*}
\]

and

\[
\begin{align*}
f_i &= \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x-x_i)^2 \right]^{x_{i+1/2}}_{x_{i-1/2}} + \frac{1}{3} C (x-x_i)^3 \bigg|^{x_{i+1/2}}_{x_{i-1/2}} \\
&= A + \frac{1}{12} C \Delta x^2 , \tag{18.18}
\end{align*}
\]

and

\[
\begin{align*}
f_{i+1} &= \frac{1}{\Delta x} \left[ A + \frac{1}{2} B (x-x_i)^2 \right]^{x_{i+3/2}}_{x_{i+1/2}} + \frac{1}{3} C (x-x_i)^3 \bigg|^{x_{i+1/2}}_{x_{i-3/2}} \\
&= A - B \Delta x + \frac{13}{12} C \Delta x^2 , \tag{18.19}
\end{align*}
\]

We may write these as

\[
\begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & -1 & \frac{13}{12} \\ 1 & 0 & \frac{13}{12} \\ 1 & 1 & \frac{13}{12} \end{bmatrix} \begin{bmatrix} A \\ B \Delta x \\ C \Delta x^2 \end{bmatrix} . \tag{18.20}
\]

Inverting this gives expressions for \( A, B, \) and \( C, \)

\[
\begin{bmatrix} A \\ B \Delta x \\ C \Delta x^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & -\frac{13}{24} & \frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix} . \tag{18.21}
\]

In two dimensions, we want a second-order interpolation function of the form

\[ f(x,y) = A + B(x-x_i) + C(x-x_i)^2 + D(y-y_j) + E(y-y_j)^2 + F(x-x_i)(y-y_j) . \]  

(18.22)
Finally, the above expressions apply only to Cartesian coordinates. In the case of cylindrical coordinates, the cell averaged quantities are given by

\[
\begin{align*}
\mathcal{F}_{i,j}(x, y) &\equiv \frac{1}{\Delta y} \Delta x \int_{x_{i-1/2}}^{x_{i+1/2}} dx' \int_{y_{j-1/2}}^{y_{j+1/2}} dy' \mathcal{F}(x', y') .
\end{align*}
\]  

(18.23)

Integrating the 9 possible cell averages gives, after some algebra,

\[
\begin{bmatrix}
f_{i-1,j+1} \\
f_{i,j-1} \\
f_{i+1,j+1} \\
f_{i,j-1} \\
\vdots \\
f_{i+1,j+1} \\
f_{i,j+1} \\
\vdots \\
f_{i+1,j+1}
\end{bmatrix} =
\begin{bmatrix}
1 & -1 & \frac{13}{13} & -1 & \frac{13}{13} & 1 \\
1 & 0 & \frac{13}{13} & -1 & \frac{13}{13} & 0 \\
1 & 1 & \frac{13}{13} & -1 & \frac{13}{13} & -1 \\
1 & -1 & \frac{13}{13} & 0 & \frac{13}{13} & 0 \\
1 & 1 & \frac{13}{13} & 0 & \frac{13}{13} & 0 \\
1 & 1 & \frac{13}{13} & 1 & \frac{13}{13} & -1 \\
1 & 0 & \frac{13}{13} & 1 & \frac{13}{13} & 0 \\
\vdots \\
1 & 1 & \frac{13}{13} & 1 & \frac{13}{13} & 1
\end{bmatrix}
\begin{bmatrix}
A \\
B \Delta x \\
C \Delta x^2 \\
D \Delta y \\
E \Delta y^2
\end{bmatrix} .
\]  

(18.24)

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

\[
\begin{bmatrix}
\mathcal{F}_{i,j-1} \\
\mathcal{F}_{i-1,j} \\
\vdots \\
\mathcal{F}_{i,j+1}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & \frac{13}{13} & -1 & \frac{13}{13} & 1 \\
1 & -1 & \frac{13}{13} & 0 & \frac{13}{13} & 0 \\
\vdots \\
1 & 0 & \frac{13}{13} & 1 & \frac{13}{13} & 1
\end{bmatrix}
\begin{bmatrix}
A \\
B \Delta x \\
C \Delta x^2 \\
D \Delta y \\
E \Delta y^2
\end{bmatrix} .
\]  

(18.25)

Inverting gives

\[
\begin{bmatrix}
A \\
B \Delta x \\
C \Delta x^2 \\
D \Delta y \\
E \Delta y^2
\end{bmatrix} =
\begin{bmatrix}
-\frac{1}{24} & -\frac{1}{24} & \frac{7}{6} & -\frac{1}{24} & -\frac{1}{24} \\
0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\
0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \\
-\frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\
\frac{1}{2} & 0 & -1 & 0 & \frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
\mathcal{F}_{i,j-1} \\
\mathcal{F}_{i-1,j} \\
\vdots \\
\mathcal{F}_{i,j+1}
\end{bmatrix}.
\]  

(18.26)

Similarly, in three dimensions, the interpolation function is

\[
f(x, y, z) = A + B (x - x_i) + C (x - x_i)^2 + D (y - y_j) + E (y - y_j)^2 + F (z - z_k) + G (z - z_k)^2 .
\]  

(18.27)

and we have

\[
\begin{bmatrix}
A \\
B \Delta x \\
C \Delta x^2 \\
D \Delta y \\
E \Delta y^2 \\
F \Delta z \\
G \Delta z^2
\end{bmatrix} =
\begin{bmatrix}
-\frac{1}{24} & -\frac{1}{24} & \frac{5}{4} & -\frac{1}{24} & -\frac{1}{24} \\
0 & 0 & -\frac{1}{2} & 0 & \frac{1}{2} \\
0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \\
0 & -\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & \frac{1}{2} & 0 & -1 & 0 \\
\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\
\frac{1}{2} & 0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
\mathcal{F}_{i,j,k-1} \\
\mathcal{F}_{i,j-1,k} \\
\mathcal{F}_{i-1,j,k} \\
\mathcal{F}_{i,j,k} \\
\mathcal{F}_{i+1,j,k} \\
\mathcal{F}_{i,j+1,k} \\
\mathcal{F}_{i,j,k+1}
\end{bmatrix}.
\]  

(18.28)

Finally, the above expressions apply only to Cartesian coordinates. In the case of cylindrical \((r, z)\) coordinates, we have

\[
f(r, z) = A + B (r - r_i) + C (r - r_i)^2 + D (z - z_j) + E (z - z_j)^2 + F (r - r_i) (z - z_j) .
\]  

(18.29)
and

\[
\begin{bmatrix}
A \\
B \Delta r \\
C \Delta r^2 \\
D \Delta z \\
E \Delta z^2
\end{bmatrix} =
\begin{bmatrix}
- \frac{1}{2h_1} & - \frac{h_1-1}{2h_1} & \frac{7}{2} & \frac{1}{2h_1} & \frac{1}{2h_1} \\
0 & \frac{3h_2}{3h_2} & 0 & - \frac{(7+6h_1)(h_1-1)}{3h_2} & 0 \\
0 & \frac{h_1h_2}{h_1h_2} & -2 & \frac{12h_1^2-13}{h_1h_2} & - \frac{(12h_1^2+12h_1-1)(h_1-1)}{h_1h_2} \\
- \frac{1}{2} & 0 & \frac{1}{7} & -1 & 0 \\
0 & \frac{7}{7} & \frac{7}{7} & 0 & \frac{1}{2h_1} \\
0 & \frac{1}{2h_1} & \frac{1}{2h_1} & \frac{1}{2h_1} & \frac{1}{2h_1} \\
\end{bmatrix}
\begin{bmatrix}
f_{i,j-1} \\
f_{i-1,j} \\
f_{i,j} \\
f_{i+1,j} \\
f_{i,j+1}
\end{bmatrix}.
\] (18.30)

18.2.2 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-alpha 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 `-with-unit=Particles/ParticlesMapping/CIC`. Two additional command line option `-with-unit=Particles/ParticlesMapping/MapToMesh` and `-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 `REQUIRES/REQUESTS` directives in the Simulation `Config` file.

18.3 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 FLASH3.1 it is possible to include multiple different types of particles in the same simulation. Particle types must be specified in the `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 `PARTICLETYPES` specifications in the `Config` files, and generates an F90 file `Particles specifyMethods.F90` that populates a data structure `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 `Particles` unit `Config` files. These various functionalities are organized into different sub-units; a brief description of each subunit is included below and further expanded in subsections in this chapter.

- The `ParticlesMain` sub-unit contains the various time-integration options for both active and passive particles. A detailed overview of the different schemes is given in Section 18.1.

- The `ParticlesMapping` sub-unit 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 18.2 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-alpha. 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 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 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-alpha distribution (they are more likely to be 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.

- The ParticlesForces subunit implements the long and short range forces described in Equation (18.3) 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/ParticlesMain/active/-longRange, which requires that the Gravity unit be included in the code. In the current release, no short-range force laws are supplied with FLASH.

After particles are moved during time integration or by forces, they may end up on other blocks within or outwith 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 18.3.3 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-alpha, 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.

### 18.3.1 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.

### 18.3.2 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 18.1). 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.

### 18.3.3 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.
18.3.4 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

```bash
./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 18.1.3. Add other options as desired.

For unitTest/ParticlesAdvance/HomologousPassive,

```bash
./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 criterium if sim_schemeOrder = 0.
- **PARAMETER sim_maxTolCoeff1 REAL 0.0001** — First order error coefficient $C_1$, used for convergence criterium if sim_schemeOrder = 1.
- **PARAMETER sim_maxTolCoeff2 REAL 0.01** — Second order error coefficient $C_2$, used for convergence criterium if sim_schemeOrder = 2.

A test for order $k$ is considered successful if the following criterium is satisfied:

$$\text{maxError} \leq C_k \times \text{maxActual Dt}^k,$$

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 dtmax by several orders of magnitude (over a range where it directly determines maxActualDt) for a given test in order to examine convergence behavior.
Chapter 19

Cosmology Unit

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.

19.1 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 $v$ is taken to be the comoving peculiar velocity $\dot{x}$. The comoving gas
density, pressure, temperature, and internal energy are defined to be

\[ \begin{align*}
\rho & \equiv a^3 \tilde{\rho} \\
p & \equiv a \tilde{p} \\
T & \equiv \frac{T}{a^2} \\
\rho\epsilon & \equiv a \tilde{\rho} \tilde{\epsilon}.
\end{align*} \tag{19.1} \]

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

\[ \rho\epsilon = \frac{p}{\gamma - 1} = \frac{\rho k T}{(\gamma - 1) \mu}. \tag{19.2} \]

With these definitions, the Euler equations of hydrodynamics can be written in the form

\[ \begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \tag{19.3} \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla p + 2 \frac{\dot{a}}{a} \rho \mathbf{v} + \rho \nabla \phi &= 0 \tag{19.4} \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p)\mathbf{v}] + \frac{\dot{a}}{a} (3\gamma - 1) \rho \epsilon + 2 \rho v^2 + \rho \mathbf{v} \cdot \nabla \phi &= 0 \tag{19.5} \\
\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + p)\mathbf{v}] - \mathbf{v} \cdot \nabla p + \frac{\dot{a}}{a} (3\gamma - 1) \rho \epsilon &= 0. \tag{19.6}
\end{align*} \]

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

\[ \nabla^2 \phi = \frac{4\pi G}{a^3} (\rho - \bar{\rho}) \tag{19.7} \]

where \( \bar{\rho} \) is the comoving mean matter density. Note that, because of the presence of \( a \) in (19.7), 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 (19.7). 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

\[ \begin{align*}
\bar{\rho} & \equiv \Omega_m \rho_{\text{crit}} \\
\rho_{\text{crit}} & \equiv \frac{3H^2}{8\pi G}. \tag{19.8}
\end{align*} \]

The Hubble parameter \( H(t) \) [to be distinguished from the Hubble “constant” \( H_0 \equiv H(t_0) \)] is given by the Friedman equation

\[ H^2(t) \equiv \left( \frac{\dot{a}}{a} \right)^2 = H_0^2 \left( \frac{\Omega_m}{a^3} + \frac{\Omega_r}{a^4} + \frac{\Omega_{\Lambda}}{a^2} \right). \tag{19.9} \]

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

\[ \Omega_c \equiv \Omega_m + \Omega_r + \Omega_{\Lambda} - 1. \tag{19.10} \]
The `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 `csm_integrateFriedman` 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 `Particles` unit. For an example, see the `LeapfrogCosmo` implementation of the `ParticlesMain` subunit in Section 18.1.1.

## 19.2 Using the Cosmology unit

To include cosmological expansion in your FLASH executable, include the line

```plaintext
REQUESTS physics/Cosmology/
```

in your setup’s `Config` file. At present, the Cosmology unit in FLASH4-alpha is built around the `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 `Cosmology` unit are described in Table 19.1. Note that the total effective mass density is not explicitly specified but is inferred from the sum of the $\Omega_{\text{Matter}}$, $\Omega_{\text{Radiation}}$, and $\Omega_{\text{CosmologicalConstant}}$ parameters. The `MaxScaleChange` parameter sets the maximum allowed fractional change in the scale factor $a$ during a single timestep. This behavior is enforced by the `Cosmology.computeDt` routine. The default value is set to the system’s `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><code>useCosmology</code></td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>True if cosmology is to be used in this simulation</td>
</tr>
<tr>
<td>$\Omega_{\text{Matter}}$</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>$\Omega_{\text{Baryon}}$</td>
<td>REAL</td>
<td>0.05</td>
<td>Ratio of baryonic (gas) mass density to critical density at the present epoch; must be $\leq \Omega_{\text{Matter}}$ ($\Omega_b$)</td>
</tr>
<tr>
<td>$\Omega_{\text{CosmologicalConstant}}$</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>$\Omega_{\text{Radiation}}$</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>$H_0$</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><code>MaxScaleChange</code></td>
<td>REAL</td>
<td><code>HUGE(1.)</code></td>
<td>Maximum permitted fractional change in the scale factor during each timestep</td>
</tr>
</tbody>
</table>

The `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.

- **Cosmology.cdmPowerSpectrum**
  Return the present-day cold dark matter power spectrum as a function of a given wavenumber. The `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).
• 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.

19.3 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:

OmegaMatter = 1.0
OmegaLambda = 0.0
OmegaBaryon = 1.0
OmegaRadiation = 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 20

Material Properties Units

Figure 20.1: The materialProperties unit with Opacity, MagneticResistivity and Viscosity subunits.

Figure 20.2: The materialProperties unit with MassDiffusivity and Conductivity subunits.
20.1 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-alpha release provides two implementations:

- **Constant** for heat conduction with a constant isochoric conductivity;
- **Constant-diff** for heat conduction with a constant coefficient of diffusion.

To use thermal conductivity in a FLASH4-alpha simulation, the runtime parameter `useConductivity` must be set to `.true.`

20.2 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 $\nu_m = \frac{c^2}{4\pi} \eta$ in CGS and $\nu_m = \frac{1}{\mu_0} \eta$ in SI. The default top level routines return zero values for resistivity (a stub functionality). The constant resistivity routine is provided in the low level subdirectory `/MagneticResistivityMain/Constant`.

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.

20.2.1 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, $\frac{c^2}{4\pi} \eta$ is returned in CGS unit, $\frac{1}{\mu_0} \eta$ in SI, and simply $\eta$ in non-dimensional unit.

In previous implementations, there used to be two runtime parameters: magnetic resistivity and magnetic viscosity. They respectively refer $\eta$ and $\frac{c^2}{4\pi} \eta$ in CGS (or $\frac{1}{\mu_0} \eta$ 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., $\frac{c^2}{4\pi} \eta$) 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.
20.3 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 \texttt{visc\_whichCoefficientIsConst}.

To use viscosity in a FLASH4-alpha simulation, the runtime parameter \texttt{useViscosity} must be set to \texttt{.true.}

20.4 Opacity

The Opacity unit residing in \texttt{physics/materialProperties/Opacity} is currently based on extracting constant and/or tabulated absorption, emission and transport opacities from files produced by some external source code (currently only IONMIX). Three tables containing Planck Absorption, Planck Emission and Rosseland opacities are created internally in FLASH. Each table contains the opacities in cm$^2$/g as a function of a temperature/density grid and radiation energy group. A fourth handle index is added linking particular tables to each of the species for which that particular opacity was created. The storage layout for each of the three tables is:

\[
\kappa(T, D, E, h) \rightarrow T = \text{Temperature grid index} \\
D = \text{Density grid index} \\
E = \text{Radiation energy group index} \\
h = \text{Species handle index}.
\] (20.1)

Since each of the files created by the external source for each species may contain a different underlying temperature/density grid, auxiliary arrays are needed to store details of each grid for each species. Since all the opacity tables produced by FLASH need to be kept in memory throughout the entire application run, efficient storage of these tables and their associated data is necessary. For this, a first browsing step is done, in which each temperature/density grid for all the needed species is analyzed and the maxima of both the temperature and density grid sizes are taken as delimiting dimensions for the opacity tables. The above mentioned introduction of the species handle index in (20.1) avoids unnecessary duplication of opacity data.

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:

\[
\tau_1 = \frac{T_2 - t}{T_2 - T_1} \\
\tau_2 = \frac{t - T_1}{T_2 - T_1} \\
\delta_1 = \frac{D_2 - d}{D_2 - D_1} \\
\delta_2 = \frac{d - D_1}{D_2 - D_1} \\
\kappa(t, d) = \sum_{i=1}^{2} \sum_{j=1}^{2} \tau_i \delta_j \kappa_{ij}.
\] (20.2)

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.

The communication between the Opacity unit and a particular HEDP run is provided via a text file \texttt{opacity\_input.txt}, containing all the opacity information needed for each species. The species are identified through an integer index and the text file must contain at least as many entries as there are species declared.
Below is a snippet of the text file corresponding to one species together with other options and explanation of the entries:

<table>
<thead>
<tr>
<th>Snippet</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planck Absorption</td>
<td>Kind of absorption opacity to be used</td>
</tr>
<tr>
<td>Planck Emission</td>
<td>Kind of emission opacity to be used</td>
</tr>
<tr>
<td>Rosseland</td>
<td>Kind of transport opacity to be used</td>
</tr>
<tr>
<td>0.000000E+00</td>
<td>Value for Constant absorption opacity (if specified)</td>
</tr>
<tr>
<td>0.000000E+00</td>
<td>Value for Constant emission opacity (if specified)</td>
</tr>
<tr>
<td>0.000000E+00</td>
<td>Value for Constant transport opacity (if specified)</td>
</tr>
<tr>
<td>IONMIX</td>
<td>type of external source tables</td>
</tr>
<tr>
<td>be-100grp-lte.imx</td>
<td>file name of external source tables</td>
</tr>
</tbody>
</table>

Note the possibility of having all four options (Constant, Planck Absorption, Planck Emission and Rosseland) for each of the three type of opacities (absorption, emission and transport). This flexibility is provided such that possible outcomes of the IONMIX program can be overridden. As an example, the user may want to have absorption opacity = emission opacity for a particular species, but IONMIX always produces different output for Planck absorption and Planck emission opacities. In this case the user would simply put 'Planck Absorption' on the first two lines of the species snippet.

There is currently only one boolean runtime parameter for the Opacity unit: \texttt{opacity\_useLogTables}. If it is set true (the default value), then the tabular interpolation will be done on the logarithmic opacities. If false, the original opacity values are taken. If the logarithmic interpolation is chosen, the user must make sure that the tabulated opacities produced by the external source code do not contain any zero values. Otherwise the logarithmic interpolation will fail. For IONMIX produced opacities this can happen for the Planck emission opacities. An easy fix around that problem is to not use the Planck emission opacities for any of the absorption, emission and transport cell opacities. A more elaborate fix would be to replace any zeros by the smallest positive computational value in the generated opacity tables.

\section*{20.5 Mass Diffusivity}

The \texttt{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 21

Radiative Transfer Unit

Figure 21.1: The organizational structure of the RadTrans unit.

The RadTrans unit is responsible for solving the radiative transfer equation

$$\frac{1}{c} \frac{\partial I}{\partial t} + \hat{\Omega} \cdot \nabla I + \rho \kappa I = \eta,$$  \hspace{1cm} (21.1)

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 cm$^2$/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

$$\frac{\partial u_e}{\partial t} = \int_0^\infty d\nu \int_{4\pi} d\Omega (\rho \kappa I - \eta),$$  \hspace{1cm} (21.2)

where $u_e$ is the electron internal energy density.

The RadTrans unit is responsible for solving the radiative transfer equation and updating the electron energy using (21.2) 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 RadTrans solver, MGD, which uses a simple multigroup diffusion approximation and is described in Section 21.1.

21.1 Multigroup Diffusion

The radiative transfer (21.1) and electron energy (21.2) 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

\[
\frac{1}{c} \frac{\partial u_g}{\partial t} - \nabla \cdot \left( \frac{1}{3\sigma_{t,g}} \nabla u_g \right) + \sigma_{a,g} u_g = \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} \left[ P(x_{g+1}) - P(x_g) \right]
\] (21.3)

\[
\frac{\partial u_g}{\partial t} = \sum_g \left\{ \sigma_{a,g} u_g - \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} \left[ P(x_{g+1}) - P(x_g) \right] \right\}
\] (21.4)

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 \( \frac{h\nu}{k_B T_e} \) where \( h \) is Planck’s constant and \( k_B \) is the Boltzmann constant. The opacities in (21.3) and (21.4) are in units of \( \text{cm}^{-1} \) and \( \sigma = \rho \kappa \). The Planck integral, \( P(x) \) is defined as

\[
P(x) = \int_0^x \frac{(x')^3}{\exp(x') - 1} \, dx'
\] (21.5)

The multigroup diffusion equation and electron internal energy equation are operator split using an explicit treatment. Thus, on each time step (21.4) 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

\[
\frac{1}{c} \frac{u_g^{n+1} - u_g^n}{\Delta t} - \nabla \cdot \left( D_g^n \nabla u_g^{n+1} \right) + \sigma_{a,g} u_g^{n+1} = \sigma_{e,g} a (T_e^n)^4 \frac{15}{\pi^4} \left[ P(x_{g+1}^{n+1}) - P(x_g^n) \right]
\] (21.6)

\[
\frac{u_g^{n+1} - u_g^n}{\Delta t} = \sum_g \left\{ \sigma_{a,g} u_g^{n+1} - \sigma_{e,g} a (T_e^n)^4 \frac{15}{\pi^4} \left[ P(x_{g+1}^{n+1}) - P(x_g^n) \right] \right\}
\] (21.7)

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, \( c u_g^n \). The following equation shows how the diffusion coefficient is modified to incorporate the flux limiter using a harmonic average

\[
D_g^n = \frac{1}{3\sigma_{t,g}^n + \frac{\|\nabla u_g^n\|}{\alpha_{e,c} u_g^n}}
\] (21.8)

where \( \alpha_c \) is the radiation flux-limiter coefficient, typically set to one. The rest of this section describes the necessary options and steps which are necessary to use multigroup radiation diffusion in FLASH.

### 21.1.1 Config File and Mesh Replication

The diffusion equation for each energy group is solved independently because of the explicit coupling of the radiation field to the plasma. In other words, the time step begins with the solution of (21.6) for each energy group independent of the others. Thus, it is possible to solve the equation for each energy group in parallel then add the total emission and absorption across the groups to solve (21.7). FLASH has a new capability, called mesh replication (MR), which is designed to take advantage of this type of parallelism. MR allows the user to create several copies of the mesh which various physics units can use to solve certain problems in parallel. In this case, several identical copies of the mesh are made and are divided among the various processes and identical operations occur on each mesh, until the \texttt{RadTrans} unit begins operating. Each mesh has an identical refinement pattern when AMR is in use. \texttt{RadTrans} assigns different energy groups to each mesh. The diffusion equation for each group is solved according to (21.6). Finally, the emission and absorption terms are summed across the meshes and the electron and total radiation specific energies are updated.

For example, lets assume that a user would like to run a simulation with \( N_g = 7 \). Furthermore, assume that the mesh has only a small number of cells so that after two processors are used for domain decomposition, no more improvement in performance is experienced. Finally, assume that there are six processors available to run the simulation. Without mesh replication, the user would run the problem with two processors and no
other steps could be taken to reduce the runtime - even though additional processors are available since after
two processors, the strong scaling limit has been reached. With mesh replication it is possible to continue
to reduce the simulation runtime by assigning a few groups to each mesh. In this case we will use three
meshes and two processors will be assigned to each mesh. Mesh 1 will be assigned energy groups 1, 4, and
7, mesh 2 will be assigned groups 2 and 5, and mesh 3 will be assigned groups 3 and 6. Now all processors
are used and the overall simulation run time is decreased. The number of meshes, \( N_m \), is specified using the
parameter `meshCopyCount` which defaults to 1, meaning that no mesh replication is used. In this example,
`meshCopyCount` would be set to 3 and 6 processors, total, would be used to run the simulation.

A mass scalar must exist for storing the specific energy within each energy group. A new Config file
directive, `NONREP`, has been developed for creating arrays of mass scalars which will be divided amongst the
various meshes when mesh replication is used. For this example, the following command must be placed
within the Config file:

```
NONREP MASS_SCALAR MGD 3 rt_mgdNumGroups R???
```

The only argument which changes between Simulations is the number, 3 in this case, which appears in the
`NONREP` command. This number, which will be referred to as \( N_{ms} \), represents the number of mass scalars
which are being created on each mesh for storing the specific energy in each group. This number is constant
among the meshes. In our example, only mesh 1 stored three energy groups - but storage for three groups
is created on meshes 2 and 3. This means that there is unused memory on meshes 2 and 3. The number
\( N_{ms} \times N_m \) represents the maximum number of radiation energy groups which can be used in a particular
simulation. The `RadTrans` is designed to function if \( N_g < N_{ms} \times N_m \). In this case there will simply be mass
scalars that are never used. However, note that in our example, while there is wasted memory on meshes 2
and 3, the amount of memory used per node has actually been reduced. If no mesh replication were used,
seven mass scalars would have to be created; this has been reduced to three. The actual number of groups
used in a simulation, \( N_g \), is specified in a runtime parameter, as described in the next subsection.

One caveat is that, FLASH can only operate in a mode where \( N_g \neq N_{ms} \times N_m \) for certain `IO` units.
These are the parallel versions of the uniform grid, non-fixed block size, and `PARAMESH` `IO` units. Most
importantly, the restriction \( N_g = N_{ms} \times N_m \) does apply to the default serial `IO` unit. It is relatively easy
to switch to alternative implementations. If this is not possible, the `NONREP` command must be set so that
\( N_g = N_{ms} \times N_m \).

The `NONREP` command that we described above is the only modification that must be made to the Config
file, with the exception of including the MGD subunit which resides in `physics/RadTrans/RadTransMain/-
MGD`. Some subunit within the `Diffuse` unit must also be included. For example, `physics/Diffuse/-
DiffuseMain/Unsplit` is acceptable. The `NONREP` command takes several arguments, but only the number
of mass scalars per mesh, \( N_{ms} \), is changed between simulations. Also, the total number of processors used in
the simulation must be an integer multiple of \( N_m \) - otherwise, different number of processors will be assigned
to each mesh and FLASH is not currently able to operate in this configuration.

### 21.1.2 Runtime Parameters

The parameter `rt_useMGD` is used to activate MGD in FLASH. The runtime parameter `rt_mgdNumGroups`
sets the number of groups, \( N_g \) in the simulation. As was described earlier, the inequality \( N_g < N_{ms} \times N_m \)
must be satisfied or insufficient memory will be available. Next, the user must specify the energy group
boundaries. Each energy group represents photons within an energy range from \( E_g = h\nu_g \) to \( E_{g+1} = h\nu_{g+1} \).
Thus, \( N_{g+1} \) energy group boundaries must be given for any simulation using MGD. The parameter
`rt_mgdBoundEntry` controls how the energy group boundaries are specified. Currently, the only supported
value is “grbd_manual” which tells FLASH that the values will be entered manually into the parameter
file. The parameters `rt_mgdBound_1` to `rt_mgdBound_101` are used to specify the group boundary energies.
Currently, the units used are electron-Volts. This was chosen as the unit because the vast majority of
opacity files specify group boundaries in eV. Finally, the flux-limiter for MGD is defined. The parameter
`rt_mgdFlMode` is set to “fl_none”, “fl_harmonic”, or “fl_minmax” to specify the type of flux limiter to use
and `rt_mgdFlCoeff`, which defaults to 1.0, sets the value of \( \alpha_r \). By default, `rt_mgdFlMode` is set to “fl_none”
meaning that no flux limiter is used. This is almost never the desired behavior - typically the user should set this to “fl_harmonic” to use the form given in (21.8).
The code segment below shows parameter file setting to use four group MGD with energy group boundaries: 1, 400, 800, 1500, and 100,000 eV.

rt_useMGD = .true.
rt_mgdNumGroups = 4
rt_mgdBoundEntry = "grbd_manual"
rt_mgdBounds_1 = 1.0
rt_mgdBounds_2 = 400.0
rt_mgdBounds_3 = 800.0
rt_mgdBounds_4 = 1500.0
rt_mgdBounds_5 = 100000.0
rt_mgdFlMode = "fl_harmonic"
rt_mgdFlCoef = 1.0

21.1.3 Specifying Initial Conditions

Finally, 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 ergs/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 three steps:

1. The value of \( e_g \) must be set for each group in each cell

2. The total radiation specific radiation energy, \( e_r = \sum_g e_g \), must be computed and stored in the ERAD mass scalar

3. Third, the radiation temperature, TRAD, must be set for each cell such that \( E_r = aT_r^4 \)

Routines have been created in the RadTrans unit for simplifying steps one and two. However, ensuring that step three is satisfied is the responsibility of the user. This can be done automatically through the EOS unit. Step two is performed using the subroutine RadTrans_sumEnergy.

There are two methods for setting \( e_g \). The easiest method involves using the routine RadTrans_mgdEFromT to automatically set \( e_g \) given a radiation temperature assuming a Planck spectrum. It is also possible to set \( e_g \) for a specific group using RadTrans_mgdSetEnergy. Using this routine, the user can ignore the details of mesh replication. The use of these methods is best illustrated through an example. Below, a code segment from a Simulation_initBlock.F90 shows how to specify the initial conditions using a radiation temperature.

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
    call Grid_putPointData(blockId, CENTER, DENS_VAR, EXTERIOR, axis, sim_rho)
    call Grid_putPointData(blockId, CENTER, TION_VAR, EXTERIOR, axis, sim_tion)
    call Grid_putPointData(blockId, CENTER, TELE_VAR, EXTERIOR, axis, sim_tele)
    call Grid_putPointData(blockId, CENTER, TEMP_VAR, EXTERIOR, axis, sim_tele)
    ! Set the radiation temperature:
    call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, sim_trad)
    ! Set the specific energy in each radiation group:
    call RadTrans_mgdEFromT(blockId, axis, sim_trad)
21.1. MULTIGROUP DIFFUSION

In this example, the user has specified the initial radiation temperature through the parameter file. The radiation temperature is manually specified through a call to Grid_putPointData. The values of $e_g$ are automatically set through a call to RadTrans_mgdEFromT. Finally, the values of $e_g$ are summed over all of the meshes for a single block using a call to RadTrans_sumEnergy.

The next example assumes that the user would like to initialize the radiation field so that $e_1 = aT^4/r$ and all other groups are initialized to zero energy. Note, that for this example, the user must compute the value of the radiation constant, $a = 4\sigma_{SB}/c$, using the PhysicalConstants unit. If this is not done, then small errors in energy conservation will manifest themselves in the simulation. This simulations used $N_g = 4$.

Using the MGD solver in FLASH is straightforward. The NONREP line must be present in the Config file, the MGD runtime parameters must be specified, and the initial conditions must be defined. Note, that this section has not discussed the process for setting the opacities of the materials. This is documented in the separate chapter for the Opacity unit.
Part VI

Monitor Units
Chapter 22

Logfile Unit

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"

22.1 Meta Data

The logfile stores meta data about a given run including the time and date of the run, the number of processors, 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:

<table>
<thead>
<tr>
<th>Number of processors:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality:</td>
<td>2</td>
</tr>
<tr>
<td>Max Number of Blocks/Proc:</td>
<td>1000</td>
</tr>
<tr>
<td>Number x zones:</td>
<td>8</td>
</tr>
<tr>
<td>Number y zones:</td>
<td>8</td>
</tr>
<tr>
<td>Number z zones:</td>
<td>1</td>
</tr>
<tr>
<td>Build stamp: Wed Apr 19 16:35:57 2006</td>
<td></td>
</tr>
<tr>
<td>System info:</td>
<td></td>
</tr>
<tr>
<td>Linux zingiber.uchicago.edu 2.6.12-1.1376_FC3smp #1 SMP Fri Aug 26 23:50:33 EDT</td>
<td></td>
</tr>
<tr>
<td>Version: FLASH 3.0.</td>
<td></td>
</tr>
<tr>
<td>Build directory: /home/kantypas/FLASH3/trunk/Sod</td>
<td></td>
</tr>
<tr>
<td>Setup syntax:</td>
<td></td>
</tr>
<tr>
<td>/home/kantypas/FLASH3/trunk/bin/setup.py Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM -objdir=Sod</td>
<td></td>
</tr>
<tr>
<td>f compiler flags:</td>
<td></td>
</tr>
<tr>
<td>/usr/local/pgi6/bin/pgf90 -I/usr/local/mpich-pg/include -c -r8 -i4 -fast -g -DMAXBLOCKS=1000 -DNXB=8 -DNYB=8 -DNZB=1 -DN_DIM=2</td>
<td></td>
</tr>
<tr>
<td>c compiler flags:</td>
<td></td>
</tr>
<tr>
<td>/usr/local/pgi6/bin/pgcc -I/usr/local/hdf5-pg/include -I/usr/local/mpich-pg/include -c -O2 -DMAXBLOCKS=1000 -DNXB=8 -DNYB=8 -DNZB=1 -DN_DIM=2</td>
<td></td>
</tr>
</tbody>
</table>

22.2 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. 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 “...”.

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:

<table>
<thead>
<tr>
<th>Unit</th>
<th>CGS Value</th>
<th>Base Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
22.3 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_expansionDomain]: create level=2
...
[GRID amr_refine_derefine] initiating refinement
[GRID amr_refine_derefine] min blks 250 max blks 251 tot blks 501
[GRID amr_refine_derefine] min leaf blks 188 max leaf blks 188 tot leaf blks 376
[GRID amr_refine_derefine] refinement complete
[ 04-19-2006 16:40.44 ] [GRID gr_expansionDomain]: create level=7
[ 04-19-2006 16:40.44 ] [GRID gr_expansionDomain]: create level=7

==============================================================================
22.4 Performance Data

Finally, the log file records performance data for the simulation. The Timers unit (see Section 23.1) 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 log file. The snippet below shows the performance data section of a log file.

```plaintext
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</th>
<th>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></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>
<td></td>
</tr>
<tr>
<td>writeCheckpoint</td>
<td>0.085</td>
<td>1</td>
<td>0.085</td>
<td>0.365</td>
<td></td>
</tr>
<tr>
<td>writePlotfile</td>
<td>0.061</td>
<td>1</td>
<td>0.061</td>
<td>0.264</td>
<td></td>
</tr>
<tr>
<td>evolution</td>
<td>22.176</td>
<td>1</td>
<td>22.176</td>
<td>95.633</td>
<td></td>
</tr>
<tr>
<td>hydro</td>
<td>18.214</td>
<td>40</td>
<td>0.033</td>
<td>11.227</td>
<td></td>
</tr>
<tr>
<td>sourceTerms</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>particles</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>Grid_updateRefinement</td>
<td>1.238</td>
<td>20</td>
<td>0.062</td>
<td>5.340</td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td>1.126</td>
<td>10</td>
<td>0.113</td>
<td>4.856</td>
<td></td>
</tr>
<tr>
<td>guardcell tree</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.459</td>
<td></td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.458</td>
<td></td>
</tr>
<tr>
<td>markRefineDerefine</td>
<td>0.339</td>
<td>10</td>
<td>0.034</td>
<td>1.460</td>
<td></td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.053</td>
<td>10</td>
<td>0.005</td>
<td>0.230</td>
<td></td>
</tr>
<tr>
<td>amr_refine_derefine</td>
<td>0.003</td>
<td>10</td>
<td>0.000</td>
<td>0.011</td>
<td></td>
</tr>
<tr>
<td>updateData</td>
<td>0.002</td>
<td>10</td>
<td>0.000</td>
<td>0.009</td>
<td></td>
</tr>
<tr>
<td>guardcell</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.453</td>
<td></td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.452</td>
<td></td>
</tr>
<tr>
<td>eos</td>
<td>0.111</td>
<td>10</td>
<td>0.011</td>
<td>0.481</td>
<td></td>
</tr>
<tr>
<td>update particle refinemen</td>
<td>0.000</td>
<td>10</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
</tbody>
</table>
```
22.5 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 23

Timer and Profiler Units

23.1 Timers

23.1.1 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 timer_summary_&lt;process id&gt;</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 22.4
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.

23.1.2 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 23.1.1) 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.
23.2 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 this release, the unit does not have an implementation, but provides a structured way to easily pick different profiling packages for the code at compile time. Expected implementation packages include the MPE trace logs from mpich, Vampir traces (now an Intel product), or the IBM HPM profilers.

The Profiler routines are called by the Timers/TimersMain routines. For instance, a call to start a timer will also call Profilers_start. This duplication means that code sections instrumented with FLASH timers will also be instrumented for the included third-party profiler package. The Profiler interface can also be called separately, so sections can be specifically instrumented for the profilers.
Part VII

Simulation Units
Chapter 24

The Supplied Test Problems

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.

24.1 Hydrodynamics Test Problems

These problems are primarily designed to test the functioning of the hydrodynamics solvers within FLASH4-alpha.
CHAPTER 24. THE SUPPLIED TEST PROBLEMS

24.1.1 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

\[
\begin{align*}
\rho_L &= 1.0 \\
\rho_R &= 0.125 \\
p_L &= 1.0 \\
p_R &= 0.1
\end{align*}
\] (24.1)

The ratio of specific heats $\gamma$ is chosen to be 1.4 on both sides of the interface.

In FLASH, the Sod problem (Sod) uses the runtime parameters listed in Table 24.1 in addition to those supplied by default with the code. For this problem we use the Gamma equation of state alternative implementation and set $\gamma$ to 1.4. The default values listed in Table 24.1 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>sim_rhoLeft</td>
<td>real</td>
<td>1</td>
<td>Initial density to the left of the interface ($\rho_L$)</td>
</tr>
<tr>
<td>sim_rhoRight</td>
<td>real</td>
<td>0.125</td>
<td>Initial density to the right ($\rho_R$)</td>
</tr>
<tr>
<td>sim_pLeft</td>
<td>real</td>
<td>1</td>
<td>Initial pressure to the left ($p_L$)</td>
</tr>
<tr>
<td>sim_pRight</td>
<td>real</td>
<td>0.1</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 24.2 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 24.3 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 24.2: 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 24.3: 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 24.4 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.

![Colormap plot of density in the diagonal 2D Sod problem with AMR grid.](image)

Figure 24.4: 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).

FLASH4-alpha also contains under the name \texttt{SodStep} a variant of the Sod problem. This setup is provided as an example for setting up simulations on domains with steps and obstacles. See the files in the \texttt{SodStep} simulation directory and the \texttt{Simulation\_defineDomain} description for more information on how to use this feature.

### 24.1.2 Variants of the Sod Problem in Curvilinear Geometries

Variants of the Sod problems can be set up in various geometries in order to test the handling of non-Cartesion geometries.

- An axisymmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with \texttt{./setup Sod -auto -2d -geometry=cylindrical} and using runtime parameters that include \texttt{geometry = "cylindrical"}. Use \texttt{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.

### 24.1.3 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

\[
p_L = 1000, \quad p_M = 0.01, \quad p_R = 100. \tag{24.2}
\]

The equation of state is that of a perfect gas with \( \gamma = 1.4 \).

Figure 24.5 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 24.5 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 24.6 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

\[
\mathcal{E}(N_{\text{ref}}; u) \equiv \frac{1}{N(N_{\text{ref}})} \sum_{i=1}^{N(N_{\text{ref}})} \left| \frac{u_i(N_{\text{ref}}) - A u_i(10)}{u_i(10)} \right| , \tag{24.3}
\]

against the effective number of cells \( (N(N_{\text{ref}})) \). In computing this norm, both the ‘converged’ solution \( u(10) \) and the test solution \( u(N_{\text{ref}}) \) are interpolated onto a uniform mesh having \( N(N_{\text{ref}}) \) cells. Values of \( N_{\text{ref}} \) between 2 (corresponding to cell size \( \Delta x = 1/16 \)) and 9 (\( \Delta x = 1/2048 \)) are shown.
Figure 24.5: Density and velocity profiles in the Woodward-Colella interacting blast-wave problem \texttt{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 24.2. This problem is configured to use the perfect-gas equation of state (gamma) with 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.
24.1. HYDRODYNAMICS TEST PROBLEMS

Figure 24.6: Self-convergence of the density, pressure, and total specific energy in the Blast2 test problem.

Table 24.2: 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 (ρL)</td>
</tr>
<tr>
<td>rho_mid</td>
<td>real</td>
<td>1</td>
<td>Initial density between the two interfaces (ρM)</td>
</tr>
<tr>
<td>rho_right</td>
<td>real</td>
<td>1</td>
<td>Initial density to the right of the right interface (ρR)</td>
</tr>
<tr>
<td>p_left</td>
<td>real</td>
<td>1000</td>
<td>Initial pressure to the left (pL)</td>
</tr>
<tr>
<td>p_mid</td>
<td>real</td>
<td>0.01</td>
<td>Initial pressure in the middle (pM)</td>
</tr>
<tr>
<td>p_right</td>
<td>real</td>
<td>100</td>
<td>Initial pressure to the right (pR)</td>
</tr>
<tr>
<td>u_left</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the left (uL)</td>
</tr>
<tr>
<td>u_mid</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) in the middle (uM)</td>
</tr>
<tr>
<td>u_right</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the right (uR)</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>
24.1.4 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

\[
p'_{0} = \frac{3(\gamma - 1)E}{(\nu + 1)\pi \delta r^{\nu}},
\]

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

\[
\rho = \rho_{1} = \frac{\gamma + 1}{\gamma - 1} \rho_{0}, \quad p = p_{1} = \frac{2}{\gamma + 1} \rho_{0} u^{2}, \quad v = v_{1} = \frac{2}{\gamma + 1} u,
\]

where \( u \equiv dR/dt \) is the speed of the shock wave. Near the center of the grid,

\[
\frac{\rho(\xi)}{\rho_{1}} \propto \xi^{\nu/(\gamma - 1)}, \quad \frac{p(\xi)}{p_{1}} = \text{constant}, \quad \frac{v(\xi)}{v_{1}} \propto \xi.
\]

Figure 24.7 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 24.7: 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 24.8 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 \texttt{PARAMESH} to refine grids using several different variables at once.

![Pressure field in the 2D Sedov explosion problem with 8 levels of refinement at \( t = 0.05 \).](image)

Figure 24.8: 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 \texttt{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 24.9. 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 \texttt{Sedov} problem are listed in Table 24.3. This problem is configured to use the perfect-gas equation of state (\texttt{Gamma}) with \texttt{gamma} set to 1.4. It is simulated in a unit-sized box.

### 24.1.4.1 Sedov Self-Gravity

Another variant of the Sedov problem is included in the release which runs with spherical coordinates in one dimension. The \texttt{Sedov} \texttt{Self-Gravity} problem also allows the effects of gravitational acceleration where the gravitational potential is calculated using the multipole solver. Figure 24.10 and 24.11 show the snapshots of the density profile and the gravitational potential at two different times during the evolution. The first snapshot is at \( t = 0.5 \) sec, when evolution is halfway through, while the second snapshot is at the end of the evolution, where \( t = 1.0 \) sec.
Figure 24.9: 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 24.3: 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 24.10: Snapshots of Sedov Self-gravity density profile and gravitational potential at time t=0.5 sec.

Figure 24.11: Snapshots of Sedov Self-gravity density profile and gravitational potential at time t=1.0 sec.
24.1.5 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) 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

\[
\begin{align*}
\delta u &= -y' \frac{\beta}{2\pi} \exp\left(\frac{1 - r^2}{2}\right) \\
\delta v &= x' \frac{\beta}{2\pi} \exp\left(\frac{1 - r^2}{2}\right) \\
\delta T^* &= -\frac{(\gamma-1)\beta}{8\gamma\pi^2} \exp\left(1 - r^2\right)
\end{align*}
\]

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

\[
\begin{align*}
T &= \frac{T_\infty}{T_\infty^*} \\
\rho &= \rho_\infty \left(\frac{T}{T_\infty}\right)^{1/\gamma}
\end{align*}
\]

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 \(n_{x,subint} \times n_{y,subint}\) subintervals. The runtime parameters for the isentropic vortex problem are listed in Table 24.4.

<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>n_{x,subint}</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (x)-direction</td>
</tr>
<tr>
<td>n_{y,subint}</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (y)-direction</td>
</tr>
</tbody>
</table>
Figure 24.12 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 $\rho = 0.95, 0.85, 0.75, \text{ and } 0.65$. 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 24.12: Density at $t = 0.0$ for the isentropic vortex problem. Shown are the initial condition and the exact solution at $t = 10.0, 20.0, \ldots$.](image)

A numerical simulation using the PPM scheme was run to illustrate such errors. The simulation used the same grid shown in Figure 24.12 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 $t = 10, 20, \ldots, \text{ etc.}$, the vortex should be back at its initial location.

Figure 24.13 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 24.14), and the minimum density is now 0.601786.

Figure 24.15 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.
Figure 24.13: Density at $t = 50.0$ for the isentropic vortex problem.

Figure 24.14: Density at $t = 100.0$ for the isentropic vortex problem.
Figure 24.15: Representative density profiles for the isentropic vortex near \( y = 0 \) at \( t = 0.0 \) (black), \( t = 50.0 \) (red), and \( t = 100.0 \) (blue).

24.1.6 Wind Tunnel With a Step

The problem of a wind tunnel containing a step, \texttt{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

\[
\rho_0 = 1.4, \quad p_0 = 1, \quad u_0 = 3, \quad \gamma = 1.4,
\]

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 \texttt{WindTunnel} problem are listed in Table 24.5. This problem is configured to use the perfect-gas equation of state (\texttt{Gamma}) with \texttt{gamma} set to 1.4. We also set \( xmax = 3, \ ymax = 1, \ Nblockx = 15 \), and \( Nblocky = 5 \) in order to create a grid with the correct dimensions. The version of \texttt{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 \texttt{REFLECTING} boundaries to the obstacle blocks. Finally, we use \texttt{xl\_boundary\_type} = "user" (\texttt{USER DEFINED} condition) and \texttt{xr\_boundary\_type} = "outflow" (\texttt{OUTFLOW} boundary) to instruct FLASH to use the correct boundary conditions in the \( x \)-direction. Boundaries in the \( y \)-direction are reflecting (\texttt{REFLECTING}).

Until \( t = 12 \), the flow is unsteady, exhibiting multiple shock reflections and interactions between different types of discontinuities. Figure 24.16 shows the evolution of density and velocity between \( t = 0 \) and \( t = 4 \).
Figure 24.16: 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 24.16: Density and velocity in the Emery wind tunnel test problem (continued).
Table 24.5: 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>

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 24.17 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 24.18 shows the detail of this structure at $t = 3$ 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.
Figure 24.17: Density at $t = 4$ in the Emery wind tunnel test problem, as computed with FLASH using several different levels of refinement.
24.1. HYDRODYNAMICS TEST PROBLEMS

24.1.7 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 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,

$$
\begin{align*}
\rho(x) & = \begin{cases} 
\rho_L & x \leq x_s \\
\rho_R (1 + a_\rho \sin(f_\rho x)) & x > x_s 
\end{cases} \\
\rho(x) & = \begin{cases} 
P_L & x \leq x_s \\
P_R & x > x_s 
\end{cases} \\
u(x) & = \begin{cases} 
u_L & x \leq x_s \\
u_R & x > x_s 
\end{cases}
\end{align*}
$$

(24.14)

where $a_\rho$ is the amplitude and $f_\rho$ is the frequency of the density perturbations. The gamma equation of state alternative implementation is used with gamma set to 1.4. The runtime parameters and their default values are listed in Table 24.6. The initial density, $x$-velocity, and pressure distributions are shown in Figure 24.19.

Figure 24.18: Detail of the Kelvin-Helmholtz instability seen at $t = 3$ in the Emery wind tunnel test problem for several different levels of refinement.

### Table 24.6: Runtime parameters used with the shu_osher test problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq x_s$</td>
<td>$\rho_L$</td>
<td>$\rho_R (1 + a_\rho \sin(f_\rho x))$</td>
<td>$\rho(x)$</td>
</tr>
<tr>
<td>$x &gt; x_s$</td>
<td>$\rho_R$</td>
<td>$\rho_R$</td>
<td>$\rho(x)$</td>
</tr>
<tr>
<td>$x \leq x_s$</td>
<td>$p_L$</td>
<td>$p_R$</td>
<td>$p(x)$</td>
</tr>
<tr>
<td>$x &gt; x_s$</td>
<td>$p_R$</td>
<td>$p_R$</td>
<td>$p(x)$</td>
</tr>
<tr>
<td>$x \leq x_s$</td>
<td>$u_L$</td>
<td>$u_R$</td>
<td>$u(x)$</td>
</tr>
<tr>
<td>$x &gt; x_s$</td>
<td>$u_R$</td>
<td>$u_R$</td>
<td>$u(x)$</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 24.20. 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.

Figure 24.21 shows 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.

### 24.1.8 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 24.7 shows values the runtime parameters values to control the amount of driving, and the Figures Figure 24.22 and Figure 24.23 show the density and x velocity profile of an xy plane in the center of the domain.

<table>
<thead>
<tr>
<th>Table 24.7: Runtime parameters used with the Driven Turbulence test problem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>st_stirmx</td>
</tr>
<tr>
<td>st_stirmin</td>
</tr>
<tr>
<td>st_energy</td>
</tr>
<tr>
<td>st_decay</td>
</tr>
<tr>
<td>st_freq</td>
</tr>
</tbody>
</table>
Figure 24.19: Initial density, $x$-velocity, and pressure for the Shu-Osher problem.
Figure 24.20: Density, $x$-velocity, and pressure for the reference solution at $t = 1.8$. 
24.1. HYDRODYNAMICS TEST PROBLEMS

Figure 24.21: Density fields on 400 and 200 mesh cells from the PPM scheme.

Figure 24.22: Density profile for the StirTurb driven turbulence problem.
24.1.9 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 $\Gamma = \frac{5}{3}$ for this problem and the left and right states are given:

\[
\begin{align*}
\rho_L &= \frac{10}{3} \\
p_L &= \frac{40}{3} \\
\rho_R &= 1.0 \\
p_R &= \frac{2}{3} \times 10^{-6}
\end{align*}
\]

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 24.8:

Figure 24.24, Figure 24.25, and Figure 24.26 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 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 24.24).
24.1. HYDRODYNAMICS TEST PROBLEMS

Figure 24.24: Density of numerical solution to the relativistic Sod problem at time $t = 0.36$.

Figure 24.25: Pressure of numerical solution to the relativistic Sod problem at time $t = 0.36$. 
Table 24.8: 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_rhoLeft</td>
<td>real</td>
<td>10</td>
<td>Initial density to the left of the interface ($\rho_L$)</td>
</tr>
<tr>
<td>sim_rhoRight</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>

24.1.10 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:

$$
(\rho, p, v_x, v_y) = \begin{cases} 
(0.5,1.0,0.0,0.0) & -1.0 \leq x < 0.0, \quad -1.0 \leq y < 0.0 \\
(0.1,1.0,0.0,0.99) & 0.0 \leq x \leq 1.0, \quad -1.0 \leq y < 0.0 \\
(0.1,1.0,0.99,0.0) & -1.0 \leq x < 0.0, \quad 0.0 \leq y \leq 1.0 \\
(\rho_1,p_1,0.0,0.0) & 0.0 \leq x \leq 1.0, \quad 0.0 \leq y \leq 1.0 
\end{cases}, \quad (24.16)
$$

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 24.27 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.
Figure 24.26: Normal velocity of numerical solution to the relativistic Sod problem at time \( t = 0.36 \).

Figure 24.27: 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.
24.1.11 Shafranov Shock

The Shafranov problem (Shafranov 1957) is a one-dimensional problem that provides a good verification for structure of shock waves in plasma. The idea is to solve the system of ODEs associated with a steady plasma shock numerically and use that as verification in FLASH4-alpha. The steady state solution of the ODE for $Z = 1$ and $Z = 54$ is provided in FLASH4-alpha and is used as input. As FLASH4-alpha advances the solution in time it maintains the steady state profile, moving with a specific speed (which can be computed numerically). This simulation verifies the implementation of Hydrodynamics (electron entropy advection), ion-electron equilibration and implicit conduction.

The setup for this problem is,

```
./setup ShafranovShock -auto -1d +pm4dev -parfile=flash.par (for Z = 1)
```

The numerical solution to the ODE is available under SimulationMain/ShafranovShock.
24.2 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`.

24.2.1 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 24.1.1). 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 24.28, Figure 24.29, Figure 24.30, Figure 24.31, and Figure 24.32. 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 monotonized 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 Swave MHD solver with a second-order MUSCL-Hancock scheme was used for the results shown in this simulation. The `StaggeredMesh` MHD solver can also be used for this Brio-Wu problem in one- and two-dimensions. However, in the latter case, the `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 `StaggeredMesh` scheme because the currently released version of the FLASH code does not truly support physically correct boundary conditions for this rotated shock geometry.

Figure 24.28: Density profile for the Brio-Wu shock tube problem.

24.2.1.1 Slowly moving shocks (SMS) issues in the Brio-Wu problem

Figure 24.33 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. 24.34, 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 24.35 – 24.38. The oscillation shedding found in the conventional PPM (e.g., Fig. 24.33) are significantly reduced in both density, and Mach number profiles. The overall qualitative solution behavior of using the upwind PPM approach, shown in Fig. 24.35, is very much similar to those from the 5th order WENO method as illustrated in Fig. 24.36. As shown in the closeup views in Fig. 24.37 and 24.38, 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.
24.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

24.2.2 Orszag-Tang MHD Vortex

The Orszag-Tang MHD vortex problem (Orszag and Tang, 1979), magnetoHD/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 $t = 0$

\[
V = V_0 \left( -\sin(2\pi y), \sin(2\pi x), 0 \right), \quad B = B_0 \left( -\sin(2\pi y), \sin(4\pi x), 0 \right), \quad (x, y) = \in [0, 1]^2; \quad (24.17)
\]

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 $p_0 = 1/\gamma$ 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 24.39 and Figure 24.40 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.
Figure 24.33: 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 24.34: 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 $B_y = 0, 0.2, 0.4, 0.6, 0.8, \text{ and } 1$, respectively. Severe numerical oscillations are evident in the solution using the conventional PPM reconstruction on 400 grid points.
Figure 24.35: Density (red) and Mach number (blue) at $t = 0.1$ of the Brio-Wu test using the upwind biased PPM limiter.

Figure 24.36: Density (red) and Mach number (blue) at $t = 0.1$ of the Brio-Wu test using the 5th order WENO scheme.
Figure 24.37: 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 24.38: 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.
The results were obtained using the directionally splitting 8\text{wave} MHD solver for this Orszag-Tang problem.

Figure 24.39: Density contours in the Orszag-Tang MHD vortex problem at $t = 0.5$.

Figure 24.40: Magnetic field contours in the Orszag-Tang MHD vortex problem at $t = 0.5$. 
24.2.3 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

\[
\rho(x, y) = \begin{cases} 
10 & r \leq r_0 \\
1 + 9f(r) & r_0 < r < r_1 \\
1 & r \geq r_1 
\end{cases} 
\]

(24.18)

\[
u(x, y) = \begin{cases} 
-f(r)u_0(y - 0.5)/r_0 & r \leq r_0 \\
-f(r)u_0(y - 0.5)/r & r_0 < r < r_1 \\
0 & r \geq r_1 
\end{cases} 
\]

(24.19)

\[
v(x, y) = \begin{cases} 
f(r)u_0(x - 0.5)/r_0 & r \leq r_0 \\
f(r)u_0(x - 0.5)/r & r_0 < r < r_1 \\
0 & r \geq r_1 
\end{cases} 
\]

(24.20)

\[
p(x, y) = 1 
\]

(24.21)

\[
B_x(x, y) = \frac{5}{\sqrt{4\pi}} 
\]

(24.22)

\[
B_y(x, y) = 0, 
\]

(24.23)

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 \(f(r) = (r_1 - r)/(r_0).\) 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 24.41 were obtained using the \textit{StaggeredMesh} MHD solver using 6 refinement levels. The divergence free evolution of the magnetic fields are well preserved as illustrated in Figure 24.42.

24.2.4 MHD Current Sheet

The two-dimensional current sheet problem, \textit{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 sides. We initialize two current sheets in the following:

\[
B_y = \begin{cases} 
B_0 & 0.0 \leq x < 0.5 \\
-B_0 & 0.5 \leq x < 1.5 \\
B_0 & 1.5 \leq x \leq 2.0 
\end{cases} 
\]

(24.24)

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 \(p = 0.1.\)
24.2. MAGNETOHYDRODYNAMICS TEST PROBLEMS

Figure 24.41: The Rotor problem at $t = 0.15$ (a) Density (b) Pressure (c) Mach number (d) Magnetic pressure.

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 24.43(a) – 24.43(f) on a $256 \times 256$ uniform grid. In Figure 24.44 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 24.42: Divergence of magnetic fields using the \texttt{StaggeredMesh} solver at $t = 0.15$ for the Rotor problem.
Figure 24.43: 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 24.44: Current density at $t = 4.0$ using the StaggeredMesh solver for the MHD CurrentSheet problem.
24.2.5 Field Loop

The 2D and 3D field loop advection problems (magnetohydrodynamics/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,

$$U = u_0(\cos \theta, \sin \theta, 1)$$  \hspace{1cm} (24.25)

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 $\nabla \cdot B = 0$ numerically in order to avoid any initial nonzero error in $\nabla \cdot B$. 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

$$A_z = \left\{ \begin{array}{ll} A_0(R - r) & r \leq R \\ 0 & \text{otherwise}. \end{array} \right.$$  \hspace{1cm} (24.26)

By using this initialization process, divergence-free magnetic fields, are constructed with a maximum value of $\nabla \cdot B$ in the order of $10^{-16}$ at the chosen resolution. The parameters in (24.27) are $A_0 = 10^{-3}$ and a field loop radius $R = 0.3$. This initial condition results in a very high plasma beta $\beta = p/B^2 = 2 \times 10^6$ 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 $A_{B_x}$ and $A_{B_y}$, 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 $w \nabla \cdot B \Delta t$ if a numerical scheme does not properly include multidimensional MHD terms.

From the results in Figure 24.45, 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. 24.46. 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. 24.46, 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.

24.2.6 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
Figure 24.45: The field loop advection problem using the \texttt{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 24.46: 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 is \(100/\sqrt{4\pi}\). 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 24.47, the explosion now becomes highly anisotropic as shown in the pressure plot in Figure 24.47. 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.

![Figure 24.47: The MHD blast test at time \(t = 0.01\) using the SWave solver. Pressure is highly anisotropic in \(x\) direction due to the strong initial magnetic field strength in \(x\) direction.](image)

### 24.3 Gravity Test Problems

#### 24.3.1 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

\[
k_J \equiv \sqrt{\frac{4\pi G \rho_0}{c_0^2}},
\]

and where \( c_0 \) is the unperturbed adiabatic sound speed

\[
c_0 = \sqrt{\frac{\gamma p_0}{\rho_0}} \tag{24.30}
\]

(Chandrasekhar 1961). If \( k > k_J \), the perturbation is stable and oscillates with frequency

\[
\omega = \sqrt{c_0^2 k^2 - 4\pi G \rho_0} ;
\]

otherwise, it grows exponentially, with a characteristic timescale given by \( \tau = (i\omega)^{-1} \).

We checked the dispersion relation (24.31) 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 \( \rho_0 = 1.5 \times 10^7 \text{ g cm}^{-3} \) and \( p_0 = 1.5 \times 10^7 \text{ dyn cm}^{-2} \), yielding \( k_J = 2.747 \text{ cm}^{-1} \). 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

\[
L = \frac{1}{2} \sqrt{\frac{\pi \gamma \rho_0}{G \rho_0^2}} \tag{24.32}
\]

This prevents roundoff errors at wavenumbers less than \( k_J \) from being amplified by the physical Jeans instability. We used wavevectors \( k \) 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 \( k \) are shown in Figure 24.48 together with the analytic solution, which is given in two dimensions by

\[
\begin{align*}
T(t) & = \frac{\rho_0 \delta^2 |\omega|^2 L^2}{8k^2} [1 - \cos(2\omega t)] \\
U(t) - U(0) & = -\frac{1}{8} \rho_0 c_0^2 \delta^2 L^2 [1 - \cos(2\omega t)] \tag{24.33} \\
W(t) & = -\frac{\pi G \rho_0^2 \delta^2 L^2}{2k^2} [1 + \cos(2\omega t)] .
\end{align*}
\]

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 24.49 compares the resulting dispersion relation to (24.31). It can be seen from this plot that FLASH correctly reproduces (24.31). 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 \( c_0, \Delta x, \) 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 24.48: 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.

The additional runtime parameters supplied with the Jeans problem are listed in Table 24.9. 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.
Figure 24.49: Computed versus expected Jeans dispersion relation (for stable modes) found using FLASH 1.62 with a six-level uniform grid.

Table 24.9: Runtime parameters used with the Jeans test problem.

<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 $\rho_{\text{ref}}$ 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 $\rho_{\text{ref}}$ 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 ($\rho_{\text{ref}}$). Density contrast is used to determine which blocks to refine; it is defined as $\max_{\text{block}} \left{ \frac{</td>
</tr>
</tbody>
</table>
24.3.2 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

\[
p_0 \ll \frac{4\pi G}{\gamma} \rho_0^2 r_0^2 .
\]  

(24.34)

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

\[
\left( \frac{8\pi G}{3} \rho_0 \right)^{1/2} t = \left( 1 - \frac{r(t)}{r_0} \right)^{1/2} \left( \frac{r(t)}{r_0} \right)^{1/2} + \sin^{-1} \left( 1 - \frac{r(t)}{r_0} \right)^{1/2}
\]

(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 cloud, which is where all of the matter ultimately ends up.

Results of a **DustCollapse** run using FLASH 3.0 appear in Figure 24.50, 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 \( \rho_0 = 10^9 \text{ g cm}^{-3} \) and \( r_0 = 6.5 \times 10^8 \text{ cm} \). In Figure 24.51a, the density, pressure, and velocity are scaled by \( 2.43 \times 10^9 \text{ g cm}^{-3} \), \( 2.08 \times 10^{17} \text{ dyn cm}^{-2} \), and \( 7.30 \times 10^3 \text{ cm s}^{-1} \), respectively. In Figure 24.51b they are scaled by \( 1.96 \times 10^{11} \text{ g cm}^{-3} \), \( 2.08 \times 10^{17} \text{ dyn cm}^{-2} \), and \( 2.90 \times 10^{10} \text{ cm s}^{-1} \). 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 (24.35).

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 (\texttt{gamma}) with \texttt{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 \texttt{Grid\_mark\_Refine\_Specialized} which refines blocks containing the center of the cloud.
CHAPTER 24. THE SUPPLIED TEST PROBLEMS

Figure 24.50: 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 24.51: 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.
24.3.3 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

\[ \rho(x, y) = \sum_{i=1}^{13} e^{-\sigma_i[(x-x_i)^2+(y-y_i)^2]} , \]  

(24.36)

where the constants \( \sigma_i, x_i, \) and \( y_i \) are given in Table 24.10. 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 24.52.

![Density field and block structure for a 14-level mesh applied to the Huang-Greengard PoisTest problem. The effective resolution of the mesh is 65,536².](image)

Table 24.10: 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>
<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 \( \text{sim.smlRho} \), the smallest allowed value of density. Runtime parameters from the Multigrid unit (both Gravity and GridSolvers) are relevant; see Section 8.10.2.3.
24.3.4 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-alpha 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-alpha \(\text{smirho}\) is used). The spheroid is motionless and in hydrostatic equilibrium. The gravitational potential of such object is analytically calculable, and is:

\[
\phi(x) = \pi G \rho \left[ 2A_1 a_1^2 - A_1 (x^2 + y^2) + A_3 (a_3^2 - z^2) \right], \tag{24.37}
\]

for a point inside the spheroid. Here

\[
A_1 = \frac{\sqrt{1 - e^2}}{e^3} \sin^{-1} e - \frac{1 - e^2}{e^2}, \tag{24.38}
\]

\[
A_3 = \frac{2}{e^2} - \frac{2\sqrt{1 - e^2}}{e^3} \sin^{-1} e, \tag{24.39}
\]

where \(e\) is the ellipticity of a spheroid:

\[
e = \sqrt{1 - \left(\frac{a_3}{a_1}\right)^2}. \tag{24.40}
\]

For a point outside the spheroid, potential is:

\[
\phi(x) = \frac{2a_3}{e^2} \pi G \rho \left[ a_1 \epsilon \tan^{-1} h - \frac{1}{2} \left( x^2 + y^2 \right) \left( \tan^{-1} h - \frac{h}{1 + h^2} \right) + 2z^2 (h - \tan^{-1} h) \right], \tag{24.41}
\]

where

\[
h = \frac{a_1 \epsilon}{\sqrt{a_3^2 + \lambda}}, \tag{24.42}
\]

and \(\lambda\) is the positive root of the equation

\[
\frac{x^2}{a_1^2 + \lambda} + \frac{y^2}{a_2^2 + \lambda} + \frac{z^2}{a_3^2 + \lambda} = 1. \tag{24.43}
\]

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 24.3.4. 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 \(\text{mpole\_lmax}\). 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:

\[
\epsilon = \left| \frac{\phi_{\text{analytical}} - \phi_{\text{FLASH}}}{\phi_{\text{analytical}}} \right| \tag{24.44}
\]

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 \(\ell\) in (8.15). In Figure 24.54–24.54 the gravitational potential for the MacLaurin spheroid, the FLASH4-alpha 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 24.54, 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
24.3. GRAVITY TEST PROBLEMS

Figure 24.53: Density of the MacLaurin spheroid (left X–Y plane, right Y–Z plane) with ellipticity $e = 0.9$. The FLASH4-alpha block structure is shown on top.

Table 24.11: Minimal and maximal relative error in all zones of the simulation, calculated using (24.44). 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>$L_{\text{in}}^2$ norm</th>
<th>$L_{\text{out}}^2$ 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>$1.4 \times 10^{-3}$</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>

odd-$l$ moments are zero: reasonable, physically motivated values for setting $\text{mpole.lmax}$ 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 24.54 – 24.56. 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 24.54: 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.
24.3. GRAVITY TEST PROBLEMS

Figure 24.55: 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 24.56: 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.
24.4 Particles Test Problems

These problems are primarily designed to test the functioning of the particle tracking routines within FLASH4-alpha.

24.4.1 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

$$|v| = \sqrt{\frac{2GM}{r_0}},$$

(24.45)

if a constant gravitational field due to a point mass $M$ at $(0.5L_x, 0.5L_y, 0.5L_z)$ is employed, or

$$|v| = \frac{1}{2} \sqrt{\frac{2G}{r_0}},$$

(24.46)

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 24.57 shows a typical pair of particle trajectories for this problem.

![Particle Trajectories](image)

Figure 24.57: 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 (*ext_field = .true.*), the field is assumed to be a central point-mass field (*physics/-Gravity/GravityMain/PointMass*), and the parameters for that unit should be assigned appropriate values. If the particles are self-gravitating (*ext_field = .false.*), the *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-alpha, 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 24.12** 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 <strong>GravityMain/Poisson</strong> should be used; in the latter, <strong>GravityMain/PointMass</strong>.</td>
</tr>
</tbody>
</table>

### 24.4.2 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
In addition, if desired.

(24.51) Figure 24.58 24.48

Using Cartesian coordinates in 1D, 2D, or 3D, with the pancake midplane tilted with respect to the coordinate axes if desired.

Table 24.13

The mean temperature \( \bar{T} \) is the corresponding Lagrangian coordinate, found by iteratively solving

\[
x_{c} = x_{\ell} - \frac{1 + z_{c}}{1 + z} \sin \frac{kx_{\ell}}{k}.
\]

The temperature solution is determined from the density under the assumption that the gas is adiabatic with ratio of specific heats \( \gamma \):

\[
T(x_{c}; z) = (1 + z)^{2} \frac{\rho(x_{c}; z_{f0})}{\rho(x_{c}; z)} \frac{\rho(x_{c}; z_{f0})}{\rho(x_{c}; z)} (1 + z)^{3/\gamma}.
\]

The mean temperature \( \bar{T}_{f0} \) is specified at a redshift \( z_{f0} \).

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

\[
\rho(x_{c}; z) \approx \bar{\rho} \frac{18}{(kx_{\ell})^{3}} (1 + z)^{3/\gamma}, \quad T(x_{c}; z) \approx \frac{\mu H_{0}^{2}}{6k_{B}k^{2}} (1 + z)(1 + z)^{2} (kx_{\ell})^{2}.
\]

At the midplane, which undergoes adiabatic compression, the comoving density and temperature are approximately

\[
\rho_{\text{center}} \approx \bar{\rho} \left[ \frac{1 + z_{f0}}{1 + z} \right]^{3} \frac{3H_{0}^{2} \mu}{k_{B}T_{f0}k^{2}} (1 + z_{c})^{4}(1 + z)^{3/\gamma} \quad \frac{1}{\gamma}
\]

\[
T_{\text{center}} \approx \frac{3H_{0}^{2} \mu}{k_{B}k^{2}} (1 + z)^{2}(1 + z_{c})^{4} \frac{\bar{\rho}}{\rho_{\text{center}}}.
\]

An example FLASH calculation of the post-caustic gas solution appears in Figure 24.58.

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 24.59.

The 2D pancake problem in FLASH4-alpha uses the runtime parameters listed in Table 24.13 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.
Figure 24.58: Example solution for the gas in a mixed particle/gas Zel’довich 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 24.59: Example solution for the dark matter in a mixed particle/gas Zel’довich pancake. Perturbation and cosmological parameters were the same as in Figure 24.58. 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 24.13: 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>zfiducial</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>
24.5 Burn Test Problem

24.5.1 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 (α,γ) and (γ,α) links among the 13 isotopes 4He, 12C, 16O, 20Ne, 24Mg, 28Si, 32S, 36Ar, 40Ca, 44Ti, 48Cr, 52Fe, and 56Ni. It is essential, however, to include (α,p)(p,γ) and (γ,p)(p,α) links in order to obtain reasonably accurate energy generation rates and abundance levels when the temperature exceeds \( \sim 2.5 \times 10^9 \) K. At these elevated temperatures the flows through the (α,p)(p,γ) sequences are faster than the flows through the (α,γ) channels. An (α,p)(p,γ) sequence is, effectively, an (α,γ) reaction through an intermediate isotope. In our α-chain reaction network, we include 8 (α,p)(p,γ) sequences plus the corresponding inverse sequences through the intermediate isotopes 27Al, 31P, 35Cl, 39K, 43Sc, 47V, 51Mn, and 55Co 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 \( 10^7 \) g cm\(^{-3}\), temperature of \( 2 \times 10^8 \) K, composition of pure carbon X(12C)=1, and material velocity of \( v_x = v_y = 0 \) 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 \( 4.236 \times 10^7 \) g cm\(^{-3}\), temperature of \( 4 \times 10^8 \) K, and composition of pure carbon X(12C)=1.
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 24.15.

<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 24.60 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 24.61 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 24.60: Steady-state conditions of the Cellular nuclear burn problem.
Figure 24.61: Close-up of the detonation front in steady-state for the Cellular nuclear burn problem.
24.6 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.

24.6.1 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. This simulation uses constant opacity of 10 cm$^{-1}$. Figure 24.62 shows the temperatures as a function of time.

![Ion/Electron/Radiation Temperature Comparison](image)

Figure 24.62: Temperatures as a function of time for the Infinite medium, Energy Equilibration simulation

24.6.2 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 24.63 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.
24.7 Other Test Problems

24.7.1 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 24.64). 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

$$ [n_i^Z]_{eq} = [n_{i+1}^Z]_{eq} \alpha_{i+1}^Z \quad (i = 1, ..., l_Z - 1) \quad (24.53) $$

$$ \sum_{i=1}^{l_Z} [n_i^Z]_{eq} = A_Z n_p \quad (24.54) $$

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, $T$, $\rho$, $\mathbf{v}$) are not updated. In practice, the continuity equations are solved with uniform density and velocity, while the momentum and energy equations are ignored.

Figure 24.65 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 24.64: Temperature profile assumed for the test.
Figure 24.65: 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 24.65: ... continued ...
Figure 24.66: As in Figure 24.65 for the solutions of the FLASH code.
Figure 24.66: ... 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 24.66 shows the population fraction vs. space after 700 s.

### 24.7.2 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 ?? for alternative ways of configuring the test problem, using either a flux-based or a standalone Diffuse interface.

### 24.7.3 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.

### 24.7.4 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 $n_b = 0.015n_c$ and $v_c = -0.2v_A$. 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 macroparticles thus represent fewer real ions than the core macroparticles. The number of magnetic field update subcycles is nine.
Figure 24.67: Temperature profile of the Delta-Function Heat Conduction Problem at two times. $t_{ini} = -1 \text{ ms}$, top: $t = 0 \text{ ms}$, bottom: $t = 2.46 \text{ ms}$.
24.7.4.1 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. 24.68 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 for a one-dimensional ion beam](image)

Figure 24.68: 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.

24.7.4.2 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. 24.69 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).
Figure 24.69: The magnetic field $y$-component for a two-dimensional ion beam at time $t = 77.85$ s $\approx 46 \Omega \cdot t^{-1}$. 
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.
Chapter 25

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/website/codesupport/visit/. Full instructions are also available at that site.
Chapter 26

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.

26.1 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.

26.2 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:
## CHAPTER 26. SERIAL FLASH OUTPUT COMPARISON UTILITY (SFOCU)

A: 2006-04-25/sod_2d_45deg_4levncmpi_chk_0001
B: 2005-12-14/sod_2d_45deg_4levncmpi_chk_0001

Min Error: $\text{inf}(|a-b| / \max(|a|+|b|, 1e-99))$
Max Error: $\text{sup}(|a-b| / \max(|a+b|, 1e-99))$
Abs Error: $\text{sup}|a-b|$
Mag Error: $\text{sup}|a-b| / \max(\text{sup}|a|, \text{sup}|b|, 1e-99)$

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>1.839e-14</td>
<td>0.302</td>
</tr>
<tr>
<td>temp</td>
<td>502</td>
<td>0</td>
<td>1.1e-11</td>
<td>8.56e-09</td>
</tr>
<tr>
<td>velx</td>
<td>516</td>
<td>0</td>
<td>2</td>
<td>5.985</td>
</tr>
<tr>
<td>vely</td>
<td>516</td>
<td>0</td>
<td>2</td>
<td>2</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>6.678e-12</td>
<td>7.3e+04</td>
<td>7.3e+04</td>
</tr>
<tr>
<td>ener</td>
<td>502</td>
<td>5.858e-12</td>
<td>8.43e+04</td>
<td>8.43e+04</td>
</tr>
<tr>
<td>gamc</td>
<td>0</td>
<td>4.85e+04</td>
<td>4.85e+04</td>
<td>4.85e+04</td>
</tr>
<tr>
<td>game</td>
<td>0</td>
<td>4.85e+04</td>
<td>4.85e+04</td>
<td>4.85e+04</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>6.678e-12</td>
<td>7.3e+04</td>
<td>7.3e+04</td>
</tr>
<tr>
<td>velx</td>
<td>516</td>
<td>3.45e-14</td>
<td>1.79e+04</td>
<td>1.79e+04</td>
</tr>
<tr>
<td>vely</td>
<td>516</td>
<td>1.05e-14</td>
<td>1.79e+04</td>
<td>1.79e+04</td>
</tr>
<tr>
<td>velz</td>
<td>0</td>
<td>3.45e+04</td>
<td>3.45e+04</td>
<td>3.45e+04</td>
</tr>
</tbody>
</table>

---

**FAILURE**

"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 27

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-alpha. 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.

27.1 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.

27.1.1 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

setenv IDL_PATH ${XFLASH3_DIR}:${IDL_DIR}:${IDL_DIR}/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.

### 27.1.2 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.

prompt> idl start_linux

### 27.2 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 27.1.

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-alpha. These sections are explained below.

#### 27.2.1 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.
27.2.1 File/Open prototype...

The Open prototype... menu option will bring up the file selection dialog box (see Figure 27.2). 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 27.3).

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.

27.2.2 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.
27.2.3 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.

27.2.4 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.

27.2.5 Plotting options available from the GUI

Various options are available on the xflash3 user interface to change the appearance of the plots.

27.2.5.1 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.

27.2.5.2 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.
27.2.5.3 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.

27.2.5.4 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.

27.2.5.5 Options

The options block allows you to toggle various options on/off. Table 27.1 lists the various options available.
Table 27.1: 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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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>

27.2.5.6 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.

27.2.5.7 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).

27.2.5.8 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.

27.2.6 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.

27.2.6.1 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 27.4). 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.

27.2.6.2 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 27.5). 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
27.2. XFLASH3: A WIDGET INTERFACE TO PLOTTING FLASH DATASETS

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.

27.2.6.3 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-alpha, this option is disabled in xflash3.

27.2.6.4 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.

27.2.6.5 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.

27.2.6.6 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.

27.2.6.7 Histogram Button
Create a histogram of the data. This option is disabled in this release of FLASH3.

27.2.6.8 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 27.6). Use the Close button to dismiss the results.
27.2.6.9 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.

27.3 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 27.5: The xflash3 velocity option subwidget.
Figure 27.6: The xfla$h$3 query widget, displaying information for a cell.
References

Bader, G. & Deuflhard, P. 1983, NuMat, 41, 373
Balsara, D. S., & Spicer, D. S. 1999, JCP, 149, 270
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
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

369
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
Forester, C. K. 1977, JCP, 23, 1
Gardiner, T., & Stone, J. 2005, JCP, 205, 509
Godunov, S. K. 1959, Mat. Sbornik, 47, 271
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
27.3. COMPARING TWO DATASETS


Lee, D. 2006, Ph.D. Dissertation, Univ. of Maryland

Lee, D. & Deane, A. 2008, JCP, in press

Li, S. 2005, JCP, 203, 344–357


LeVeque, R. J. 1997, JCP, 131, 327


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., Sonnendrucker, & Voß, U. 2000, JCP, 161, 484

Nadyozhin, D. K. 1974, Nauchnye informatsii Astron, Sov. USSR, 32, 33

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
Sod, G. 1978, JCP, 27, 1
Tóth, G. 2000, JCP, 161, 605
von Leer, B. 1979, JCP, 32, 101
Williams, F. A. 1988, Combustion Theory (Menlo Park: Benjamin-Cummings)
Williamson, J. H. 1980, JCP, 35, 48
Woodward, P. & Colella, P. 1984, JCP, 54, 115
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, 204, 207
- enucDtFactor, 210
- odeStepper, 204, 209
- useBurnTable, 209
- useShockBurn, 209

## Conductivity
- useConductivity, 248

## Cosmology
- CosmologicalConstant, 245
- MaxScaleChange, 245
- OmegaMatter, 245
- OmegaRadiation, 245

## Diffuse
- useDiffuse, 216
- useDiffuseTherm, 216
- useDiffuseVisc, 216

## Driver
- dtmax, 241
- meshCopyCount, 86
- nend, 142
- restart, 143
- tmax, 141

## Eos
- eintSwitch, 176, 183
- eos_coulumbMult, 202
- eos_maxNewton, 202
- eos_singleSpeciesA, 200, 201
- eos_singleSpeciesZ, 200, 201
- eos_tolerance, 202
- gamma, 200, 201, 272, 282, 288
- gammaEle, 200
- gammaIon, 200
- gammaRad, 200

## Gravity
- grav_boundary_type, 89

## Grid
- convertToConsvdForMeshCalls, 50, 51, 97, 101
- convertToConsvdInMeshInterp, 50, 51, 96, 97, 101
- derefine_cutoff_, 98
- eosMode, 182
- eosModeInit, 182
- flux_correct, 92
- geometry, 131
- gr_lrefineMaxRedDoByLogR, 99
- gr_lrefineMaxRedDoByTime, 99
- gr_lrefineMaxRedLogBase, 99
- gr_lrefineMaxRedRadiusFact, 99
- gr_lrefineMaxRedTimeScale, 99
- gr_lrefineMaxRefTRef, 99
- hy_eosModeAfter, 182
- iGridSize, 93
- interpol_order, 96
- jGridSize, 93
- kGridSize, 93
- lrefine_max, 99
- max_particles_per_blk, 98
- mpole_lmax, 324, 325
- mpole_subSample, 114
- Nblockx, 88, 288
- nblockx, 87
- Nblocky, 88, 288
- nblocky, 87
- Nblockz, 88
- nblockz, 87
- pfst_setupOnce, 111
- refine_cutoff_, 98
- refine_filter_, 98
- refine_on_particle_count, 98
- refine_var_, 97, 98
- smlrho, 324
- tagradius, 100
- xl_boundary_type, 88, 89, 288
- xmax, 87, 285, 288
- xmin, 87, 285
- xr_boundary_type, 288
- ymax, 87, 285, 288
- ymin, 87, 285
- zmax, 87
- zmin, 87

## Hydro
- chomboLikeUpdateSoln, 183

375
hybrid_riemann, 178
ppm_modifystates, 178
use_cma_flattening, 178
use_steepeening, 178

IO
alwaysComputeUserVars, 155
alwaysRestrictCheckpoint, 155
checkpointFileIntervalStep, 160
checkpointFileNumber, 143
chkGuardCellsInput, 149
chkGuardCellsOutput, 149
outputSplitNum, 155
particleFileName, 143
plot_grid_var_1, 148
plot_grid_var_2, 148
plot_var_1, 143
plot_var_2, 143
plotFileNumber, 161
plotfileNumber, 143

Ioniz
dneimax, 211
dneimin, 211
tneimax, 211
tneimin, 211

Logfile
log_file, 259

meshCopyCount, 253

Particles
particle_attribute_1, 240
particle_attribute_2, 240
pt_dtChangeToleranceDown, 235
pt_dtChangeToleranceUp, 235
pt_dtFactor, 240
pt_initialXMin, 240
pt_numX, 240
pt_numx, 160
pt_numy, 160
useParticles, 240

PhysicalConstants
pc_unitsBase, 170

RadTrans
rt_mgdBound_1, 253
rt_mgdBound_101, 253
rt_mgdFlCoeff, 253
rt_mgdFlMode, 253
rt_mgdNumGroups, 253
rt_useMGD, 253
rt_mgdBoundEntry, 253

Simulation
sim_maxTolCoeff0, 241
sim_maxTolCoeff1, 241
sim_maxTolCoeff2, 241
sim_schemeOrder, 241
sim_smlRho, 323

Stir
st_decay, 212
st_energy, 212

Timers
eachProcWritesSummary, 266
writeStatSummary, 266

Viscosity
useViscosity, 249
visc_whichCoefficientIsConst, 249
FLASH3 API

Driver
Driver_abortFlash, 79, 147, 161, 262
Driver_abortFlashC, 79
Driver_evolveFlash, 35, 38, 76, 216, 220, 232
Driver_finalizeFlash, 78
Driver_getDt, 78
Driver_getElapsedTime, 78
Driver_getNStep, 78
Driver_getSimTime, 78
Driver_initFlash, 19, 36, 76, 141, 160, 170
Driver_initParallel, 76
Driver_sendOutputData, 147

Grid
Grid_advanceDiffusion, 127
Grid_applyBCEdge, 90
Grid_applyBCEdgeAllUnkVars, 90
Grid_bcApplyToRegion, 90
Grid_bcApplyToRegionSpecialized, 90
Grid_fillGuardCells, 34, 91, 94, 102, 103, 106
Grid_getBlkBoundBox, 95
Grid_getBlkCenterCoords, 95, 132
Grid_getBlkCornerID, 95
Grid_getBlkData, 34, 102, 132
Grid_getBlkIndexLimits, 23, 102
Grid_getBlkPhysicalSize, 95, 132
Grid_getBlkPtr, 101
Grid_getBlkRefineLevel, 95
Grid_getBlkType, 95
Grid_getCellCoords, 23, 34, 95, 132
Grid_getDeltas, 132
Grid_getGeometry, 132
Grid_getListOfBlocks, 35, 102
Grid_getPlaneData, 102, 132
Grid_getPointData, 102, 132
Grid_getRowData, 102, 132
Grid_getSingleCellVol, 132
Grid_init, 87, 101
Grid_initDomain, 87, 101
Grid_mapParticlesToMesh, 224, 239
Grid_markRefineDerefine, 17, 99
Grid_markRefineSpecialized, 99
Grid_pfft, 111
Grid_pfftFinalize, 111

Grid_pfftInit, 111
Grid_putBlkData, 34, 102
Grid_restrictAllLevels, 4
Grid_sendOutputData, 103, 147
Grid_solvePoisson, 124
Grid_updateRefinement, 34, 35, 94, 103
Grid_getBlkPhysicalSize, 95

IO
IO_getPrevScalar, 147
IO_getScalar, 147
IO_init, 141, 161
IO_output, 141, 143, 155
IO_readUserArray, 147
IO_setScalar, 147, 150
IO_writeCheckpoint, 141, 155
IO_writeIntegralQuantities, 145, 240
IO_writeParticles, 145
IO_writePlotfile, 143, 155
IO_writeUserArray, 147

monitors
Logfile
Logfile_init, 259
Logfile_stamp, 262
Logfile_stampMessage, 262
Logfile_writeSummary, 263

Timers
Timers_create, 266
Timers_getSummary, 266
Timers_init, 266
Timers_reset, 266
Timers_start, 266
Timers_stop, 266

Multispecies
Multispecies_getAvg, 166
Multispecies_getProperty, 165
Multispecies_getSum, 165
Multispecies_getSumFrac, 166
Multispecies_getSumInv, 166
Multispecies_getSumSqr, 166
Multispecies_list, 166
Multispecies_setProperty, 165
Multispecies_unitTest, 167
Particles
   Particles_advance, 227, 232, 241
   Particles_computeDt, 232
   Particles_dump, 240
   Particles_init, 240
   Particles_initPositions, 239
   Particles_mapToMeshOneBlk, 239
   Particles_specifyMethods, 58, 238
   Particles_unitTest, 240
   Particles_updateAttributes, 240

PhysicalConstants
   PhysicalConstants_get, 170
   PhysicalConstants_init, 170
   PhysicalConstants_list, 170
   PhysicalConstants_listUnits, 170
   PhysicalConstants_unitTest, 170, 171

physics
   Cosmology
      Cosmology_cdmPowerSpectrum, 245
      Cosmology_computeDeltaCrit, 246
      Cosmology_computeDt, 245
      Cosmology_computeVariance, 246
      Cosmology_getRedshift, 244
      Cosmology_massToLength, 246
      Cosmology_redshiftToTime, 246
      Cosmology_solveFriedmannEqn, 245
   Diffuse
      Diffuse, 215, 216
      Diffuse_species, 215
      Diffuse_therm, 215, 344
      Diffuse_visc, 215
   Eos
      Eos, 51, 101, 195, 196, 199, 200, 202
      Eos_getData, 51
      Eos_init, 201
      Eos_putData, 51
      Eos_wrapped, 51, 52, 196, 200, 202
   Gravity
      Gravity_init, 89
      Gravity_potentialListOfBlocks, 224

RuntimeParameters
   RuntimeParameters_get, 160, 161
   RuntimeParameters_getPrev, 147
   RuntimeParameters_mapStrToInt, 88, 89
   RuntimeParameters_set, 161

Simulation
   Simulation_defineDomain, 87, 90, 275, 288
   Simulation_getVarnameType, 87
   Simulation_init, 19
   Simulation_initBlock, 101, 200
   Simulation_initSpecies, 164, 165, 201, 204
   Simulation_mapIntToStr, 58
Index

.dump_checkpoint, see IO
.dump_restart, see IO
subunit, 36
  ptherimary, 36
adiabatic mesh, 93
adiabatic indices, 196
angular coordinates, 87
block, 85
boundary conditions, 89
computational domain, 87
Config, 17, 18
double precision, 23
equation of state, 195, 196
  Eos, 202
  Eos_wrapped, 202
  gamma-law, 196
  Helmholtz, 196, 197
  multi-gamma, 196, 197
  rhd-gamma, 196
fidlr
  xflash3, 12
FIXEDBLOCKSIZE mode, 66, 71, 92
Flash.h, 65
flash.par, 11, 25, 143, 260
flux conservation, 94
geometry, see mesh
gmake
  gmake clean, 10
  Makefile.h, 10
    parallel build, 10
gmake
  gmake realclean, 10
grid
  boundary, see boundary conditions
data averaging, 94
  guardcells, see guardcells
  interpolation, 92, 94, 96, 130, 135
    order, 96
  prolongation, 96, 135
  restriction, 96, 135
  xmin/xmax, 87
  ymin/ymax, 87
  zmin/zmax, 87
Grid_getCellCoords, 23
  guardcells, 85
HDF5, see IO
I/O, 138
IDL, 9
IO
  .dump_checkpoint, 142
  .dump_particle_file, 145
  .dump_plotfile, 143
  .dump_restart, 142
  checkpoint file, 141
  forced Plotfile, 143
  HDF5, 149
  output file names, 148
  particle file, 144
  plotfile, 143
kernel, 37
Makefile.h, 39
MAXBLOCKS, 41, 94
memory, 93, 94
mesh
  geometry, 130
    CARTESIAN, 132
    Cartesian, 133
    CYLINDRICAL, 132
    cylindrical, 133
    POLAR, 132
    polar, 135
    SPHERICAL, 132
    spherical, 134
    refinement criteria, 21
NONFIXEDBLOCKSIZE mode, 66, 92, 93, 102
PARAMESH, 93
  paramesh, 86

379
Physical Constants, 169
  CGS, CGS, 169
  MKS, MKS, 169

sedov.dat, 12
sedov.log, 12

setup
+default, 46
- [123]d, 41
-auto, 41
-curvilinear, 43
-debug, 42
-defines, 43
-fbs, 43
-geometry, 43
-gridinterpolation, 43
-index-reorder, 44
-library, 45
-makefile, 44
-makehide, 44
-maxblocks, 41
-n [xyz]b, 41
-noclobber, 44
-nofbs, 43
-objdir, 42
-opt, 42
-os, 44
-parfile, 44
-particlemethods, 45
-portable, 45
-site, 45
-tau, 45
-test, 42
-unit, 42
-unitsfile, 45
-verbose, 40
-with-library, 45
-with-unit, 42
-without-library, 45
-without-unit, 46
shortcuts, 149, 155

setup, 10, 39
setup_params, 27

setups
  example, 17
  sedov, 10

shortcuts, 140
Simulation_data, 19
Simulation_init, 19
Simulation_initBlock, 17, 22
  creating new, 17

unit, 33

variable