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 ASC/Alliances Center for Astrophysical Thermonuclear Flashes (Flash Center) 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 CON-
TRIBUTORS 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:

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

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.
## Contents

**License**
- 0.1 Acknowledgments in Publication .................................................. i
- 0.2 Full License Agreement ................................................................. i

**1 Introduction**
- 1.1 What’s New in FLASH3 ................................................................. 1
- 1.2 What’s New in This Release ........................................................... 2
- 1.3 Known Issues in This Release ....................................................... 4
- 1.4 About the User’s Guide ................................................................. 4

**I Getting Started**

**2 Quick Start**
- 2.1 System requirements ................................................................. 9
- 2.2 Unpacking and configuring FLASH for quick start .......................... 10
- 2.3 Running FLASH ................................................................. 12

**3 Setting Up New Problems**
- 3.1 Creating a `Config` file ........................................................... 18
- 3.2 Creating a `Makefile` ................................................................. 18
- 3.3 Creating a `Simulation_data.F90` ................................................ 19
- 3.4 Creating a `Simulation_init.F90` .................................................. 19
- 3.5 Creating a `Simulation_initBlock.F90` ............................................. 21
- 3.6 The runtime parameter file (`flash.par`) ...................................... 25

**II FLASH Software System**

**4 Overview of FLASH architecture**
- 4.1 FLASH Inheritance ................................................................. 32
- 4.2 Unit Architecture ................................................................. 32
  - 4.2.1 Stub Implementations ............................................................. 33
  - 4.2.2 Subunits ............................................................................. 34
  - 4.2.3 Unit Data Modules, `init`, and `finalize` routines .................. 34
  - 4.2.4 Private Routines: kernels and helpers .................................. 35
- 4.3 Unit Test Framework ................................................................. 36

**5 The FLASH configuration script (`setup`)**
- 5.1 Basic Setup Options ................................................................. 40
- 5.2 Advanced Setup Options ............................................................. 43
- 5.3 Using Shortcuts ........................................................................ 46
- 5.4 Setup Variables and Preprocessing `Config` Files .......................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Config Files</td>
<td>48</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Configuration file syntax</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Creating a Site-specific Makefile</td>
<td>55</td>
</tr>
<tr>
<td>5.7</td>
<td>Files Created During the setup Process</td>
<td>56</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Informational files</td>
<td>56</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Code generated by the setup call</td>
<td>56</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Makefiles generated by setup</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>The Flash.h file</td>
<td>59</td>
</tr>
<tr>
<td>6.1</td>
<td>UNK, FACE(XYZ) Dimensions</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Property Variables, Species and Mass Scalars</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Fluxes</td>
<td>61</td>
</tr>
<tr>
<td>6.4</td>
<td>Scratch Grid Vars</td>
<td>62</td>
</tr>
<tr>
<td>6.5</td>
<td>Fluid Variables Example</td>
<td>62</td>
</tr>
<tr>
<td>6.6</td>
<td>Particles</td>
<td>63</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Particles Types</td>
<td>63</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Particles Properties</td>
<td>64</td>
</tr>
<tr>
<td>6.7</td>
<td>Other Preprocessor Symbols</td>
<td>64</td>
</tr>
<tr>
<td>III</td>
<td>Driver Unit</td>
<td>65</td>
</tr>
<tr>
<td>7</td>
<td>Driver Unit</td>
<td>67</td>
</tr>
<tr>
<td>7.1</td>
<td>Driver Routines</td>
<td>67</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Driver_initFlash</td>
<td>68</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Driver_evolveFlash</td>
<td>68</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Driver_finalizeFlash</td>
<td>69</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Driver accessor functions</td>
<td>69</td>
</tr>
<tr>
<td>IV</td>
<td>Infrastructure Units</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>Grid Unit</td>
<td>73</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview</td>
<td>75</td>
</tr>
<tr>
<td>8.2</td>
<td>GridMain Data Structures</td>
<td>76</td>
</tr>
<tr>
<td>8.3</td>
<td>Computational Domain</td>
<td>77</td>
</tr>
<tr>
<td>8.4</td>
<td>Boundary Conditions</td>
<td>78</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Boundary Condition Types</td>
<td>78</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Boundary Conditions at Obstacles</td>
<td>80</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Implementing Boundary Conditions</td>
<td>80</td>
</tr>
<tr>
<td>8.5</td>
<td>Uniform Grid</td>
<td>82</td>
</tr>
<tr>
<td>8.5.1</td>
<td>FIXEDBLOCKSIZE Mode</td>
<td>82</td>
</tr>
<tr>
<td>8.5.2</td>
<td>NONFIXEDBLOCKSIZE mode</td>
<td>82</td>
</tr>
<tr>
<td>8.6</td>
<td>Adaptive Mesh Refinement (AMR) Grid</td>
<td>83</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Additional Data Structures</td>
<td>85</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Grid Interpolation</td>
<td>86</td>
</tr>
<tr>
<td>8.6.3</td>
<td>Refinement</td>
<td>87</td>
</tr>
<tr>
<td>8.6.4</td>
<td>Nonpermanent Guard Cells</td>
<td>89</td>
</tr>
<tr>
<td>8.7</td>
<td>GridMain Usage</td>
<td>90</td>
</tr>
<tr>
<td>8.8</td>
<td>GridParticles</td>
<td>92</td>
</tr>
<tr>
<td>8.8.1</td>
<td>GridParticlesMove</td>
<td>92</td>
</tr>
<tr>
<td>8.8.2</td>
<td>GridParticlesMapToMesh</td>
<td>94</td>
</tr>
<tr>
<td>8.9</td>
<td>GridSolvers</td>
<td>98</td>
</tr>
<tr>
<td>8.9.1</td>
<td>Pfft</td>
<td>99</td>
</tr>
<tr>
<td>8.9.2</td>
<td>Poisson equation</td>
<td>101</td>
</tr>
</tbody>
</table>
## CONTENTS

8.9.3 Using the Poisson solvers ........................................... 108
8.10 Grid Geometry .......................................................... 109
  8.10.1 Understanding Curvilinear ...................................... 110
  8.10.2 Choosing a Geometry ............................................ 111
  8.10.3 Geometry Information in Code ................................. 111
  8.10.4 Available Geometries ........................................... 112
  8.10.5 Conservative Prolongation/Restriction on Non-Cartesian Grids 115
8.11 Unit Test ............................................................... 115

9 IO Unit ........................................................................... 117
  9.1 IO Implementations ..................................................... 119
  9.2 Output Files ............................................................. 121
    9.2.1 Checkpoint files - Restarting a Simulation .................. 121
    9.2.2 Plotfiles .......................................................... 123
    9.2.3 Particle files ..................................................... 124
    9.2.4 Integrated Grid Quantities – flash.dat ...................... 125
    9.2.5 General Runtime Parameters ................................. 126
  9.3 Restarts and Runtime Parameters .................................. 127
  9.4 Output Scalars ......................................................... 127
  9.5 Output User-defined Arrays ....................................... 128
  9.6 Output Grid Variables ............................................... 128
  9.7 Face-Centered Data ................................................... 128
  9.8 Output Filenames ....................................................... 128
  9.9 Output Formats ........................................................ 129
    9.9.1 HDF5 ............................................................. 129
    9.9.2 Parallel-NetCDF .................................................. 135
    9.9.3 Direct IO .......................................................... 135
    9.9.4 Output Side Effects ............................................. 136
  9.10 Working with Output Files ......................................... 136
  9.11 Unit Test ............................................................... 136
    9.11.1 Online tips for working with the IO Unit .................. 137

10 Runtime Parameters Unit ................................................ 139
  10.1 Defining Runtime Parameters ..................................... 139
  10.2 Identifying Valid Runtime Parameters ........................... 139
  10.3 Routine Descriptions ............................................... 140
  10.4 Example Usage ........................................................ 141

11 Multispecies Unit .......................................................... 143
  11.1 Defining Species ..................................................... 143
  11.2 Initializing Species Information in Simulation_initSpecies 144
  11.3 Routine Descriptions ............................................... 145
  11.4 Example Usage ........................................................ 146
  11.5 Unit Test ............................................................... 147

12 Physical Constants Unit .................................................. 149
  12.1 Available Constants and Units ................................. 150
  12.2 Applicable Runtime Parameters ................................ 150
  12.3 Routine Descriptions ............................................... 150
  12.4 Unit Test ............................................................... 151
## V Physics Units

### 13 Hydrodynamics Units
- **13.1 Gas hydrodynamics**
  - 13.1.1 Usage ........................................... 157
  - 13.1.2 The piecewise-parabolic method (PPM) .................. 157
  - 13.1.3 The unsplit hydro solver .......................... 158
- **13.2 Relativistic hydrodynamics (RHD)**
  - 13.2.1 Overview ....................................... 161
  - 13.2.2 Equations ........................................ 162
  - 13.2.3 Relativistic Equation of State ................... 162
  - 13.2.4 Additional Runtime Parameter .................... 162
- **13.3 Magnetohydrodynamics (MHD)**
  - 13.3.1 Description .................................... 163
  - 13.3.2 Usage ........................................... 164
  - 13.3.3 Algorithm: The Eight-wave Solver ................. 164
  - 13.3.4 Algorithm: The Unsplit Staggered Mesh Solver .... 166
  - 13.3.5 Non-ideal MHD .................................. 170

### 14 Equation of State Unit
- **14.1 Introduction** ..................................... 173
- **14.2 Gamma Law and Multigamma**
  - 14.2.1 Ideal Gamma Law for Relativistic Hydrodynamics .. 175
- **14.3 Helmholtz** ........................................ 175
- **14.4 Usage** ............................................ 177
  - 14.4.1 Initialization .................................... 177
  - 14.4.2 Runtime Parameters .............................. 177
  - 14.4.3 Direct and Wrapped Calls .................... 178
- **14.5 Unit Test** ......................................... 178

### 15 Local Source Terms
- **15.1 Burn Unit** ....................................... 181
  - 15.1.1 Algorithms ....................................... 181
  - 15.1.2 Reaction networks ............................... 182
  - 15.1.3 Detecting shocks ............................... 186
  - 15.1.4 Energy generation rates and reaction rates ....... 186
  - 15.1.5 Temperature-based timestep limiting ............ 187
- **15.2 Ionization Unit** .................................. 187
  - 15.2.1 Algorithms ....................................... 187
  - 15.2.2 Usage ........................................... 188
- **15.3 Stir Unit** .......................................... 189

### 16 Diffusive Terms
- **16.1 Diffuse Unit** ..................................... 191
  - 16.1.1 Diffuse Flux-Based implementations ............... 191
  - 16.1.2 Diffuse Split implementation ................... 193

### 17 Gravity Unit
- **17.1 Introduction** ..................................... 195
- **17.2 Externally Applied Fields**
  - 17.2.1 Constant Gravitational Field .................. 196
  - 17.2.2 Plane-parallel Gravitational field ............. 196
  - 17.2.3 Gravitational Field of a Point Mass ........... 196
- **17.3 Self-gravity** ..................................... 196
  - 17.3.1 Coupling Gravity with Hydrodynamics .......... 197
# CONTENTS

17.4 Usage ......................................................... 197
17.5 Unit Tests .................................................. 198

18 Particles Unit .............................................. 199
18.1 Time Integration .......................................... 201
  18.1.1 Active Particles .................................... 201
  18.1.2 Passive Particles ................................... 202
18.2 Mesh/Particle Mapping .................................... 206
  18.2.1 Quadratic Mesh Mapping ............................ 206
  18.2.2 Cloud in Cell Mapping ............................. 208
18.3 Using the Particles Unit .................................. 208
  18.3.1 Particles Runtime Parameters ..................... 210
  18.3.2 Particle Attributes ................................ 210
  18.3.3 Particle I/O ........................................ 210
  18.3.4 Unit Tests ........................................... 211

19 Cosmology Unit ............................................ 213
  19.1 Algorithms and Equations ............................. 213
  19.2 Using the Cosmology unit ............................. 215
  19.3 Unit Test ................................................ 216

20 Material Properties Units ................................. 217
  20.1 Thermal Conductivity .................................. 218
  20.2 Magnetic Resistivity ................................... 218
    20.2.1 Constant resistivity .............................. 218
  20.3 Viscosity ............................................... 218
  20.4 Mass Diffusivity ....................................... 218

VI Monitor Units .............................................. 219

21 Logfile Unit ................................................ 221
  21.1 Meta Data ............................................... 222
  21.2 Runtime Parameters, Physical Constants, and Multispecies Data ........................................ 222
  21.3 Accessor Functions and Timestep Data ................ 224
  21.4 Performance Data ...................................... 225
  21.5 Example Usage ......................................... 226

22 Timer and Profiler Units .................................. 227
  22.1 Timers ................................................... 227
    22.1.1 MPINative ......................................... 227
    22.1.2 Tau ................................................. 228
  22.2 Profiler ................................................ 229

VII Simulation Units ......................................... 231

23 The Supplied Test Problems ............................... 233
  23.1 Hydrodynamics Test Problems ......................... 233
    23.1.1 Sod Shock-Tube ................................... 234
    23.1.2 Variants of the Sod Problem in Curvilinear Geometries ........................................... 237
    23.1.3 Interacting Blast-Wave Blast2 ................. 238
    23.1.4 Sedov Explosion .................................. 242
    23.1.5 Isentropic Vortex ................................ 247
    23.1.6 Wind Tunnel With a Step ....................... 250
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1.7 Driven Turbulence StirTurb</td>
<td>255</td>
</tr>
<tr>
<td>23.1.8 Relativistic Sod Shock-Tube</td>
<td>257</td>
</tr>
<tr>
<td>23.1.9 Relativistic Two-dimensional Riemann</td>
<td>259</td>
</tr>
<tr>
<td>23.2 Magnetohydrodynamics Test Problems</td>
<td>260</td>
</tr>
<tr>
<td>23.2.1 Brio-Wu MHD Shock Tube</td>
<td>260</td>
</tr>
<tr>
<td>23.2.2 Orszag-Tang MHD Vortex</td>
<td>264</td>
</tr>
<tr>
<td>23.2.3 MHD Rotor</td>
<td>266</td>
</tr>
<tr>
<td>23.2.4 MHD Current Sheet</td>
<td>269</td>
</tr>
<tr>
<td>23.2.5 Field Loop</td>
<td>272</td>
</tr>
<tr>
<td>23.2.6 3D MHD Blast</td>
<td>272</td>
</tr>
<tr>
<td>23.3 Gravity Test Problems</td>
<td>273</td>
</tr>
<tr>
<td>23.3.1 Jeans Instability</td>
<td>273</td>
</tr>
<tr>
<td>23.3.2 Homologous Dust Collapse</td>
<td>277</td>
</tr>
<tr>
<td>23.3.3 Huang-Greengard Poisson Test</td>
<td>279</td>
</tr>
<tr>
<td>23.3.4 MacLaurin</td>
<td>280</td>
</tr>
<tr>
<td>23.4 Particles Test Problems</td>
<td>285</td>
</tr>
<tr>
<td>23.4.1 Two-particle Orbit</td>
<td>285</td>
</tr>
<tr>
<td>23.4.2 Zel'dovich Pancake</td>
<td>286</td>
</tr>
<tr>
<td>23.4.3 Modified Huang-Greengard Poisson Test</td>
<td>289</td>
</tr>
<tr>
<td>23.5 Burn Test Problem</td>
<td>289</td>
</tr>
<tr>
<td>23.5.1 Cellular Nuclear Burning</td>
<td>289</td>
</tr>
<tr>
<td>23.6 Other Test Problems</td>
<td>293</td>
</tr>
<tr>
<td>23.6.1 The non-equilibrium ionization test problem</td>
<td>293</td>
</tr>
<tr>
<td>23.6.2 The Delta-Function Heat Conduction Problem</td>
<td>298</td>
</tr>
<tr>
<td>23.6.3 The HydroStatic Test Problem</td>
<td>298</td>
</tr>
<tr>
<td>VIII Tools</td>
<td>301</td>
</tr>
<tr>
<td>23.7 VisIt</td>
<td>303</td>
</tr>
<tr>
<td>24 Serial FLASH Output Comparison Utility (sfocu)</td>
<td>305</td>
</tr>
<tr>
<td>24.1 Building sfocu</td>
<td>305</td>
</tr>
<tr>
<td>24.2 Using sfocu</td>
<td>305</td>
</tr>
<tr>
<td>25 FLASH IDL Routines (fidlr3.0)</td>
<td>307</td>
</tr>
<tr>
<td>25.1 Installing and Running fidlr3.0</td>
<td>307</td>
</tr>
<tr>
<td>25.1.1 Setting Up fidlr3.0 Environment Variables</td>
<td>308</td>
</tr>
<tr>
<td>25.1.2 Running IDL</td>
<td>308</td>
</tr>
<tr>
<td>25.2 xflash3: A Widget Interface to Plotting FLASH Datasets</td>
<td>308</td>
</tr>
<tr>
<td>25.2.1 File Menu</td>
<td>309</td>
</tr>
<tr>
<td>25.2.2 Defaults Menu</td>
<td>309</td>
</tr>
<tr>
<td>25.2.3 Colormap Menu</td>
<td>310</td>
</tr>
<tr>
<td>25.2.4 X/Y plot count Menu</td>
<td>310</td>
</tr>
<tr>
<td>25.2.5 Plotting options available from the GUI</td>
<td>310</td>
</tr>
<tr>
<td>25.2.6 Plotting buttons</td>
<td>312</td>
</tr>
<tr>
<td>25.3 Comparing two datasets</td>
<td>314</td>
</tr>
<tr>
<td>References</td>
<td>317</td>
</tr>
<tr>
<td>Runtime Parameters</td>
<td>321</td>
</tr>
<tr>
<td>API Index</td>
<td>323</td>
</tr>
<tr>
<td>Index</td>
<td>325</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The FLASH code is a modular, parallel multiphysics simulation code capable of handling general compressible flow problems found in many astrophysical environments. It is a set of independent code units put together with a Python language setup tool to form various applications. The code is written in FORTRAN90 and C. It uses the Message-Passing Interface (MPI) library for inter-processor communication and the HDF5 or PnetCDF library for parallel I/O to achieve portability and scalability on a variety of different parallel computers. FLASH3 has two interchangeable discretization grids: a Uniform Grid and a block-structured adaptive grid using the PARAMESH library, which places resolution elements only where they are needed most. The code’s architecture is designed to be flexible and easily extensible. Users can configure initial and boundary conditions, change algorithms, and add new physics units with minimal effort.

The Center for Astrophysical Thermonuclear Flashes, or 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 is 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 are 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. Additionally, the code needs to efficiently use the powerful, parallel supercomputing platforms made available through the ASC program.

In working towards these goals, 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, with more than 200 active users who customize the code for their own research.

1.1 What’s New in FLASH3

This is the official FLASH3 release and includes all the well tested capabilities of FLASH2. There are 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, 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 FLASH3 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, FLASH3 is a significant departure from FLASH2. Much 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 FLASH3 has a well defined interface and the rules for inheritance and encapsulation have been more clearly defined, enabling users to add new functionality to the code very simply. One of the largest achievements of FLASH3 has been the separation of the discretized
CHAPTER 1. INTRODUCTION

‘grid’ architecture from the actual physics. This untangling required changes in the deepest levels of the code but has added valuable new flexibility and capabilities allowing users to freely switch between a uniform and an adaptive grid. 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 FLASH3 are backwards compatible with FLASH2. The Flash code group has written extensive documentation detailing how to make the transition from FLASH2 to FLASH3 as smooth as possible. The user should follow the "Name changes from FLASH2 to FLASH3" link on the Code Support Web Page for help on transitioning to FLASH3.

More specific new features of FLASH3 include:

- Increased modularity through better encapsulation of code units.
- Clean public interfaces for each code unit.
- Support for a uniform grid.
- Ability to switch between uniform and adaptive Grid for same application.
- Support for face variables
- More comprehensive management of curvilinear coordinate systems and interpolation in the Grid.
- Use of masking in guardcell fill.
- Ability to use directionally unsplit time integration.
- Support for blocksizes not defined at compile time.
- Unit test framework and many unit tests.
- Enhanced setup script with seamless library management and the ability to suggest units a user might have neglected.
- New, more scalable algorithms for active and tracer particles interaction with the mesh.
- More robust second order time integration scheme for Lagrangian tracer particles.
- New multigrid solver for self gravity.
- New, directionally unsplit, staggered mesh MHD solver.
- New, directionally unsplit, gas hydrodynamics solver.
- Extensive web published API documentation for all units in the code.

1.2 What’s New in This Release

FLASH3.3 is a minor version change that adds several capabilities and resolves bugs and performance limitations found in the 3.2 release. This will also be the final release of FLASH3. The next major version change will include capabilities to simulate High Energy Density Physics experiments using FLASH.

Many of new features and capabilities included in the release relate to the unsplit Hydro and MHD implementations, and in the I/O unit. Several modifications in the I/O unit improve the performance of the unit, and resulted from collaborative work with Rob Latham at Argonne National Laboratory.

Capabilities added since the FLASH3.2 release include:
1.2. WHAT'S NEW IN THIS RELEASE

- New high-order scheme implementations (3rd order PPM and 5th order WENO) for the unsplit hydrodynamics and MHD solvers. A support for 6-guardcell in the WENO scheme.
- New Eulerian advection scheme for species and mass scalars for the two unsplit solvers.
- New implementation of interpolating two adiabatic indices (gamc and game), and gravity components in the Riemann states for the two unsplit solvers.
- New implementation of supporting a 2nd order predictor scheme in time for gravitational acceleration for the two unsplit solvers.
- New flux implementations for the two unsplit solvers: Marquina flux for both solvers; hybrid (HLLC + HLL) for the unsplit hydro solver.
- New high-order interpolation schemes in including transverse fluxes in the predictor step for the unsplit solvers.
- New Riemann state reconstruction implementation based on limiting characteristic variables for the split PPM solver.
- New implementation of hydrostatic boundary conditions (Dean Townsley)
- New implementation for thermal diffusion in uniform grids, based on pencil grid, implicit if diffusivity is constant.
- New runtime parameter to force a short timestep if necessary to reach tmax exactly.
- Ability to write a custom subset of particles to particle output files.
- Paramesh4dev option to "Avoid Orrery". This improves the scaling of PARAMESH regridding events. It is only available in pm4dev PARAMESH implementation and is switched on by default.
- New, more scalable default restriction method for the AMR mesh.

The modifications from version 3.2, many of which fix the bugs and improve the performance, include

- "Diffuse" code unit has been reorganized and its API is redefined.
- Coupling of gravity with the unsplit hydro solvers is more reliable and robust.
- The use of scratch variables has been reorganization to reduce the memory footprint.
- Isolated boundary conditions can be used in hybrid (Multigrid+Pfft) Poisson solver, thus in Pfft Poisson solver with Paramesh. Support for additional boundary condition types in Pfft Poisson solver (periodic, hom. Dirichlet, hom. Neumann in various combinations)
- Grid interpolation works correctly for more than 4 guard cells (should be even; 6 is tested and used)
- Obvious errors in timesteps such as drop to zero are trapped, so that FLASH doesn’t attempt to keep running when it shouldn’t.
- PARAMESH MPI calls that involve overlapping send and receive buffers have been fixed. There are no longer "memcpy argument memory ranges overlap" errors with new MPI library versions.
- HDF5 collective optimizations have been more thoroughly tested. The version in the last release was found to give silent errors when running on Blue Gene / P. The metadata caching bug thus discovered has been fixed in ROMIO and is included in mpich2-1.0.8 or higher, openmpi-1.4 or higher and incorporated in the V1R4 series of BlueGene drivers.
- HDF5 attributes are now created and written collectively when using a FLASH parallel I/O HDF5 implementation (See http://www.hdfgroup.org/HDF5/doc/RM/CollectiveCalls.html).
• Instead of a setup shortcut collective hdf5, usage of collective I/O optimizations with HDF5 is now fully controlled by useCollectiveHDF5 runtime parameter.

• Setup script keywords GRIDFACE, GRIDCENTERVAR, GRIDVAR have been renamed to SCRATCHFACE, SCRATCHCENTERVAR, SCRATCHVAR respectively.

• Several changes have been made in the API for Grid routines implemented in GridParticles subunit. Changes in argument lists due to reorganization in preparation for the new capabilities to be added to the code. Grid moveParticlesGlobal has been removed, use Grid moveParticles with regrid= .TRUE. instead.

1.3 Known Issues in This Release

• 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.

• 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.

1.4 About the User’s Guide

This User’s Guide is designed to enable individuals unfamiliar with the FLASH code to quickly get acquainted with its structure and to move beyond the simple test problems distributed with FLASH, customizing it to suit their own needs. Code users and developers are encouraged to visit the FLASH code documentation page for other references to 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 will 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://hdf.ncsa.uiuc.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 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 9.1. If IDL is not available, another visual analysis option is ViSit, described in Section 23.7.) 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 FLASH3.1.tar for FLASH version 3.1). This will create a directory called FLASHX.Y. Type ‘cd FLASHX.Y’ 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., 3 for X.Y = 3.1). 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 setup 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.

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
command-line option to use a different name for this file, described in the next section. Here we will create
a simple flash.par that sets a few parameters and allows the rest to take on default values. With your text
editor, edit the flash.par in the object directory so it looks like Figure 2.1.

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

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

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

Note the format of the file — each line is of the form variable = value, a comment (denoted by a hash
mark, #), or a blank. String values are enclosed in double quotes (". Boolean values are indicated in the
FORTRAN style, .true. or .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 setup.params located in the
object/ directory, which also includes brief comments for each parameter. If you wish to skip the creation of a
flash.par, a complete example is provided in the source/Simulation/SimulationMain/Sedov/ directory.
2.3 Running FLASH

We are now ready to run FLASH. To run FLASH on \( N \) processors, type

\[
\text{mpirun -np } N \text{ flash}X
\]

remembering to replace \( N \) and \( X \) 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 FLASH3 executable accepts an optional command-line argument for the runtime parameters file. If “-par file-name” is present, FLASH reads the file specified on command line for runtime parameters, otherwise it reads 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:

- **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.

- **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 gnuplot; note that the first line begins with a hash (#) and is thus ignored by gnuplot.

- **sedov hdf5 chk 000*” are the different checkpoint files. These are complete dumps of the entire simulation state at intervals of checkpointFileIntervalTime and are suitable for use in restarting the simulation.

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

\[
\text{setenv XFLASH3_DIR "<flash home dir>/tools/fidlr3.0"}
\]
\[
\text{setenv IDL_PATH "$\{XFLASH3_DIR\}:$IDL\{IDL_DIR\}:$\{IDL_DIR\}/lib"}
\]

where <flash home dir> is the location of the FLASH.X.Y directory. If you get a message indicating that IDL_PATH is not defined, enter

\[
\text{setenv IDL_PATH "$\{XFLASH3_DIR\}:$\{IDL_DIR\}:$\{IDL_DIR\}/lib"}
\]

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

**FLASH3 Transition**

Please note! The environment variable used with FLASH3 is `XFLASH3_DIR` instead of `XFLASH_DIR`. Similarly, the main routine name for interactive plotting has been changed from `xflash` to `xflash3`. There are some changes to the output formats in FLASH3, and a FLASH2 version of `xflash` will not read FLASH3 files. However, `xflash3` is partially backwards compatible and will read FLASH2 plot files, and checkpoint files without particle data. Due to the differences in the storage of particle data in output files in FLASH3, FLASH2 files with particle data must be read with `xflash2`. The change of name permits users to keep their FLASH2 environment intact, while also preventing accidental use of mismatched versions of FLASH and fidlr.

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

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

When the control panel settings are to your satisfaction, click the ‘Plot’ button to generate the plot. For Postscript or PNG output, a file is created in the current directory. The result should look something like Figure 2.3, although this figure was generated from a run with eight levels of refinement rather than the five used in the quick start example run. With fewer levels of refinement, the Cartesian grid causes the explosion to appear somewhat diamond-shaped. Please see Chapter 25 for more information about visualizing FLASH output with idl routines.

FLASH is intended to be customized by the user to work with interesting initial and boundary conditions. In the following sections, we will cover in more detail the algorithms and structure of FLASH and the sample problems and tools distributed with it.
Figure 2.2: The main xflash3 widget.
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 FLASH3/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 FLASH3, 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 FLASH3 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 FLASH3 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.

**FLASH3 Transition**

In FLASH2, the contents of the `if (.firstcall.)` clause are now in the `Simulation_init` routine in FLASH3.

The basic structure of the routine `Simulation_init` should consist of
1. Fortran module `use` statement for the `Simulation_data`

2. Fortran module `use` statement for the `Unit_interfaces` to access the interface of the `RuntimeParameters` unit, and any other units being used.

3. Variable typing `implicit none` statement

4. Necessary `#include` header files

5. Declaration of arguments and local variables.

6. Calls to the `RuntimeParameters` unit interface to obtain the values of runtime parameters.

7. Calls to the `PhysicalConstants` unit interface to initialize any necessary physical constants.

8. Calls to the `Multispecies` unit interface to initialize the species' properties, if there multiple species are in use

9. Initialize other unit scope variables, packages and functions

10. Any other calculations that are needed only once at the beginning of the run.

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

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

FLASH3 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(myPE)
    use Simulation_data
    use RuntimeParameters_interface, ONLY : RuntimeParameters_get
    implicit none
```
3.5 CREATING A SIMULATION_INITBLOCK.F90

#include "Flash.h"
#include "constants.h"

integer, intent(in) :: myPE

! 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 or 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, myPE)
  ! get the needed unit scope data
  use Simulation_data, ONLY: sim_posn, sim_xCos, sim_xCos, 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
  integer, intent(in) :: myPE

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

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

  ! array to get integer indices defining the beginning and the end 
  ! of a block.
```
integer, dimension(2,MDIM) :: blkLimits, blkLimitsGC

! the number of grid points along each dimension
integer :: sizeX,sizeY,sizeZ

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

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

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

FLASH3 Transition

FLASH3 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))
allocate(xRight(sizeX))
allocate(xCenter(sizeX))
allocate(yCoord(sizeY))
allocate(zCoord(sizeZ))

The next part of the routine involves setting up the initial conditions. This section could be code for
interpolating a given set of initial conditions, constructing some analytic model, or reading in a table of initial
values. In the present example, we begin by getting the coordinates for the cells in the current block. This
is done by a set of calls to 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_getCellCoods(IAXIS, blockID, LEFT_EDGE, gcell, xLeft, sizeX)
call Grid_getCellCoods(IAXIS, blockID, CENTER, gcell, xCenter, sizeX)
call Grid_getCellCoods(IAXIS, blockID, RIGHT_EDGE, gcell, xRight, sizeX)
call Grid_getCellCoods(JAXIS, blockID, CENTER, gcell, yCoord, sizeY)
call Grid_getCellCoods(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.
      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
         xxL = xxL/sim_xCos
         xxR = xxR/sim_xCos
      enddo
      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
      endif
   enddo
enddo

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!
velzZone = 0.5 * (sim_uLeft + sim_uRight) * sim_zCos

! initialize cells to the right of the initial shock.
else
  rhoZone = sim_rhoRight
  presZone = sim_pRight
  velxZone = sim_uRight * sim_xCos
  velyZone = sim_uRight * sim_yCos
  velzZone = sim_uRight * sim_zCos
endif
axis(IAXIS) = i  ! Get the position of the cell in the block
axis(JAXIS) = j
axis(KAXIS) = k

! Compute the gas energy and set the gamma-values
! needed for the equation of state.

ekinZone = 0.5 * (velxZone**2 + velyZone**2 + velzZone**2)
enerZone = presZone / (sim_gamma - 1.)
enerZone = enerZone / rhoZone
enerZone = enerZone + ekinZone
enerZone = max(enerZone, sim_smallP)

! store the variables in the current zone via the Grid_putPointData method

call Grid_putPointData(blockId, CENTER, DENS_VAR, EXTERIOR, axis, rhoZone)
call Grid_putPointData(blockId, CENTER, PRES_VAR, EXTERIOR, axis, presZone)
call Grid_putPointData(blockId, CENTER, VELX_VAR, EXTERIOR, axis, velxZone)
call Grid_putPointData(blockId, CENTER, VELY_VAR, EXTERIOR, axis, velyZone)
call Grid_putPointData(blockId, CENTER, VELZ_VAR, EXTERIOR, axis, velzZone)
call Grid_putPointData(blockId, CENTER, GAME_VAR, EXTERIOR, axis, sim_gamma)
call Grid_putPointData(blockId, CENTER, GAMC_VAR, EXTERIOR, axis, sim_gamma)

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.

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

# Density, pressure, and velocity on either side of interface
sim_rhoLeft = 1.
sim_rhoRight = 0.125
sim_pLeft = 1.
sim_pRight = 0.1
sim_uLeft = 0.
sim_uRight = 0.

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

# Gas ratio of specific heats
gamma = 1.4

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"

# 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.
Figure 3.1: Image of the initial distribution of density in example setup.
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.
FLASH3 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 uses the word “module” to refer exclusively to an F90 module, and the word “unit” for the basic FLASH code component. Also, unlike FLASH2, FLASH3 does not use F90 modules to implement units. Fortran’s limitation of one file per module is too restrictive for some of FLASH3’s units, which are too complex to be described by a single file. Instead, FLASH3 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_getBkData, and Grid_putBkData, 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 the way other units communicate. Code does not have to be rewritten if the users decides to implement the uniform grid instead of the adaptive grid.

4.2.1 Stub Implementations

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

```fortran
subroutine Grid_getListOfBlocks(blockType, listOfBlocks, count)
    the
    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.

FLASH3 Transition

While the concept of “null” or “stub” functions exists in FLASH2, FLASH3 has 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 FLASH3 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.

```fortran
subroutine Driver_evolveFlash()

use Driver_data, ONLY: dr_myPE, dr_numProcs, dr_nbegin, &
    dr_nend, dr_dt, dr_wallClockTimeLimit, &
```
4.2. UNIT ARCHITECTURE

```fortran
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.

**FLASH3 Transition**

From a user’s perspective, the most noticeable changes in architecture from FLASH2 to FLASH3 are related to the ownership and access to data. FLASH2 has a centralized database into which all the units (called modules in FLASH2) register their data when the code is initialized. Modules query the central database for their data needs, including for the data that they themselves registered. FLASH3, on the other hand has no central database. Instead, units own their data and provide accessor/mutator functions for the data they are willing to share. For example, the Grid unit owns the data structures which represent variables on the physical grid. Various physics units need to operate on and modify this data as the solution evolves. Therefore, the Grid unit provides a suite of get and put functions to access and modify the physical grid data. Similarly, time advancement related data such as Dt (timestep) and simTime (simulation time reached) belong to the Driver unit, but most units need to use them. The Driver unit provides get functions for Dt and simTime, but no put functions since no other unit should modify them.

Another change from FLASH2 is the introduction of the Unit_init files which among other initializations read the runtime parameters for a particular unit. In FLASH2 runtime parameters were initialized in if(.firstcall.) blocks from anywhere in the code. FLASH3 has streamlined these calls by having all of the runtime parameters relevant to a specific unit initialized once at the beginning of the simulation and stored in the Unit_data Fortran module.

### 4.3 Unit Test Framework

In keeping with good software practice, FLASH3 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, whereUnitTestName 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 14.5. 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 FLASH3 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
FLASH3 Transition

In FLASH2, the Units file was located in the FLASH root directory. In FLASH3, 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. FLASH3 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.

---

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>

Overides the default object directory with <dir>. Using this option allows you to have different simulations configured simultaneously in the FLASH3 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-\texttt{cartesian} geometry is chosen with the \texttt{-geometry} flag; so specifying \texttt{-curvilinear} only has an effect in the Cartesian case.

- **-defines=\langle\texttt{def}\rangle[,\langle\texttt{def}\rangle]...**
  \texttt{\langle\texttt{def}\rangle} is of the form \texttt{SYMBOL} or \texttt{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 \textit{e.g.}, \texttt{-defines=DEBUG\_ALL} turns on all debugging messages. Each unit may have its own \texttt{DEBUG\_UNIT} flag which you can selectively turn on.

- **\texttt{[-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=\langle\texttt{geometry}\rangle**
  Choose one of the supported geometries \texttt{cartesian}, \texttt{cylindrical}, \texttt{spherical}, or \texttt{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 \texttt{setup}-time. Choosing a non-Cartesian geometry here automatically sets the \texttt{-gridinterpolation=monotonic} option below.

- **-gridinterpolation=\langle\texttt{scheme}\rangle**
  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 \texttt{monotonic} scheme is required for curvilinear coordinates and is automatically enabled if a non-\texttt{cartesian} geometry is chosen with the \texttt{-geometry} flag. For AMR Grid implementations, this flag will automatically add additional directories so that appropriate data interpolation methods are compiled. The \texttt{monotonic} scheme is the default (by way of the \texttt{+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 FLASH3.0**

Note that the default interpolation behavior has changed as of the FLASH3 beta release: the \texttt{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 FLASH3. 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*
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 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]
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

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

can just type

```bash
./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

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

equivalent to

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

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

Some of the more commonly used shortcuts are described below:

### 5.4 Setup Variables and Preprocessing Config Files

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

```bash
./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/IMain/hdf5/Config file contains

```
DEFAULT serial
USESETUPVARS parallelIO
```

---

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>
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:

```
# 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.
5.5. CONFIG FILES

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

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

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

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

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

- **SUGGEST unitname unitname ...**
  Unlike REQUIRES, this keyword suggests that the current unit be used along with one of the specified units. The setup script will print details of the suggestions which have been ignored. This is useful in catching inadvertantly 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 \texttt{val} is a shortcut for \texttt{val ... val}.

For example

\begin{verbatim}
PARAMETER pres REAL 1.0 [ 0.1 ... 9.9, 25.0 ... ]
PARAMETER coords STRING "polar" ["polar","cylindrical","2d","3d"]
\end{verbatim}

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.

- \texttt{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.

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

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

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

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

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

In the current version of the code, the \texttt{TYPE} attribute is only used to determine which variables should be converted to conservative form for certain \texttt{Grid} operations that may require interpolation (\texttt{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
5.5. CONFIG FILES

51

type PER_MASS are converted: values are multiplied cell-by-cell with the value of the "dens" variable, and potential interpolation results are converted back by cell-by-cell division by "dens" values after interpolation.

Note that therefore

– variables types are irrelevant for uniform grids,
– variables types are irrelevant if neither convertToConsvdForMeshCalls nor convertToConsvdInMeshInterp is true, and
– variable types (and conversion to and from conserved form) only take effect if a VARIABLE dens ...

exists.

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

VARIABLE eint TYPE: PER_MASS EOSMAPIN: EINT

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

VARIABLE gamc EOSMAPOUT: GAMC

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

VARIABLE pres EOSMAP: PRES

has the same effect as

VARIABLE pres EOSMAPIN: PRES EOSMAPOUT: PRES

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

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

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

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

• 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 Eos_wrapped is called with an optional gridDataStruct argument of SCRATCH.
• **MASS\_SCALAR** name /\texttt{RENORM: group-name}/ {\texttt{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 \texttt{eosmap-spec}, see above under \texttt{VARIABLE}. An \texttt{eosmap-spec} for a MASS\_SCALAR may be used in an \texttt{Eos\_wrapped} invocation when the optional \texttt{gridDataStruct} argument is absent or has a value of CENTER.

---

\textbf{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 \texttt{Flash.h} file can distinguish between these two types of variables, many plotting routines cannot.

---

• **PARTICLETYPE** particle-type \texttt{INITMETHOD} initialization-method \texttt{MAPMETHOD} map-method

This keyword associates a particle type with mapping and initialization sub-units of \texttt{Particles} unit to operate on this particle type during the simulation. Here, \texttt{map-method} describes the method used to map the particle properties to and from the mesh (see Section 18.2), and \texttt{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 Parts\_Mapping and Parts\_Initialization subunits. None of the Parts\_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 \texttt{Particles.h}. The user may select any particle-type name, but the \texttt{map-method} and \texttt{initialization-method} must correspond to existing identifiers defined in \texttt{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 Parts\_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.

\texttt{PARTICLETYPE passive INITMETHOD lattice MAPMETHOD quadratic}

\texttt{REQUESTS Parts/ParticlesMain}

\texttt{REQUESTS Parts/ParticlesMapping/Quadratic}

\texttt{REQUESTS Parts/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:

  ```
  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**.

### 5.6 Creating a Site-specific Makefile

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

• Create the directory `sites/<hostname>`, where `<hostname>` is the hostname of your machine.

• Start by copying `os/<your os>/Makefile.h` to `sites/<hostname>`

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

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

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

---

**Actual Compiler or MPI wrapper?**

If you have **MPI** installed, you can either specify the actual compiler (e.g. `f90`) or the mpi-wrapper (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](#).
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>Description</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup_call</td>
<td>contains the options with which setup was called and the command line resulting after shortcut expansion</td>
</tr>
<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.
5.7. FILES CREATED DURING THE SETUP PROCESS

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

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.PhysicalConstants 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.
Chapter 6

The Flash.h file

Flash.h is a critical header file in FLASH3 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 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 \(^1\). However, in FLASH3 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.

\textbf{NPROP\_VARS}

The number of property variables in the simulation

\textbf{NSPECIES}

The total number of species in the simulation

\textbf{NMASS\_SCALARS}

The number of mass scalars in the simulation

\textbf{NUNK\_VARS}

The total number of quantities stored for each cell in the simulation. This equals \textbf{NPROP\_VARS} + \textbf{NSPECIES} + \textbf{NMASS\_SCALARS}

\textbf{PROP\_VARS\_BEGIN,PROP\_VARS\_END}

The indices in the unk array used for property variable data

\textbf{SPECIES\_BEGIN,SPECIES\_END}

The indices in the unk array used for species data

\textbf{MASS\_SCALARS\_BEGIN,MASS\_SCALARS\_END}

The indices in the unk array used for mass scalars data

\textbf{UNK\_VARS\_BEGIN,UNK\_VARS\_END}

The low and high indices for the unk array

The indices where specific properties (\textit{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 (\textit{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

\(^1\)See (13.9) for more information about mass scalars
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

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.

FLASH3 Transition

The Flash.h file defines many of the quantities that were previously stored in the database in FLASH2. In FLASH2, the user had to get an ‘integer’ key to access a variable stored in the unk data structure, but in FLASH3 we access the variable’s index directly using a #defined constant. For example, the density variable can be accessed from the unk data structure by unk(DENS_VAR,i,j,k,blockID). In a FIXEDBLOCKSIZE simulation unk has the dimensions:

unk(UNK_VARS_BEGIN:UNK_VARS_END,
    GRID_ILO_GC:GRID_IHI_GC,
    GRID_JLO_GC:GRID_JHI_GC,
    GRID_KLO_GC:GRID_KHI_GC,
    MAXBLOCKS)

The next example illustrates how data is stored in unk in FLASH2 and FLASH3:

FLASH2:

idens = dBaseKeyNumber('dens') ! get index of variable
call dBasePutData(idens, ......) ! make call to the routine

FLASH3:

call Grid_putPointData(blockID, CENTER, DENS_VAR, ......) !use index DENS_VAR

Note: In FLASH2 the user declared the number of species in a simulation in the Config file with the NSPECIES keyword. In FLASH3 the user only declares a species with the keyword SPECIES. The number of species is summed by the setup script as it parses the various Config files in the simulation. The NSPECIES keyword in no longer used in FLASH3 Config files.
The total number of quantities stored for each cell in the simulation. This equals \((NPROP_{FLUX} + NSPECIES_{FLUX} + NMASS_{SCALARS}_{FLUX})\)

The indices in the \texttt{fluxes} data structure used for property variable data

The indices in the \texttt{fluxes} data structure used for species data

The indices in the \texttt{fluxes} data structure used for mass scalars data

The first index for the \texttt{fluxes} data structure

### 6.4 Scratch Grid Vars

In FLASH3 the user is allowed to declare ‘scratch’ space for grid scope variables with the keyword \texttt{GRIDVAR} in the \texttt{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 \texttt{unk}. Scratch variables are indexed by postpending \texttt{_SCRATCH\_GRID\_VAR} to the capitalized four letter variable defined in the \texttt{Config} file. Similarly to property variables, \texttt{NSCRATCH\_GRID\_VARS}, \texttt{SCRATCH\_GRID\_VARS\_BEGIN}, and \texttt{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 \texttt{Config} file and parts of a corresponding \texttt{Flash.h} file.

```bash
# 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 RH0_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

FLASH3 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

```plaintext
PARTICLEPROP dens
PARTICLEPROP pres
PARTICLEPROP velx
```

then the relevant portion of `Flash.h` will contain

```c
#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 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.
Chapter 7

Driver Unit

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

7.1 Driver Routines

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

```fortran
program Flash
    implicit none

    call Driver_initFlash()
    call Driver_evolveFlash()
    call Driver_finalizeFlash()

end program Flash
```

![Figure 7.1: The Driver unit directory tree.](image-url)
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_initFlash. This 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 $t_{\text{max}}$, $n_{\text{end}}$ and $z_{\text{Final}}$ 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 $t_{\text{min}}$, $n_{\text{begin}}$, and $z_{\text{Initial}}$. 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 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.
### 7.1. DRIVER ROUTINES

#### FLASH3 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.

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

In FLASH3 the individual unit declares a runtime parameter that determines whether the unit is used during the simulation *e.g.*, the **Burn** unit declares `useBurn` within the **Burn** unit code that turns burning on or off. This way the **Driver** is no longer responsible for knowing what is included in a simulation. A unit gets called from the **Driver**, and if it is not included in a simulation, a stub gets called. If a unit, like **Burn**, is included but the user wants to turn burning off, then the runtime parameter declared in the **Burn** unit would be set to false.

#### 7.1.3 Driver_finalizeFlash

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

#### 7.1.4 Driver accessor functions

In FLASH3 the **Driver** unit also provides a number of accessor functions to get data stored in the **Driver** unit including `Driver_getDt`, `Driver_getNStep`, `Driver_getElapsedWCTime`, `Driver_getSimTime`.

---

### FLASH3 Transition

In FLASH3 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 FLASH3 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.

FLASH3 can interchangeably use either a uniform or adaptive grid for most problems. 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. The adaptive grid
currently supported in FLASH is a block-structured oct-tree based AMR package, PARAMESH. Both version
2 and version 3 are supported. By default, PARAMESH 4 is chosen when setting up an application, unless
another implementation is explicitly specified. The users are strongly encouraged to use PARAMESH 4, since
it is significantly better, and PARAMESH 2 is kept mostly to enable cross checking against FLASH2 results.

FLASH3 Transition
The following two commands will create the same (identical) application: a simulation of a Sod shock tube in 3 dimensions with PARAMESH 4 managing the grid.

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

However, if the command is changed to

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

the application is set up with a uniform grid instead. Additionally, because two different grids types are supported in FLASH3, the user must match up the correct IO alternative implementation with the correct Grid alternative implementation. Please see Chapter 9 for more details. Note that the setup script has capabilities to let the user set up shortcuts, such as "+ugio", which makes sure that the appropriate branch of IO is included when the uniform grid is being used. Please see Section 5.3 for more information. Also see grid tips for shortcuts useful for the Grid unit.

8.2 GridMain Data Structures
The Grid unit is the most extensive infrastructure unit in the FLASH code, and it owns data that most other units wish to fetch and modify. Since the data layout in this unit has implications on the manageability and performance of the code, we describe it in some detail here. First, unlike other units, Grid has two different F90 modules which contain the data variables. In common with the structure of other units, there is Grid_data.F90. Additionally, there is the physicaldata.F90 module, which contains the data types for all the physical-mesh based data. The split structure is necessary because FLASH shares the physicaldata.F90 module with PARAMESH. To lessen confusion, the same name physicaldata.F90 module is also defined in the UG (uniform grid) implementation.

FLASH can be run with a grid discretization that assumes cell-centered data, face-centered data, or a combination of the two. The data structure to store cell-centered physical data is a multidimensional F90 array named unk, short for “unknowns.” The face-centered variables are stored in arrays called facevarx, facevary, and facevarz, which contain the face-centered data along the x, y, and z dimensions, respectively. The cell-centered array unk is dimensioned as array(NUNK_VARS,nxb,nyb,nzb,blocks), where nxb, nyb, nzb are the spatial dimensions of a single block, and blocks is the number of blocks per processor (MAXBLOCKS for PARAMESH and 1 for UG). The face-centered arrays have one extra data point along the dimension they are representing, for example facevarx is dimensioned as array(NFACE_VARS,nxb+1,nyb,nzb,blocks). Some or all of the actual values dimensioning these arrays are determined at application setup time. The number of variables and the value of MAXBLOCKS are always determined at setup time. The spatial dimensions nxb,nyb,nzb can either be fixed at setup time, or they may be determined at runtime. These two modes are referred to as FIXEDBLOCKSIZE and NONFIXEDBLOCKSIZE. More details about this new capability in FLASH3 are available in Section 8.5.2. There is more extensive support for the cell-centered data structures since all physics solvers except the unsplit MHD method use only cell-centered data. And even the unsplit MHD solvers uses face-centered variables in a very limited way. For example, the face-centered variables
cannot be included in the plotfiles. They are included in the checkpoint file, and the plotfile support will be included in future.

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 stored in `unk`, 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 \((x_{min}, x_{max}), (y_{min}, y_{max}), \text{ and } (z_{min}, z_{max})\) runtime parameters. When working with curvilinear coordinates (see below in Section 8.10), 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.

FLASH3 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 `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 \(n_{blockx} \ast n_{blocky} \ast n_{blockz}\) top level blocks, where \(n_{blockx}, n_{blocky}, \text{ and } n_{blockz}\) are runtime parameters which default to 1. \(^2\) The resolution of the computational domain is usually very coarse and unsuitable for computation after the initial mapping. The `gr_expandDomain` routine remedies the situation by applying the refinement process to the initial domain until a satisfactory level of resolution is reached everywhere in the domain. This method of mapping the physical domain to computational domain is effective because the resultant resolution in any section is related to the demands of the initial conditions there.

\(^1\)Note that the term processor, as used here and elsewhere in the FLASH3 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.
FLASH3 Transition

FLASH2 supported only an AMR grid. 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 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 grid_createDomain is responsible for setting the neighbors of top-level blocks (to either other top-level blocks or to external boundary conditions) at initialization, after Nblockx × Nblocky × Nblockz root blocks have been created. Periodic (wrap-around) boundary conditions are initially configured in this routine as well. If periodic boundary conditions are set in the x-direction, for instance, the first blocks in the x-direction are set to have as their left-most neighbor the blocks that are last in the x-direction, and vice versa. Thus, when the guard cell filling is performed, the periodic boundary conditions are automatically maintained.

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

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

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

Although the GridBoundaryConditions subunit is included in a setup by default, it can be excluded (if no Config file “REQUIRES” it) by specifying -without-unit=Grid/GridBoundaryConditions. This will generally only make sense if all domain boundaries are to be treated as periodic. (All relevant runtime parameters x1_boundary_type etc. need to be set to “periodic” in that case.)

8.4.1 Boundary Condition Types

Boundary conditions are determined by the physical problem. Within FLASH, the parallel structure of blocks means that each processor works independently. If a block is on a physical boundary, the guard
cells are filled by calculation since there are no neighboring blocks from which to copy values. Bound-
aries are selected by setting runtime parameters such as \texttt{xl\_boundary\_type} (for the ‘left’ \textit{X}–boundary) to one of the supported boundary types (Table 8.1) in \texttt{flash.par}. Even though the runtime parameters for specifying boundary condition types are strings, the Grid unit understands them as defined integer con-
stants defined in the file \texttt{constants.h}, which contains all global constants for the code. The translation
from the string specified in “\texttt{flash.par}” to the constant understood by the Grid unit is done by the routine
\texttt{RuntimeParameters\_map\_Str\_ToInt}.

<table>
<thead>
<tr>
<th>\texttt{ab_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>
</tbody>
</table>
| hydrostatic-f2+nvrefl,  | Variants of \texttt{hydrostatic-f2}, where the normal velocity is han-
| hydrostatic-f2+nvout,   | dled specially in various ways, analogous to \texttt{reflect, outflow, |
| hydrostatic-f2+nvdiode   | and \texttt{diode} boundary conditions, respectively. See remark in text. |
| user-defined or user    | The user must implement the desired boundary behavior; see text. |

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

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

\begin{verbatim}
REQUIRES Grid/GridBoundaryConditions/Flash2HSE
\end{verbatim}

in the simulation directory’s \texttt{Config} file.

Note that the \texttt{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 \texttt{Gravity\_init}. This runtime parameter is separate from the \texttt{ab\_boundary\_type} ones interpreted by \texttt{GridBoundaryConditions}, and its recognized values are not the same (although there is some overlap).

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

Table 8.2: Additional boundary condition types recognized by FLASH. Boundary type \texttt{ab} may be replaced with \texttt{a=\{x,y,z\}} for direction and \texttt{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.
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.

The simple boundary condition types reflect, outflow, diode are implemented in the Grid_bcApplyToRegion.F90 file in Grid/GridBoundaryConditions. A users can add or modify the handling of some boundary condition types in a version of this file in the simulation directory, which overrides the regular version. There is, however, also the interface Grid_bcApplyToRegionSpecialized which by default is only provided as a stub and is explicitly intended to be implemented by users.

A Grid_bcApplyToRegionSpecialized implementation gets called before Grid_bcApplyToRegion and can decide to either handle a specific combination of boundary condition type, direction, grid data structure, etc., or leave it to Grid_bcApplyToRegion. These calls operate on a region of one block’s cells at a time. FLASH will pass additional information beyond that needed for handling simple boundary conditions to Grid_bcApplyToRegionSpecialized, in particular a block handle through which an implementation can retrieve coordinate information and access other information associated with a block and its cells.

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

Note that in all of these cases, i.e., whether boundary guard cell filling for a boundary type is implemented in Grid_bcApplyToRegion, Grid_bcApplyToRegionSpecialized, Grid_applyBCEdge, or Grid_applyBCEdgeAllUnkVars, the implementation does not fill guard cells in permanent data storage (the unk array and similar data structures) directly, but operates on buffers. FLASH3 fills some parts of the buffers with current values for interior cells before the call, and copies updated guardcell data from some (other) parts of the buffers back into unk (or similar) storage after the handling routine returns.

All calls to handlers for boundary conditions are for one face in a given dimension at a time. Thus for each of the IAXIS, JAXIS, and KAXIS dimensions there can be up to two series of calls, once for the left, i.e., “LOW,” and once for the right, i.e., “HIGH,” face. PARAMESH 4 makes additional calls for filling guard cells in edge and corner regions of blocks, these calls result in additional Grid_bcApplyToRegion* invocations for those cells that lie in diagonal directions from the block interior.
The boundary condition handling interfaces described so far can be implemented (and will be used!) independent of the Grid implementation chosen. At a lower level, the various implementations of Grid have different ways of requesting that boundary guard cells be filled. The GridBoundaryConditions subunit collaborates with GridMain implementations to provide to user code uniform interfaces that are agnostic of lower-level details. However, it is also possible — but not recommended — for users to replace a routine that is located deeper in the Grid unit. For PARAMESH 4, the most relevant routine is amr_1blk_bcsset.F90, for PARAMESH 2 it is tot_bnd.F90, and for uniform grid UG it is gr_bcApplyToAllBlks.F90.

8.4.3.1 Additional Concerns with PARAMESH 4

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

1. In other Grid implementations, filling of domain boundary guard cells is under control of the “user” (in this context, the user of the grid implementation, i.e., FLASH): These cells can be filled for all blocks at a time that is predictable to the user code, as a standard part of handling Grid_fillGuardCells, only. With PARAMESH 4, the user-provided amr_1blk_bcsset routine can be called from within the depths of PARAMESH on individual blocks (and cell regions, see below) during guard cell filling and at other times when the user has called a PARAMESH routine. It is not easy to predict when and in which sequence this will happen.

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

3. PARAMESH 4 does not let the user routine amr_1blk_bcsset operate on permanent storage (unk etc.) directly, but on (regions of) one-block buffers.

4. PARAMESH 4 occasionally invokes amr_1blk_bcsset to operate on regions of a block that belongs to a remote processor (and for which data has been cached locally). Such block data is not associated with a valid local blockID, making it more complicated for user code to retrieve metadata that may be needed to implement the desired boundary handling.

Some consequences of this for FLASH3 users:

- User code that implements boundary conditions for the grid inherits the requirement that it must be ready to be called at various times (when certain Grid routines are called).

- User code that implements boundary conditions for the grid inherits the requirement that it must operate on a region of the cells of a block, where the region is specified by the caller.

- Such user code must not assume that it operates on permanent data (in unk etc.). Rather, it must be prepared to fill guardcells for a block-shaped buffer that may or may not end up being copied back to permanent storage.

User code also is not allowed to make certain PARAMESH 4 calls while a call to amr_1blk_bcsset is active, namely those that would modify the same one-block buffers that the current call is working on.

- The user code must not assume that the block data it is acting on belongs to a local block. The data may not have a valid blockID. The code will be passed a “block handle” which can be used in some ways, but not all, like a valid blockID.

Caveat Block Handles

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

The FLASH3 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. PARAMESH 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

In previous versions, 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 we eliminate this constraint through an option at setup time. The two main reasons for this development are: 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 one of the patch-based AMR package with FLASH. Patch-based packages typically have different-sized block configurations at different times. In FLASH3, the new mode, called the “NONFIXEDBLOCKSIZE” mode, can currently be selected for use with Uniform Grid. To run an application in “NONFIXEDBLOCKSIZE”
mode, the “-nofbs” option must be used when invoking the setup tool; see Chapter 5 for more information.

For example:

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

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

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

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

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 
\[nvar \times (2 \times nguard + nxb) \times (2 \times nguard + nyb) \times (2 \times nguard + nzb)\] 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 × 64 × 64, being run on 8 processors.

`iprocs = 2`

`jprocs = 2`

`kprocs = 2`

`iGridSize = 64`

`jGridSize = 64`

`kGridSize = 64`

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

`iprocs = 4`

`jprocs = 2`

`kprocs = 1`

In this case, each processor will now have blocks of size 16 × 32 × 64.

### 8.6 Adaptive Mesh Refinement (AMR) Grid

We use a package known as PARAMESH (MacNeice et al. 1999) for implementing the adaptive mesh refinement (AMR) grid in FLASH. 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 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 nxb × nyb × nzb 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 × (2 * nguard + nxb) × (2 * nguard + nyb) × (2*nguard + nzb) × 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
8.6. ADAPTIVE MESH REFINEMENT (AMR) GRID

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).

becoming coarser), the new leaf nodes automatically have valid data. This averaging happens as an initial step in \texttt{Grid\_fillGuardCells} processing.

\texttt{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 \texttt{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

\texttt{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 \texttt{tree}. It includes the physical coordinate of a block's center, its physical size, level of refinement, and much more. These data structures also acts as temporary storage while updating refinement in the grid and moving the blocks. This metadata should in general not be accessed directly by application code. The \texttt{Grid} API contains subroutines for accessing the most important pars of this metadata on a block by block basis, like \texttt{Grid\_getBlkBoundBox}, \texttt{Grid\_getBlkCenterCoords}, \texttt{Grid\_getBlkPhysicalSize}, \texttt{Grid\_getBlkRefineLevel}, and \texttt{Grid\_getBlkType}.

\texttt{FLASH} has its own \texttt{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 \texttt{LEFT\_EDGE}, the \texttt{RIGHT\_EDGE} and the center of the cell. The coordinates are determined from \texttt{"cornerID"} which is also a part of this data structure.

The concept of \texttt{cornerID} is new in \texttt{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 \texttt{PARAMESH}. Another useful integer variable is the concept of a \texttt{stride}. A stride indicates the spacing factor between one cell and the cell directly to its right when calculating the \texttt{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 \(2^{\text{refine}_{-\text{max}}-n}\) where \(1 \leq n \leq \text{refine}_{-\text{max}}\).

The routine \texttt{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 8.10.5. PARAMESH supports various interpolation schemes, to which user-specified interpolation schemes can be added. FLASH3 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 FLASH3 supply their own default interpolation scheme, which is used when FLASH3 is configured with the `-gridinterpolation=native` setup option (see Section 5.2). The default schemes are only appropriate for Cartesian coordinates. If FLASH3 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 8.10.5 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.

FLASH3 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

\footnote{Particles 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.}
8.6. ADAPTIVE MESH REFINEMENT (AMR) GRID

VARIABLE dens TYPE: PER_VOLUME

in a Config file).

This behavior is available in both PARAMESH 2 and PARAMESH 4. It is enabled by setting the 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 FLASH3.1, 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.

FLASH3 Transition

FLASH3 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

$$E_i = \frac{|u_{i+1} - 2u_i + u_{i-1}|}{|u_{i+1} - u_i| + |u_i - u_{i-1}| + \epsilon |u_{i+1}| + 2 |u_i| + |u_{i-1}|},$$

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

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

where the sums are carried out over coordinate directions, and where, unless otherwise noted, partial derivatives are evaluated at the center of the $i_1i_2i_3$-th cell.
The estimator actually used in FLASH3’s default refinement criterion is a modification of the above, as follows:

\[ E_i = \left( \frac{\left| u_{i+2} - 2u_i + u_{i-2} \right|}{\left| u_{i+2} - u_i \right| + \left| u_i - u_{i-2} \right| + \epsilon \left| u_{i+2} \right| + 2 \left| u_i \right| + \left| u_{i-2} \right|} \right)^2, \tag{8.3} \]

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

\[ E_{ixiyiz} = \left( \frac{\sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \right)^2}{\sum_{pq} \left( \frac{1}{2 \Delta x_q} \left( \frac{\partial u}{\partial x_p} \bigg|_{i_p+1} + \frac{\partial u}{\partial x_p} \bigg|_{i_p-1} \right) + \epsilon \bar{u}_{pq} \right)^2} \right)^{1/2}, \tag{8.4} \]

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 \( iXijK \)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 \( E_{ixiyiz} \) 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 \( E_{ixiyiz} \) 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 derefinement involves deletion of all of a parent’s child blocks (\( 2^d \) blocks). As child blocks are created, they are temporarily placed at the end of the processor’s block list. After the refinements and derefinements are complete, the blocks are redistributed among the processors using a work-weighted Morton space-filling curve in a manner similar to that described by Warren and Salmon (1987) for a parallel treecode. An example of a Morton curve is shown in Figure 8.7.

During the distribution step, each block is assigned a weight which estimates the relative amount of time required to update the block. The Morton number of the block is then computed by interleaving the bits of its integer coordinates, as described by Warren and Salmon (1987). This reordering determines its location 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
8.6. ADAPTIVE MESH REFINEMENT (AMR) GRID

Figure 8.7: Morton space-filling curve for adaptive mesh grids.

region or a circular region in Cartesian coordinates, or can automatically refine by using some variable threshold, are available through the Grid\_markRefineSpecialized. It is intended to be called from the Grid\_markRefineDerefine routine. The interface works by allowing the calling routine to pick one of the routines in the suite through an integer argument. The calling routine is also expected to populate the data structure specs before making the call. A copy of the file Grid\_markRefineDerefine.F90 should be placed in the Simulation directory, and the interface file Grid\_interface.F90 should be used in the header of the function.

Warning

This collection of specialized refinement routines have had limited testing with FLASH3. 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 had added additional support in the standard implementation of refinement routine Grid\_markRefineDerefine for enforcing a maximum on refinement based on location or simulation time. These work by effectively lowering the absolute ceiling on refinement level represented by lrefine\_max. See the runtime parameters gr\_lrefineMaxRedDoByLogR, gr\_lrefineMaxRedRadiusFact, gr\_lrefineMaxRedDoByTime, gr\_lrefineMaxRedTimeScale, gr\_lrefineMaxRedTRef, and gr\_lrefineMaxRedLogBase.

8.6.4 Nonpermanent Guard Cells

PARAMESH 4 supports a NO\_PERMANENT\_GUARDCELLS mode, see the PARAMESH Users Guide for description.
Caveat

This release of FLASH3 provides mechanisms for using the NO_PERMANENT_GUARDCELLS mode. However, not all units are fully compatible with this mode. Users must exercise caution if they choose to use this mode. It is known to be very inefficient if a lot of data is moved around using the Grid unit’s get/put data functions. Also, Equation Of State must be applied to the guardcells every time a block is formulated since there is interpolation at fine-coarse boundaries.

8.7 GridMain Usage

The Grid unit has the largest API of all units, since it is the custodian of the bulk of the simulation data, and is responsible for most of the code housekeeping. The Grid_init routine, like all other Unit_init routines, collects the runtime parameters needed by the unit and stores values in the data module. If using UG, the Grid_init also creates the computational domain and the housekeeping data structures and initializes them. If using PARAMESH, the computational domain is created by the Grid_initDomain routine, which also makes a call to PARAMESH’s own initialization routine. The physical variables are all owned by the Grid unit, and it initializes them by calling the Simulation_initBlock routine which applies the specified initial conditions to the domain. If using an adaptive grid, the initialization routine also goes through a few refinement iterations to bring the grid to desired initial resolution, and then applies the Eos function to bring all simulation variables to thermodynamic equilibrium. Even though the mesh-based variables are under Grid’s control, all the physics units can operate on and modify them.

A suite of get/put accessor/mutator functions allows the calling unit to fetch or send data by the block. One option is to get a pointer Grid_getBlkPtr, which gives unrestricted access to the whole block and the client unit can modify the data as needed. The more conservative but slower option is to get some portion of the block data, make a local copy, operate on and modify the local copy and then send the data back through the “put” functions. The Grid interface allows the client units to fetch the whole block (Grid_getBlkData), a partial or full plane from a block (Grid_getPlaneData), a partial or full row (Grid_getRowData), or a single point (Grid_getPointData). Corresponding “put” functions allow the data to be sent back to the Grid unit after the calling routine has operated on it. Various getData functions can also be used to fetch some derived quantities such as the cell volume or face area of individual cells or groups of cells. There are several other accessor functions available to query the housekeeping information from the grid. For example Grid_getListOfBlocks returns a list of blocks that meet the specified criterion such as being “LEAF” blocks in PARAMESH, or residing on the physical boundary.

FLASH3 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 FLASH3 some of the data, such as simulation parameters like dt and simulation time are owned by the Driver unit instead of the Grid unit.
8.7. GRIDMAIN USAGE

FLASH3 Transition

In FLASH2, variables nxb, nyb and nzb traditionally described blocksize. In FLASH3, 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 are 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 GridGetBlkIndexLimits 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 GridFillGuardCells. 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 GridFillGuardCells routine makes sure that all the blocks get the right data in their guard cells 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 GridFillGuardCells 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. In FLASH3.1, GridFillGuardCells provides ways to

- operate on only a subset of the variables in unk (and facevarx, facevary, and facevarz), by masking out other variables;
- fill only some the nguard layers of guard cells that surround the interior of a block (while possibly excepting a “sweep” direction);
- combine guard cell filling with EOS calls (which often follow guard cell exchanges in the normal flow of execution of a simulation in order to ensure thermodynamical consistency in all cells, and which may also be very expensive), by letting GridFillGuardCells make the calls on cells where necessary;
- automatically determine masks and whether to call EOS, based on the set of variables that the calling code actually needs updated. by masking out other variables.

These options are controlled by OPTIONAL arguments, see GridFillGuardCells for documentation. When these optional arguments are absent or when a Grid implementation does not support them, FLASH falls back to safe default behavior which may, however, be needlessly expensive.

Another routine that may change the global state of the grid is GridUpdateRefinement. 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.

FLASH3 Transition

As mentioned in Chapter 4, FLASH3 allows every unit to identify scalar variables for checkpointing. In the Grid unit, the function that takes care of consolidating user specified checkpoint variable is GridSendOutputData. Users can select their own variables to checkpoint by including an implementation of this function specific to their requirements in their Simulation setup directory.
8.8 GridParticles

FLASH is primarily an Eulerian code, however, there is support for tracing the flow using Lagrangian particles. FLASH also uses active particles with mass in cosmological simulations. Each particle has an associated data structure, which contains fields such as its physical position and velocity, and mass in active particles. Depending upon the time integration 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 integration 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.

8.8.1 GridParticlesMove

FLASH3 has implementations of four different parallel algorithms for moving the particles data when they are displaced from their current block. The simplest algorithm UG Algorithm is applicably 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 two neighbors along one or more of its faces. The distribution of blocks based on morton ordering is an added complication since the neighboring blocks
8.8. GRIDPARTICLES

along a face may reside at a non-neighboring processor. The remaining three algorithms included in FLASH3 implement GridParticlesMove subunit for the adaptive mesh; two of the three algorithms, Point to Point and Sieve, move the particles data strictly between leaf nodes, while the third one, Perfect Tree Level, makes use of knowledge of the oct-tree. Of the three algorithms only the Sieve algorithm is able to move the data when the mesh refines. Thus even when a user opts for the PointToPoint of the PerfectTreeLevel implementations for moving particles with time evolution, some part of the Sieve implementation must necessarily be included to successfully move the data upon refinement.

8.8.1.1 UG Algorithm

The Uniform Grid algorithm for moving particles uses directional movement of data to minimize the number of communication steps. It implements 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.

8.8.1.2 Point To Point Algorithm

As a part of the data cached by Paramesh, there is wealth of information about the neighborhood of all the blocks on a processor. This information includes the processor and block number of all neighbors (face and corners) if they are at the same refinement level. If those neighbors are at lower refinement level, then the neighbor block is in reality the current block’s parent’s neighbor, and the parent’s neighborhood information is part of the cached data. Similarly, if the neighbor is at a higher resolution then the current blocks neighbor is in reality the parent of the neighbor. The parent’s metadata is also cached, from which information about all of its children can be derived. Thus it is possible to determine the processor and block number of the destination block for each particle. The PointToPoint implementation finds out the destinations for every
particles that is getting displaced from its block. Particles going to local destination blocks are moved first. The remaining particles are sorted based on their destination processor number, followed by a couple of global operations that allow every processor to determine the number of particles it is expected to receive from all of the other processors. A processor then posts asynchronous receives for every source processor that had at least one particle to send to it. In the next step, the processor cycles through the sorted list of particles and sends them to the appropriate destinations using synchronous mode of communication.

8.8.1.3 Sieve Algorithm

The Sieve algorithm does not concern itself with the configuration of the oct-tree 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 $numProc$s arithmetic meets the requirements. Here $MyPE$ is the local processor number and $numProc$s is the number of processors.

8.8.1.4 Perfect Tree Level Algorithm

PerfectTreeLevel implementation exploits the oct-tree structure to move the data when there is no refinement. The algorithm can be written in four simple steps.

1. identify particles leaving the block
2. move those particles up the oct-tree until they reach a level that is full (the level where all nodes exist)
3. At this point the grid is reduced to Uniform Grid, apply the algorithm described in Section 8.8.1.1.
4. move the newly arrived particles back down the tree to the appropriate leaves.

Since each block contains complete information about its parent, and children it is relatively easy to navigate the tree. Figure 8.10 shows the steps in the algorithm.

8.8.2 GridParticlesMapToMesh

FLASH3 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 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 FLASH3, 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 8.8.2.1 and Section 8.8.2.2, respectively.

**FLASH3 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:

```
-define=DEBUG_GRIDMAPPARTICLES
```
8.8.2.1 Uniform Grid

The Uniform Grid algorithm for accumulating particles' attribute on the grid is similar to the particle redistribution algorithm described in Section 8.8.1.1. 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.

8.8.2.2 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 8.8.1.3 and Section 8.8.1.2 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*(-1)^n+1 and receives from MyPE + n*(-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 Morton 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 8.8.1.2. 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 8.8.1.3 or Section 8.8.1.2.

The guard cell region decomposition described in Step 2 is illustrated in Figure 8.11. 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 8.11: 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 FLASH3 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. It is found that keeping a local copy of the entire PARAMESH tree structure on each processor is a large memory overhead, which restricts the size of active particle simulations. Therefore, by eliminating the memory overhead, FLASH3 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. Therefore, in FLASH3.1 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. Finally, this implementation (under PttoPt subdirectory) is a proof on concept, and is not well tested compared to the MoveSieve implementation described earlier. Also, we have no results showing the relative performance of each implementation.

8.9 GridSolvers

Figure 8.12: 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 8.9.1) and various solvers for the Poisson equation (Section 8.9.2).

8.9.1 Pfft

Pfft is a parallel frame work for computing a Fast Fourier Transform (FFT) on uniform grids. In the current release Pfft is available as a general utility for computing the FFTs. In future it will be combined with the Multigrid solver described below in Section 8.9.2.2 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 tranforms 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:

\[
N_1 \times N_2 \times N_3 / P, \tag{8.5}\]

where \( N_1 \times N_2 \times N_3 \) is the global data size and \( P \) is the number of processors. Here, the first transpose is local, while the second one is distributed. The internode communication is limited to one distributed transpose involving all the processors. However, there are two distinct disadvantages of this distribution of work:

- The size of the problem imposes an upper limit on the number of processors, in that the largest individual dimension is also the largest number of active processors. A three dimensional problem is forced to have modest individual dimensions to fit in the processor memory.
- As the machine size grows, the internode exchanges become long range, and the possibility of contention grows.

We have chosen a domain decomposition where each subdomain is a column of size

\[ N_1 \times N_2 / P_1 \times N_3 / P_2 \]
\[ P = P_1 \times P_2. \]  

With this distribution both the transposes are distributed in parallel. The data exchange along any one processor grid dimension is a collection of disjointed distributed transposes. Here, the contention and communication range is reduced, while the volume of data exchange is unaltered. The distributed transposes are implemented using collective MPI operation `alltoall`. In a slabwise distribution, the upper limit on the number of processors is determined by the smallest of \(< N_1, N_2, N_3 >\), where as in our distribution, the upper limit on the number of processors is the smallest of \(< N_1 \times N_2, N_2 \times N_3, N_1 \times N_3 >\).

### 8.9.1.1 Using Pfft

Pfft can only be used with a pencil grid, with the constraint that the number of processors along the IAXIS must be 1. This is because all one dimensional transforms are computed locally within a processor. However, FLASH contains a set of data movement subroutines that generate a usable pencil grid from any UG grid or any level of a PM grid. These routines are briefly explained in Section 8.9.1.2.

During the course of a simulation, Pfft can be used in two different modes. In the first mode, every instance of Pfft use will be exactly identical to every other instance in terms of domain size and the type of transforms. In this mode, the user can set the runtime parameter `pfft_setupOnce` to true, which enables the FLASH initialization process to also create and initialize all the data structures of Pfft. The finalization of the Pfft subunit is also done automatically by the FLASH shutdown process in this mode. However, if a simulation needs to use Pfft in different configurations at different instances of its use, then each set of calls to `Grid_pfft` for computing the transforms must be preceded by a call to `Grid_pfftInit` and followed by a call to `Grid_pfftFinalize`. In addition, the runtime parameter `pfft_setupOnce` should be set to false. A few other helper routines are available in the subunit to allow the user to query Pfft about the dimensioning of the domain, and also to map the Mesh variables from the `unk` array to and from Pfft compatible (single dimensional) arrays. Pfft also provides the location of wave numbers in the parallel domain; information that users can utilize to develop their own customized PDE solvers using FFT based techniques.

### 8.9.1.2 Pfft data movement subroutines

Mesh reconfiguration subroutines are available to generate a pencil grid for the Pfft unit. Two different implementations are available at `Grid/GridSolvers/Pfft/MeshReconfiguration/PtToPt` and `Grid/GridSolvers/Pfft/MeshReconfiguration/Collective`, with the PtToPt implementation being the default. Both implementations are able to generate an appropriate pencil grid in UG and PM mode. The pencil processor grid is automatically selected, but can be overridden by passing optional arguments to `Grid_pfftInit`. In UG mode they are invoked when the number of processors in the IAXIS of the FLASH grid is greater than one, and in PM mode they are always invoked. In PM mode they generate a pencil grid from a single level of the AMR grid, which may be manually specified or automatically selected as the maximum level that is fully-refined (i.e. has blocks that completely cover the computational domain at this level).

The pencil grid processor topology is stored in an MPI communicator, and the communicator may contain fewer processors than are used in the simulation. This is to ensure the pencil grid points are never distributed too finely over the processors, and naturally handles the case where the user may wish to obtain a pencil grid at a very coarse level in the AMR grid. If there are more blocks than processors then we are safe to distribute the pencil grid over all processors, otherwise we must remove a number of processors. Currently, we eliminate those processors that own zero FLASH blocks at this level, as this is a simple calculation that can be computed locally.

Both mesh reconfiguration implementations generate a map describing the data movement before moving any grid data. The map is retained between calls to the Pfft routines and is only regenerated when the grid changes. This avoids repeating the same global communications, but means communication buffers are left allocated between calls to Pfft.
In the Collective implementation, the map coordinates are used to specify where the FLASH data is copied into a send communication buffer. Two MPI_Alltoall calls then move 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_Irecv, and then traversing the FLASH list and sending block fragment using MPI_Isend. 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.

8.9.1.3 Unit Test

The unit test for Pfft solver solves the following equation:

\( \nabla^2(F) = -13.0 \times \cos 2x \times \sin 3y \)  \hspace{1cm} (8.7)

The simplest analytical solution of this equation assuming no constants is

\( F = \cos 2x \times \sin 3y \)  \hspace{1cm} (8.8)

We discretize the domain by assuming \( x_{min}, y_{min}, z_{min} = 0 \), and \( x_{max}, y_{max}, z_{max} = 2\pi \). 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.

8.9.2 Poisson equation

The GridSolvers subunit contains several different algorithms for solving the general Poisson equation for a potential \( \phi(x) \) given a source \( \rho(x) \)

\( \nabla^2 \phi(x) = \alpha \rho(x) \)  \hspace{1cm} (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.
8.9.2.1 Multipole Poisson solver

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. It currently works in 1D and 2D spherical, 2D axisymmetric cylindrical \((r,z)\), and 3D Cartesian and axisymmetric geometries. Because of the imposed symmetries, in the 1D spherical case, only the monopole term \((\ell = 0)\) makes sense, while in the axisymmetric and 2D spherical cases, only the \(m = 0\) moments are used (i.e., the basis functions are Legendre polynomials).

The multipole algorithm consists of the following steps. First, find the center of mass \(x_{\text{cm}}\)

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

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

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

The Green’s function for this equation satisfies the relationship

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

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

\[
r_< \equiv \min\{|x|, |x'|\}, \quad r_> \equiv \max\{|x|, |x'|\}.
\]

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

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

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

\[
\phi(x) = -\alpha \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell + 1} \left\{ Y_{\ell m}(\theta, \varphi) \times \right.
\]

\[
\left[ r_<^{\ell} \int_{r<r'} d^3 x' \frac{\rho(x') Y_{\ell m}^*(\theta', \varphi')}{r'^{\ell+1}} + \frac{1}{r^{\ell+1}} \int_{r>r'} d^3 x' \rho(x') Y_{\ell m}^*(\theta', \varphi') r'^{\ell} \right] \right\}.
\]

In practice, we carry out the first summation up to some limiting multipole \(\ell_{\text{max}}\). By taking spherical harmonic expansions about the center of mass, we ensure that the expansions are dominated by low-multipole terms, so that for a given value of \(\ell_{\text{max}}\), the error created by neglecting high-multipole terms is minimized. Note that the product of spherical harmonics in (8.15) is real-valued

\[
\sum_{m=-\ell}^{\ell} Y_{\ell m}^*(\theta', \varphi') Y_{\ell m}(\theta, \varphi) = \frac{2\ell + 1}{4\pi} \left[ P_{\ell 0}(\cos \theta) P_{\ell 0}(\cos \theta') + 2 \sum_{m=1}^{\ell} \frac{(\ell - m)!}{(\ell + m)!} P_{\ell m}(\cos \theta) P_{\ell m}(\cos \theta') \cos (m(\varphi - \varphi')) \right].
\]

Using a trigonometric identity to split up the last cosine in this expression and substituting for the inner
moment samples spaced a distance $\Delta$ apart in radius.

Because of the radial dependence of the multipole moments of the source function, these moments must be assumed to be cell-averaged quantities discretized on a block-structured mesh with varying cell size. Also, and where

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

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

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

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

$$\mu_{\ell m,q \geq q'}^{i} \leftarrow \mu_{\ell m,q \geq q'}^{i} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N^3} r_i^\ell p(x'_{i'j'k'}) \rho(x'_{i'j'k'}) P_{\ell m} (\cos \theta'_{i'j'k'}) \cos m \phi'_{i'j'k'}$$

(8.27)

$$\mu_{\ell m,q \leq q'}^{i} \leftarrow \mu_{\ell m,q \leq q'}^{i} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N^3} r_i^\ell p(x'_{i'j'k'}) \rho(x'_{i'j'k'}) P_{\ell m} (\cos \theta'_{i'j'k'}) \sin m \phi'_{i'j'k'}$$

(8.28)

$$\mu_{\ell m,q \leq q'}^{0} \leftarrow \mu_{\ell m,q \leq q'}^{0} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N^3} r_i^\ell p(x'_{i'j'k'}) \rho(x'_{i'j'k'}) P_{\ell m} (\cos \theta'_{i'j'k'}) \cos m \phi'_{i'j'k'}$$

(8.29)

$$\mu_{\ell m,q \leq q'}^{0} \leftarrow \mu_{\ell m,q \leq q'}^{0} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N^3} r_i^\ell p(x'_{i'j'k'}) \rho(x'_{i'j'k'}) P_{\ell m} (\cos \theta'_{i'j'k'}) \sin m \phi'_{i'j'k'}$$

(8.30)

where

$$q' = \left\lfloor \frac{|x'_{i'j'k'}|}{\Delta} \right\rfloor + 1$$

(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 ($\mu_{\ell m,q}^{0}$) 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

$$x'_{i'} = x_i + (i' - 0.5 N') \frac{\Delta x_i}{N'}$$

$$j'_{j'} = y_j + (j' - 0.5 N') \frac{\Delta y_j}{N'}$$

$$k'_{k'} = z_k + (k' - 0.5 N') \frac{\Delta z_k}{N'}$$

(8.32)

(8.33)

(8.34)

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

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

(8.35)

where the terms in the sum are evaluated via (8.17) up to the limiting multipole order $\ell_{\text{max}}$.

**FLASH3 Transition**

In FLASH2, two different parameters were available to control subsampling. In FLASH3, the sub-sampling is made consistent with the use of only one runtime parameter `mpole_subSample` to avoid excessive wasted computation in one section of the calculation.

The default value of `mpole_subSample` is 1; meaning no subsampling is performed by default and the code runs much faster. Should additional accuracy be required, increase this runtime parameter.
8.9.2.2 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. 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.

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:

\[
R(\mathbf{x}) \equiv 4\pi G \rho(\mathbf{x}) - \nabla^2 \tilde{\phi}(\mathbf{x}) .
\] (8.36)

This is accomplished through the application of the finite difference laplacian, defined on level \( \ell \) with length-scales \( \Delta x_\ell \), \( \Delta y_\ell \) and \( \Delta z_\ell \).

\[
D_\ell \tilde{\phi}^{bt}_{ijk} = \frac{1}{\Delta x_\ell^2} \left( \tilde{\phi}^{bt}_{i+1,j,k} - 2\tilde{\phi}^{bt}_{i,j,k} + \tilde{\phi}^{bt}_{i-1,j,k} \right) + \frac{1}{\Delta y_\ell^2} \left( \tilde{\phi}^{bt}_{i,j+1,k} - 2\tilde{\phi}^{bt}_{i,j,k} + \tilde{\phi}^{bt}_{i,j-1,k} \right) + \frac{1}{\Delta z_\ell^2} \left( \tilde{\phi}^{bt}_{i,j,k+1} - 2\tilde{\phi}^{bt}_{i,j,k} + \tilde{\phi}^{bt}_{i,j,k-1} \right) .
\] (8.37)
The restriction operator $\mathcal{R}_c$ for block interior zones $(i,j,k)$ is:

$$ (\mathcal{R}_c \hat{\phi})_{ijk} = \frac{1}{2d} \sum_{i',j',k'} \hat{\phi}^{i',j',k'}_{ijk}, \quad (8.39) $$

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

$$ (\mathcal{I}_c \hat{\phi})_{i'j'k'} = \sum_{p,q,r=-2}^{2} \alpha_{i'j'k'pqrs} \hat{\phi}_{i+p,j+q,k+r} \mathcal{P}(c), \quad (8.40) $$

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,

$$ \alpha_{1/2,j'k'pqrs} = \beta_{p} \gamma_{j'q} \gamma_{k'r} \quad (8.41) $$

$$ (\beta_{p}) = \left( \begin{array}{cccc} 1 & 7 & 7 & 12 \\ 12 & 12 & 12 & 12 \\ \end{array} \right) $$

$$ (\gamma_{j'q}) = \left\{ \begin{array}{c} -3 & 11 & 3 \\ \frac{11}{128} & 1 & \frac{11}{128} \\ \frac{3}{128} & \frac{11}{64} & \frac{3}{64} \\ \frac{11}{64} & \frac{11}{128} \end{array} \right\} \begin{array}{c} j' \text{ odd} \\ j' \text{ even} \end{array} \right. $$

Interpolation coefficients are defined analogously for the other faces. Note that we use half-integer zone indices to refer to averages over the faces of a zone; integer zone indices refer to zone averages.

### 8.9.2.3 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:

$$ G^\ell_{ijk} = -16\pi G \left[ \frac{1}{\Delta x^\ell} \sin^2 \left( \frac{i\pi}{2n_x} \right) + \frac{1}{\Delta y^\ell} \sin^2 \left( \frac{j\pi}{2n_y} \right) + \frac{1}{\Delta z^\ell} \sin^2 \left( \frac{k\pi}{2n_z} \right) \right]^{-1}. \quad (8.42) $$

However, to be able to use the block solver in a general fashion, we must be able to impose arbitrary boundary conditions per-block. In the case of nonhomogeneous Dirichlet boundary values, boundary value elimination may be used to generalize the solver. For instance, at the $-x$ boundary:

$$ \rho_{1jk} \rightarrow \rho_{1jk} - \frac{2}{\Delta x^\ell} \phi(x_{1/2}, y_j, z_k). \quad (8.43) $$

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:

$$ G^\ell_{ijk} = \begin{cases} -16\pi G \left[ \frac{1}{\Delta x^\ell} \sin^2 \left( \frac{(i-1)\pi}{n_x} \right) + \frac{1}{\Delta y^\ell} \sin^2 \left( \frac{(j-1)\pi}{n_y} \right) + \frac{1}{\Delta z^\ell} \sin^2 \left( \frac{(k-1)\pi}{n_z} \right) \right]^{-1} & \text{if } i, j, \text{ or } k \neq 1 \\ 0 & \text{if } i = j = k = 1 \end{cases} \quad (8.44) $$

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
8.9. GRID SOLVERS

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 $D_{ij} \tilde{\phi}_{ijk}^\ell = 4\pi G \rho_{ijk}^\ell$ for all blocks $b$ on level $\ell$.
   (c) Compute the residual $R_{ijk}^\ell = 4\pi G \rho_{ijk}^\ell - D_{ij} \tilde{\phi}_{ijk}^\ell$
   (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 $D_{ij} C_{ijk}^\ell = R_{ijk}^\ell$ for all blocks $b$ on level $\ell$.
   (c) Overwrite $R_{ijk}^\ell$ with the new residual $R_{ijk}^\ell - D_{ij} C_{ijk}^\ell$ for all blocks $b$ on level $\ell$.
   (d) Correct the solution on all leaf-node blocks $b$ on level $\ell$: $\tilde{\phi}_{ijk}^\ell \rightarrow \tilde{\phi}_{ijk}^\ell + C_{ijk}^\ell$.
   (e) For each block $b$ on level $\ell$ that has children, interpolate face boundary values of $C_{ijk}^\ell$ for each
       child.

5. If a correction step was performed, return to the residual propagation step.

The above procedure requires storage for $\tilde{\phi}$, $C$, $R$, and $\rho$ on each block, for a total storage requirement
of $4n_x n_y n_z$ values per block. Global communication is required in computing the tolerance-based stopping
 criterion.

8.9.2.4 Interfacing PFFT with Multigrid

We demonstrate a technique to improve the performance of the Multigrid solver 8.9.2.2. 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 8.9.2.2). 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 ($l_{\text{refine max}}=4$,
$l_{\text{refine max}}=5$ configuration), and 256 and 512 processors ($l_{\text{refine min}}=4$, $l_{\text{refine max}}=6$ configuration)
using the Collective mesh reconfiguration. For the $l_{\text{refine max}}=5$ configuration we reached a solution
in 43%(128 processors) less time and 35%(256 processors) less time than a pure Multigrid solution. For
the $l_{\text{refine max}}=6$ 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 \texttt{Pfft} time is actually spent moving data from a \texttt{FLASH} grid to a pencil grid, and then back again (The \texttt{PtToPt} mesh reconfiguration implementation has not been profiled yet). However, the time spent in the \texttt{Pfft} unit is relatively small (about a fifth of total Poisson solve time during the 512 processor run) compared to the \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{PtToPt} mesh reconfiguration implementation.

### 8.9.3 Using the Poisson solvers

The \texttt{GridSolvers} subunit solves the Poisson equation ((8.9)). Two different elliptic solvers are supplied with \texttt{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 \texttt{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 \texttt{Grid_solvePoisson} routine is the primary entry point to the Poisson solver module and has the following interface

\begin{verbatim}
call Grid_solvePoisson (iSoln, iSrc, bcTypes(6), bcValues(2,6), poisfact) ,
\end{verbatim}

where \texttt{iSoln} and \texttt{iSrc} are the integer-valued indices of the solution and source (density) variables, respectively. \texttt{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>\texttt{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, \texttt{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, \texttt{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 (\textit{e.g.}, in computing the gravitational acceleration), it is strongly recommended that you use the \texttt{quadratic} \texttt{cartesian/cylindrical/spherical} (quadratic) interpolants supplied by \texttt{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.
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>

8.9.3.1 Multipole

The poisson/multipole sub-module takes two runtime parameters, listed in Table 8.3. Note that storage and CPU costs scale roughly as the square of mpole_lmax, so it is best to use this module only for nearly spherical matter distributions.

8.9.3.2 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. 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:

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

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

```
Grid/GridSolvers/Multigrid
```

In which case the interface is as described above. The supported boundary conditions for the module are periodic, Dirichlet, given-value, and isolated. Due to the nature of the FFT block solver, the same type of boundary condition must be used in all directions. Therefore, only the value of bcTypes(1) will be considered in the call to Grid_solvePoisson.

The multigrid solver requires the use of two internally-used grid variables: isls and icor. These are used to store the calculated residual and solved-for correction, respectively. If it is used as a Gravity solver with isolated boundary conditions, then two additional grid variables, imgm and imgp, are used to store the image mass and image potential.

Table 8.4: Runtime parameters used with Grid/GridSolvers/Multigrid.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 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 8.5). 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.
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 FLASH3 also currently take geometry into account, see Section 8.6.3 above.

Table 8.5: 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cylindrical</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>y</td>
<td></td>
<td>r</td>
<td>z</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spherical</td>
<td>2</td>
<td>pm4,pm2,UG</td>
<td>y</td>
<td></td>
<td>r</td>
<td>θ</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>polar &quot;polar + z&quot; (cylindrical with a different ordering of coordinates)</td>
<td>3</td>
<td>—</td>
<td>n</td>
<td>r</td>
<td>ϕ</td>
<td>z</td>
</tr>
</tbody>
</table>

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 FLASH3, respectively. Only in the cartesian case do these correspond directly to their small-letter counterparts. For other geometries, the correspondence is given in Table 8.5.

8.10.1 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 FLASH3 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
8.10. GRID GEOMETRY

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 angle $\phi$ in these geometries implies the assumption of axial symmetry, and this is noted in Table 8.5. 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 $r_l = 0$, 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.

8.10.2 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.

8.10.3 Getting Geometry Information in Program Code

The Grid unit provides an accessor _Grid.getGeometry_ property that returns the geometry as an integer, which can be compared to the symbols \{CARTESIAN, SPHERICAL, CYLINDRICAL, POLAR\} defined in "constants.h" to determine which of the supported geometries we are using. A unit writer can therefore determine flow-control based on the geometry type (see Figure 8.14). Furthermore, this provides a mechanism for a unit to determine at runtime whether it supports the current geometry, and if not, to abort.

Coordinate information for the mesh can be determined via the _Grid.getCellCoords_ routine. This routine can provide the coordinates of cells at the left edge, right edge, or center. The width of cells
#include "constants.h"

integer :: geometry

call Grid_getGeometry(geometry)

select case (geometry)

case (CARTESIAN)
  ! do Cartesian stuff here ...

case (SPHERICAL)
  ! do spherical stuff here ...

case (CYLINDRICAL)
  ! do cylindrical stuff here ...

case (POLAR)
  ! do polar stuff here ...
end select

Figure 8.14: Branching based on geometry type

can be determined via the Grid_getDeltas routine. Angle values and differences are given in radians. Coordinate information for a block of cells as a whole is available through Grid_getBlkCenterCoords and Grid_getBlkPhysicalSize.

The volume of a single cell can be obtained via the Grid_getSingleCellVol or the Grid_getPointData routine. Use the Grid_getBlkData, Grid_getPlaneData, or Grid_getRowData routines with argument dataType=CELL_VOLUME To retrieve cell volumes for more than one cell of a block. To retrieve cell face areas, use the same Grid_get*Data interfaces with the appropriate dataType argument.

Note the following difference between the two groups of routines mentioned in the previous two paragraphs: the routines for volumes and areas take the chosen geometry into account in order to return geometric measures of physical volumes and faces (or their RD equivalents). On the other hand, the routines for coordinate values and widths return values for $X$, $Y$, and $Z$ directly, without converting angles to (arc) lengths.

8.10.4 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.
8.10.4.1 Cartesian geometry

FLASH uses Cartesian (plane-parallel) geometry by default. This is equivalent to specifying

\texttt{geometry = "cartesian"}

in the runtime parameter file.

\textit{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>

8.10.4.2 Cylindrical geometry

To run FLASH with cylindrical coordinates, the \texttt{geometry} parameter must be set thus:

\texttt{geometry = "cylindrical"}

\textit{Cell Volume in Cylindrical Coordinates}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>$\pi (r_2^2 - r_1^2)$</td>
</tr>
<tr>
<td>2-d</td>
<td>$\pi (r_2^2 - r_1^2) \Delta z$</td>
</tr>
<tr>
<td>3-d</td>
<td>$\frac{1}{2} (r_2^2 - r_1^2) \Delta z \Delta \phi$</td>
</tr>
</tbody>
</table>

As in other non-cartesian geometries, if the minimum radius is chosen to be zero ($x_{\text{min}} = 0.$), the left-hand boundary type should be reflecting. Of all supported non-cartesian geometries, the cylindrical is in 2-d most similar to a 2-d coordinate system: it uses two linear coordinate axes ($r$ and $z$) that form a rectangular grid physically as well as logically.

As an illustrative example of the kinds of considerations necessary in curved coordinates, Figure 8.15 shows a jump in refinement along the cylindrical ‘$z$’ direction. When performing the flux correction step at a jump in refinement, we must take into account the area of the annulus through which each flux passes to do the proper weighting. We define the cross-sectional area through which the $z$-flux passes as

$$A = \pi (r_2^2 - r_1^2).$$

The flux entering the coarse cell above the jump in refinement is corrected to agree with the fluxes leaving the fine cells that border it. This correction is weighted according to the areas

$$f_3 = \frac{A_1 f_1 + A_2 f_2}{A_3}.$$  \hspace{1cm} (8.46)

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.

8.10.4.3 Spherical geometry

One or two dimensional spherical problems can be performed by specifying

\texttt{geometry = "spherical"}

in the runtime parameter file.
Figure 8.15: Diagram showing two fine cells and a coarse cell at a jump in refinement in the cylindrical ‘z’ direction. The block boundary has been cut apart here for illustrative purposes. The fluxes out of the fine blocks are shown as \( f_1 \) and \( f_2 \). These will be used to compute a more accurate flux entering the coarse flux \( f_3 \). The area that the flux passes through is shown as the annuli at the top of each fine cell and the annulus below the coarse cell.

**Cell Volume in Spherical Coordinates**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>( \frac{4}{3}\pi(r^3_r - r^3_l) )</td>
</tr>
<tr>
<td>2-d</td>
<td>( \frac{2}{3}\pi(r^2_r - r^2_l)(\cos(\theta_l) - \cos(\theta_r)) )</td>
</tr>
<tr>
<td>3-d</td>
<td>( \frac{1}{3}(r^3_r - r^3_l)(\cos(\theta_l) - \cos(\theta_r))\Delta\phi )</td>
</tr>
</tbody>
</table>

If the minimum radius is chosen to be zero (\( \text{xmin} = 0. \)), the left-hand boundary type should be reflecting.

### 8.10.4.4 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.

**Cell Volume in Polar Coordinates**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-d</td>
<td>( \pi(r^2_r - r^2_l) )</td>
</tr>
<tr>
<td>2-d</td>
<td>( \frac{1}{2}(r^2_r - r^2_l)\Delta\phi )</td>
</tr>
<tr>
<td>3-d</td>
<td>( \frac{1}{2}(r^2_r - r^2_l)\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 (\( \text{xmin} = 0. \)), the left-hand boundary type should be reflecting.
8.10.5 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.

8.10.5.1 Prolongation

When using a supported non-Cartesian geometry, FLASH has to use geometrically correct prolongation routines. These are located in:

- `source/Grid/GridMain/paramesh/Paramesh2/monotonic` (for PARAMESH 2)
- `source/Grid/GridMain/paramesh/interpolation/Paramesh4/prolong` (for PARAMESH 4)

These paths will be be automatically added by the setup script when the `-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.

**FLASH3 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.

8.10.5.2 Restriction

The default restriction routines understand the supported geometries by default. A cell-volume weighted average is used when restricting the child data up to the parent. For example, in 2-d, the restriction would look like

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

(8.47)

where \( V_{i,j} \) is the volume of the cell with indices, \( i,j \), and the \( ic, jc \) indices refer to the children.

8.11 Unit Test

The Grid unit test has implementations to test both the 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.
the configuration of processors, it is possible to determine the part of the sinusoid on each processor. Since
guard cells 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. As a new option, FLASH3 also offers 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 25) and sfocu (Chapter 24) do not support the direct IO format.

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

FLASH3 supports multiple IO implementations: direct, serial and parallel implementations as well as support for different parallel libraries. In addition, FLASH3 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 FLASH3 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>
</tbody>
</table>

FLASH3 also comes with some predefined setup shortcuts which make choosing the correct IO significantly easier; see Chapter 5 for more details about shortcuts. In FLASH3 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 +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
9.2 OUTPUT FILES

-unit=IO/IOMain/pnetcdf/PM or -unit=IO/IOMain/pnetcdf/UG

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

+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.

Note:

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

In keeping with the FLASH3 code architecture, the F90 module IO/data stores all the data with IO unit scope. The routine IO/init is called once by 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 21 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 genereal IO parameters.

9.2.1 Checkpoint files - Restarting a Simulation

Checkpoint files are used to restart a simulation. In a typical production run, a simulation can be interrupted for a number of reasons—e.g., if the machine crashes, the present queue window closes, the machine runs out of disk space, or perhaps (gasp) there is a bug in FLASH. Once the problem is fixed, a simulation can be restarted from the last checkpoint file rather than the beginning of the run. A checkpoint file contains all the information needed to restart the simulation. The data is stored at full precision of the code (8-byte reals) and includes all of the variables, species, grid reconstruction data, scalar values, as well as meta-data about the run.

The API routine for writing a checkpoint file is IO/writeCheckpoint. Users usually will not need to call this routine directly because the FLASH IO unit calls IO/writeCheckpoint from the routine IO/output which checks the runtime parameters to see if it is appropriate to write a checkpoint file at this time. There are a number of ways to get FLASH to produce a checkpoint file for restarting. Within the flash.par, runtime parameters can be set to dump output. A checkpoint file can be dumped based on elapsed simulation time, elapsed wall clock time or the number of timesteps advanced. A checkpoint file is also produced when the simulation ends, when the max simulation time tmax, the minimum cosmological redshift, or the total number of steps nend 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 rolling_checkpoint, then the checkpoint number is reset to 0. There will be at most rolling_checkpoint checkpoint files kept. This parameter is intended to be used when disk space is at a premium.</td>
</tr>
<tr>
<td>wall_clock_checkpoint</td>
<td>REAL</td>
<td>43200</td>
<td>The maximum amount of wall clock time (seconds) to elapse between checkpoints. When the simulation is started, the current time is stored. If wall_clock_checkpoint seconds elapse over the course of the simulation, a checkpoint file is stored. This is useful for ensuring that a checkpoint file is produced before a queue closes.</td>
</tr>
<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 \texttt{plotfileNumber}
and \texttt{particleFileNumber} to the number of the next plotfile and particle file you want FLASH to output.
In FLASH3 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 \texttt{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 \texttt{plotfileMetadataDP}
and/or \texttt{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 \texttt{plot} \texttt{var} \texttt{1}, \texttt{plot} \texttt{var} \texttt{2}, etc., by setting them in the \texttt{flash.par}
file. For example:

\begin{verbatim}
plot_var_1 = "dens"
plot_var_2 = "pres"
\end{verbatim}

Currently, we support a number of plotvars named \texttt{plot_var}\texttt{n} up to the number of \texttt{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.

#### FLASH3 Transition

In FLASH2 a few variables like density and pressure were output to the plotfiles by default. Because FLASH3 supports a wider range of simulations, it makes no assumptions that
density or pressure variables are even included in the simulation. In FLASH3 a user must
define plotfile variables in the \texttt{flash.par} file, otherwise the plotfiles will not contain any
variables.

The interface for writing a plotfile is the routine \texttt{IO\_writePlotfile}. As with checkpoint files, the user
will not need to call this routine directly because it is invoked indirectly through calling \texttt{IO\_output} when,
based on runtime parameters, FLASH3 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 \texttt{.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>
</tbody>
</table>
### Table 9.3: Plotfile IO parameters (continued).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotFileIntervalTime</td>
<td>REAL</td>
<td>1.0</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, ..., 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 unk, are output in single precision or double precision in plotfiles.</td>
</tr>
</tbody>
</table>

### 9.2.3 Particle files

When Lagrangian particles are included in a simulation, the ParticleIO subunit controls input and output of the particle information. The particle files are stored in double precision. Particle data is written to the checkpoint file in order to restart the simulation, but is not written to plotfiles. Hence analysis and metadata about particles is also written to the particle files. The particle files are intended for more frequent dumps. The interface for writing the particle file is `IO.writeParticles`. Again the user will not usually call this function directly because the routine `IO.output` controls particle output based on the runtime parameters controlling particle files. They are controlled in much of the same way as the plotfiles or checkpoint files and can be dumped based on elapsed simulation time, on steps since the last particle dump or by forcing a particle file to be written by hand by creating a `.dump.particle_file` in the output directory. The following runtime parameters pertain to controlling particle files:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
</table>

### Table 9.4: Particle File IO runtime parameters.
9.2. OUTPUT FILES

Table 9.4: Particle File IO parameters (continued).

<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>

FLASH3 Transition

In FLASH3 each particle dump is written to a separate file. In FLASH2 the particles data structure was broken up into real and integer parts, whereas in FLASH3 all particle properties are real values. See Section 9.9 and Chapter 18 for more information about the particles data structure in FLASH3. Additionally, filtered particles are not implemented in FLASH3.

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

```
./setup IsentropicVortex -2d -auto -unit=Particles +ug
```

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

9.2.4 Integrated Grid Quantities – flash.dat

At each simulation time step, values which represent the overall state (e.g., total energy and momentum) are computed by calculating over all cells in the computations domain. These integral quantities are written to the ASCI file flash.dat. A default routine IO_writeIntegralQuantities is provided to output standard measures for hydrodynamic simulations. The user should copy and modify the routine 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>
</tbody>
</table>
Table 9.5: flash.dat parameters (continued).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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

FLASH3 Transition

Although consistency of runtime parameters was attempted between FLASH2 and FLASH3, a number of them caused confusion in FLASH2. These names were clarified in FLASH3.

Table 9.7: Parameter name changes

<table>
<thead>
<tr>
<th>FLASH2</th>
<th>FLASH3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpnumber</td>
<td>checkpointFileNumber</td>
</tr>
<tr>
<td>trstrt</td>
<td>checkpointFileIntervalTime</td>
</tr>
<tr>
<td>nrstrt</td>
<td>checkpointFileIntervalStep</td>
</tr>
<tr>
<td>ptnumber</td>
<td>plotFileNumber</td>
</tr>
<tr>
<td>tplot</td>
<td>plotfileFileIntervalTime</td>
</tr>
<tr>
<td>pptnumber</td>
<td>particleFileNumber</td>
</tr>
<tr>
<td>ptplot</td>
<td>particleFileIntervalTime</td>
</tr>
<tr>
<td>nppart</td>
<td>particleFileIntervalStep</td>
</tr>
</tbody>
</table>

9.3 Restarts and Runtime Parameters

FLASH3 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 FLASH3 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 FLASH3, each unit has the opportunity to request scalar data to be output to checkpoint or plotfiles. Because there is no central database, each unit “owns” different data in the simulation. For example, the `Driver` unit owns the timestep variable `dt`, the simulation variable `simTime`, and the simulation step number `nStep`. The `Grid` unit owns the sizes of each block, `nxb`, `nyb`, and `nzb`. The `IO` unit owns the variable `checkpointFileNumber`. Each of these quantities are output into checkpoint files. Instead of hard coding the values into checkpoint routines, FLASH3 offers a more flexible interface whereby each unit sends its data to the `IO` unit. The `IO` unit then stores these values in a linked list and writes them to the checkpoint file or plotfile. Each unit has a routine called “`Unit_sendOutputData`”, e.g., `Driver_sendOutputData` and `Grid_sendOutputData`. These routines in turn call `IO_setScalar`. For example, the routine `Grid_sendOutputData` calls

```
IO_setScalar("nxb", NXB)
IO_setScalar("nyb", NYB)
IO_setScalar("nzb", NZB)
```

To output additional simulation scalars in a checkpoint file, the user should override one of the “`Unit_sendOutputData`” or `Simulation_sendOutputData`.

After restarting a simulation from a checkpoint file, a unit might call `IO_getScalar` to reset a variable value. For example, the `Driver` unit calls `IO_getScalar("dt", dr_dt)` to get the value of the timestep `dt` reinitialized from the checkpoint file. A value from the checkpoint file can be obtained by calling
CHAPTER 9. IO UNIT

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 FLASH3, 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 FLASH3 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 plot_grid_var_1, plot_grid_var_2, etc., by setting them in the flash.par file. For example to output the magnitude of vorticity with a declaration in a Config file of GRIDVAR mvrt:

plot_grid_var_1 = "mvrt"

Note that the post-processing routines like fidlr do not display these variables, although they are present in the output file. Future implementations may support this visualization.

9.7 Face-Centered Data

Face-centered variables are now output to checkpoint files, when they are declared in a configuration file. Presently, up to nine face-centered variables are supported in checkpoint files. Plotfile output of face-centered data is not yet supported.

9.8 Output Filenames

FLASH constructs the output filenames based on the user-supplied basename, (runtime parameter 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:

basename_s0000.\{hdf5\ ncmpi\}.chk_0000 ,

where hdf5 or ncmpi (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 ”ncmpi” derived from the serial NetCDF calls beginning with ”nc”)

The general plotfile filename is:

basename_s0000.\{hdf5\ ncmpi\}.plt.\{crn\ cnt\}.0000 ,
9.9. OUTPUT FORMATS

where hdf5 or ncmpl is picked depending on the IO implementation used, crn and cnt indicate data stored at the cell corners or centers respectively, the number following “s” is the split file number, if used, and the number at the end of the filename is the current value of plotfileNumber. crn is reserved, even though corner data output is not presently supported by FLASH3’s IO.

FLASH3 Transition

In FLASH2 the correct format of the names of the checkpoint, plotfile and particle file were necessary in order to read the files with the FLASH fidlr visualization tool. In FLASH3 the name of the file is irrelevant to fidlr3.0 (see Chapter 25). 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 FLASH3 we also offer a serial direct FORTRAN IO which is currently only implemented for the uniform grid. This option is intended to provide users a way to output data if they do not have access to HDF5 or PnetCDF. Additionally, if HDF5 or PnetCDF are not performing well on a given platform the direct IO implementation can be used as a last resort. Our tools, fidlr and sfocu (Part VIII), do not currently support the direct IO implementation, and the output files from this mode are not portable across platforms.

9.9.1 HDF5

HDF5 is supported on a large variety of platforms and offers large file support and parallel IO via MPI-IO. Information about the different versions of HDF can be found at http://hdf.ncsa.uiuc.edu. The IO in FLASH3 implementations require HDF5 1.4.0 or later. Please note that HDF5 1.6.2 requires IDL 1.6 or higher in order to use fidlr3.0 for post processing.

Implementations of the HDF5 IO unit use the HDF application programming interface (API) for organizing data in a database fashion. In addition to the raw data, information about the data type and byte ordering (little- or big-endian), rank, and dimensions of the dataset is stored. This makes the HDF format extremely portable across platforms. Different packages can query the file for its contents without knowing the details of the routine that generated the data.

FLASH provides different HDF5 IO unit implementations – the serial and parallel versions for each supported grid, Uniform Grid and PARAMESH. It is important to remember to match the IO implementation with the correct grid, although the setup script generally takes care of this matching. PARAMESH 2, PARAMESH 4.0, and PARAMESH 4dev all work with the PARAMESH (PM) implementation of IO. Nonfixed blocksize IO has its own implementation in parallel, and is presently not supported in serial mode. Examples are given below for the five different HDF5 IO implementations.

```
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/serial/PM (included by default)
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/serial/UG
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/UG
./setup Sod -2d -auto -nofbs -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/NoFbs
```

The default IO implementation is IO/IOMain/hdf5/serial/PM. It can be included simply by adding -unit=IO to the setup line. In FLASH3, 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 FLASH3 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 FLASH3.

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. FLASH3.2 can make use of this mode by setting the runtime parameter useCollectiveHDF5 to true.

FLASH3 Transition

We recommend that HDF5 version 1.6 or later be used with the HDF5 IO implementations with FLASH3. While it is possible to use any version of HDF5 1.4.0 or later, files produced with versions predating version 1.6 will not be compatible with code using the libraries post HDF5 1.6.

Caution

If you are using version HDF5 >= 1.8 then you must explicitly use HDF5 1.6 API bindings. Either build HDF5 library with “–with-default-api-version=v16” configure option or compile FLASH with the C preprocessor definition H5_USE_16_API. Our preference is to set CFLAGS_HDF5 Makefile.h variable, e.g., for a GNU compilation:

CFLAGS_HDF5 = -I\${HDF5_PATH}/include -DH5_USE_16_API

9.9.1.2 Machine Compatibility

The HDF5 modules have been tested successfully on the ASC platforms and on a Linux clusters. Performance varies widely across the platforms, but the parallel version is usually faster than the serial version. Experience on performing parallel IO on a Linux Cluster using PVFS is reported in Ross et al. (2001). Note that for clusters without a parallel filesystem, you should not use the parallel HDF5 IO module with an NFS mounted filesystem. In this case, all of the information will still have to pass through the node from which the disk is hanging, resulting in contention. It is recommended that a serial version of the HDF5 unit be used instead.

9.9.1.3 HDF5 Data Format

The HDF5 data format for FLASH3 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 \texttt{mdim}, or the maximum dimensions supported by FLASH’s grids, which is three, rather than \texttt{ndim}. \texttt{PARAMESH} 4.0 and \texttt{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 \texttt{time}, \texttt{dt}, \texttt{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 \texttt{IO::setScalar}. See Section 9.4 for more details. The FLASH3 HDF5 format is summarized in Table 9.8.

Table 9.8: FLASH3 HDF5 file format.

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simulation Meta Data: included in all files</strong></td>
<td></td>
</tr>
<tr>
<td>sim info</td>
<td>Stores simulation meta data in a user defined C structure. Structure datatype and attributes of the structure are described below.</td>
</tr>
</tbody>
</table>

```c
typedef struct sim_info_t {
  int file_format_version;
  char setup_call[400];
  char file_creation_time[MAX_STRING_LENGTH];
  char flash_version[MAX_STRING_LENGTH];
  char build_date[MAX_STRING_LENGTH];
  char build_dir[MAX_STRING_LENGTH];
  char build_machine[MAX_STRING_LENGTH];
  char cflags[400];
  char fflags[400];
  char setup_time_stamp[MAX_STRING_LENGTH];
  char build_time_stamp[MAX_STRING_LENGTH];
} sim_info_t;

sim_info_t sim_info;
```

| sim_info.file_format_version: | An integer giving the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file. |
| sim_info.setup_call:         | The complete syntax of the \texttt{setup} command used when creating the current FLASH executable. |
| sim_info.file_creation_time: | The time and date that the file was created. |
| sim_info.flash_version:      | The version of FLASH used for the current simulation. This is returned by routine \texttt{setup::flashVersion}. |
| sim_info.build_date:         | The date and time that the FLASH executable was compiled. |
| sim_info.build_dir:          | The complete path to the FLASH root directory of the source tree used when compiling the FLASH executable. This is generated by the subroutine \texttt{setup::buildstats} which is created at compile time by the Makefile. |
Table 9.8: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_info.build_machine:</td>
<td>The name of the machine (and anything else returned from <code>uname -a</code>) 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 <code>setup\_buildstats</code> is written by the <code>setup</code> script at compile time and also includes the <code>fflags</code> 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 <code>setup\_buildstamp</code> is created by the <code>setup</code> script at compile time.</td>
</tr>
<tr>
<td>sim_info.build_time_stamp:</td>
<td>The date and time the given simulation was built. The routine <code>setup\_buildstamp</code> is created by the <code>setup</code> script at compile time.</td>
</tr>
</tbody>
</table>

_RuntimeParameter and Scalar data_

Data are stored in linked lists with the nodes of each entry for each type listed below.

```c
typedef struct int_list_t {
    char name[MAX_STRING_LENGTH];
    int value;
} int_list_t;

typedef struct real_list_t {
    char name[MAX_STRING_LENGTH];
    double value;
} real_list_t;

typedef struct str_list_t {
    char name[MAX_STRING_LENGTH];
    char value[MAX_STRING_LENGTH];
} str_list_t;

typedef struct log_list_t {
    char name[MAX_STRING_LENGTH];
    int value;
} log_list_t;

int_list_t *int_list;
real_list_t *real_list;
str_list_t *str_list;
log_list_t *log_list;

integer runtime parameters

int_list_t int_list(numIntParams)

A linked list holding the names and values of all the integer runtime parameters.
Table 9.8: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>real runtime parameters</td>
<td><code>real_list_t real_list(numRealParams)</code>&lt;br&gt;A linked list holding the names and values of all the real runtime parameters.</td>
</tr>
<tr>
<td>string runtime parameters</td>
<td><code>str_list_t str_list(numStrParams)</code>&lt;br&gt;A linked list holding the names and values of all the string runtime parameters.</td>
</tr>
<tr>
<td>logical runtime parameters</td>
<td><code>log_list_t log_list(numLogParams)</code>&lt;br&gt;A linked list holding the names and values of all the logical runtime parameters.</td>
</tr>
<tr>
<td>integer scalars</td>
<td><code>int_list_t int_list(numIntScalars)</code>&lt;br&gt;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>&lt;br&gt;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(numStr Scalars)</code>&lt;br&gt;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>&lt;br&gt;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)`<br>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)`<br>This array stores the refinement level for each block. |
| node type                     | `integer nodetype(globalNumBlocks)`<br>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)`
<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinates</td>
<td>This array stores the coordinates of the center of the block.</td>
</tr>
<tr>
<td>block size</td>
<td>This array stores the dimensions of the current block.</td>
</tr>
<tr>
<td>bounding box</td>
<td>This array stores the minimum and maximum coordinate of a block in each spatial direction.</td>
</tr>
<tr>
<td>which child</td>
<td>An integer array identifying which part of the parents’ volume this child corresponds to.</td>
</tr>
<tr>
<td>variable</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>localnp</td>
<td>This array holds the number of particles on each processor.</td>
</tr>
<tr>
<td>particle names</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>
</tbody>
</table>

*Particle Data: included in checkpoint files and particle files*
### 9.9. OUTPUT FORMATS

#### Table 9.8: HDF5 format (continued).

<table>
<thead>
<tr>
<th>Record label</th>
<th>Description of the record</th>
</tr>
</thead>
<tbody>
<tr>
<td>tracer particles</td>
<td>real particles(NPART_PROPS, globalNumParticles) 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.

```
./setup Sod -2d -auto -unit=IO/IOMain/pnetcdf/PM
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/pnetcdf/UG
```

The paths to these IO implementations can be long and tedious to type, users are advised to set up shortcuts for various implementations. See Chapter 5 for information about creating shortcuts.

To the end-user, the PnetCDF data format is very similar to the HDF5 format. (Under the hood the data storage is quite different.) In HDF5 there are datasets and dataspaces, in PnetCDF there are dimensions and variables. All the same data is stored in the PnetCDF checkpoint as in the HDF5 checkpoint file, although there are some differences in how the data is stored. The grid data is stored in multidimensional arrays, as it is in HDF5. These are unknown names, refine level, node type, gid, coordinates, proc number, block size and bounding box. The particles data structure is also stored in the same way. The simulation metadata, like file format version, file creation time, `setup` command line, etc., are stored as global attributes. The runtime parameters and the output scalars are also stored as attributes. The `unk` and particle labels are also stored as global attributes. In PnetCDF, all global quantities must be consistent across all processors involved in a write to a file, or else the write will fail. All IO calls are run in a collective mode in PnetCDF.

#### 9.9.3 Direct IO

As mentioned above, the direct IO implementation has been added so users can always output data even if the HDF5 or pnetCDF libraries are unavailable. The user should examine the two helper routines `io_writeData`
and `io_readData`. Copy the base implementation to a simulation directory, and modify them in order to write out specifically what is needed. To include the direct IO implementation add the following to your setup line:

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

### 9.9.4 Output Side Effects

In FLASH3 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 25). In addition, ViSit version 10.0 and higher (see Section 23.7) can natively read FLASH3 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.

#### FLASH3 Transition

The `IO` unit test replaces the simulation setup `io_benchmark` in FLASH2.

The `unitTest` is setup like any other FLASH3 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:

mpirun -np numProcs flash3

There are a few things to keep in mind when working with the IO unit test:

- The Config file in unitTest/IO declares some dummy grid scope variables which are stored in the unk array. If the user wants a more intensive IO test, more variables can be added. Variables are initialized to dummy values in Driver_evolveFlash.

- Variables will only be output to the plotfile if they are declared in the flash.par (see the example flash.par in the unit test).

- The only units besides the simulation unit included in this simulation are Grid, IO, Driver, Timers, Logfile, RuntimeParameters and PhysicalConstants.

- If the PARAMESH Grid implementation is being used, it is important to note that the grid will not refine on its own. The user should set lrefine_min to a value ≥ 1 to create more blocks. The user could also set the runtime parameters nblockx, nblocky, nblockz to make a bigger problem.

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

9.11.1 Online tips for working with the IO Unit

Please see the website for further hints on efficient usage of the IO unit.
Chapter 10

Runtime Parameters Unit

![Diagram showing the directory tree for RuntimeParameters 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**.

```plaintext
#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.

```plaintext
#snippet from a flash.par
myParameter = 45.0
```

See **Section 5.5** for more information on declaring parameters in a **Config** file.

### 10.2 Identifying Valid Runtime Parameters

The values of runtime parameters are initialized either from default values defined in the **Config** files, or from values explicitly set in the file *flash.par*. Variables that have been changed from default are noted in the
CHAPTER 10.  RUNTIME PARAMETERS UNIT

=======================================================
RuntimeParameters:
=======================================================
pt_numx  =  10 [CHANGED]
pt_numy  =  5 [CHANGED]
checkpointfileintervalstep = 0

Figure 10.2: Section of output log showing runtime parameters values

physics/Eos/EosMain/Multigamma
  gamma [REAL] [1.6667]
    Valid Values: Unconstrained
    Ratio of specific heats for gas

physics/Hydro/HydroMain
  cfl [REAL] [0.8]
    Valid Values: Unconstrained
    Courant factor

Figure 10.3: Portion of a setup_params file from an object directory.

simulation’s output log. For example, the RuntimeParameters section of the output log shown in Figure 10.2 indicates that pt_numx and pt_numy have been read in from flash.par and are different than the default values, whereas the runtime parameter checkpointFileIntervalStep has been left at the default value of 0.

After a simulation has been configured with a setup call, all possible valid runtime parameters are listed in the file setup_params located in the object directory (or whatever directory was chosen with -objdir=) with their default values. This file groups the runtime parameters according to the units with which they are associated and in alphabetical order. A short description of the runtime parameter, and the valid range or values if known, are also given. See Figure 10.3 for an example listing.

10.3 Routine Descriptions

The Runtime Parameters unit is included by default in all of the provided FLASH simulation examples, through a dependence within the Driver unit. The main FLASH initialization routine (Driver_initFlash) and the initialization code created by setup handles the creation and initialization of the runtime parameters, so users will mainly be interested in querying parameter values. Because the RuntimeParameters routines are overloaded functions which can handle character, real, integer, or logical arguments, the user must make sure to use the interface file RuntimeParameters_Interfaces in the calling routines.

The user will typically only have to use one routine from the Runtime Parameters API, RuntimeParameters_get. This routine retrieves the value of a parameter stored in the linked list in the RuntimeParameters_data module. In FLASH3 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.

### FLASH3 Transition

In FLASH2, the user called the routine get_parm_from_context from within if (.firstcall.) blocks from anywhere in the code. In FLASH3, code within the if (.firstcall.) block is typically moved to the Unit_init.F90 file. Put all calls to RuntimeParameter_get within this initialization routine. Since the _init routine for each unit is only called once, FLASH3 eliminates the need to have messy if (.firstcall.) brackets throughout the code.

### FLASH3 Transition

FLASH3 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.

```fortran
subroutine IO_init(myPE, numProcs)
    use IO_data
    use RuntimeParameters_interface, ONLY : RuntimeParameters_get
    implicit none

    integer, intent(in) :: myPE, numProcs

    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_`.

**FLASH3 Transition**

FLASH3 Tip: A Note About Restarting In the case of FLASH3 restarts, runtime parameter values are also read in from the checkpoint file and are stored as "previous" values in the runtime parameter lists. For more information, consult Section 9.3.
Chapter 11

Multispecies Unit

Figure 11.1: The Multispecies unit directory tree.

FLASH has the ability to track multiple fluids, each of which can have its own properties. The Multispecies unit handles setting, 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.

# Portion of a \code{Config} file for a Simulation
SPECIES AIR
SPECIES SF6

Figure 11.2: Config file showing how to define required fluid species.
Figure 11.3: Header file Flash.h showing integer definition of fluid species.

<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.

FLASH3 Transition

In FLASH2, you found the integer index of a species by using find_fluid_index. In FLASH3, 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 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.

FLASH3 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).

  ```fortran
  subroutine Simulation_initSpecies()
  implicit none
  ! 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
  ```

- **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 \( A_i, Z_i, \ldots, \gamma_i \)) is denoted \( P_i \), the sum calculated is

\[
\sum_{i=1}^{N} \frac{w_i}{P_i}.
\]

For example, the average atomic mass of a collection of fluids is typically defined by

\[
\bar{A} = \frac{\sum X_i A_i}{\sum X_i},
\]

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

```fortran
call Multispecies_getSumInv(A, abarinv, xn(:))
abar = 1.e0 / abarinv
```

where \( xn(:) \) is an array of the mass fractions of each species in FLASH. This method allows some of the mass fractions to be zero.

• **Multispecies-getSumFrac** Same as **Multispecies-getSum**, but compute the weighted sum of the chosen property divided by the total number of particles (\( A_i \)). If the weights give the mass fractions \( X_i \) of the fluids at a point in space and the selected property (one of \( A_i, Z_i, \ldots, \gamma_i \)) is denoted \( P_i \), the sum calculated is

\[
\sum_{i} \frac{X_i A_i P_i}{\sum X_i}.
\]

• **Multispecies-getSumSqr** Same as **Multispecies-getSum**, but compute the weighted sum of the squares of the chosen property values. If the weights are denoted \( w_i \) and the selected property (one of \( A_i, Z_i, \ldots, \gamma_i \)) is denoted \( P_i \), the sum calculated is

\[
\sum_{i} w_i P_i^2.
\]

• **Multispecies-list** List the contents of the multifluid database in a snappy table format.

### 11.4 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.

```
#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.

FLASH3 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 \( \text{Mpc}^3 \text{Gyr}^{-2} \text{Msun}^{-1} \) by calling

\[
\text{call PhysicalConstants_get ("Newton", G, len_unit="Mpc", time_unit="Gyr", mass_unit="Msun")}
\]

In this example, the local variable \( G \) is set equal to the result, \( 4.4983 \times 10^{-15} \) (to five significant figures).


149
### 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.

#### FLASH3 Transition

The header file `PhysicalConstants.h` must be included in the calling routine due to the optional arguments of `PhysicalConstants_get`.
### 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>1.660540210E-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>$\frac{2}{3H_0}$</td>
</tr>
<tr>
<td>mass</td>
<td>MFLY</td>
<td>9.8847E45</td>
<td>5.23e12 Msun</td>
</tr>
</tbody>
</table>

### 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

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

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{13.1}
\]

\[
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P = \rho \mathbf{g} \tag{13.2}
\]

\[
\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} \tag{13.3}
\]
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,
\[
E = \epsilon + \frac{1}{2}|\mathbf{v}|^2 ,
\]
(13.4)
\( g \) 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
\[
P = (\gamma - 1)\rho \epsilon ,
\]
(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
\[
\epsilon = E - \frac{1}{2}|\mathbf{v}|^2
\]
(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
\[
\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + P) \mathbf{v}] - \mathbf{v} \cdot \nabla P = 0 .
\]
(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
\[
\frac{\partial \rho X_\ell}{\partial t} + \nabla \cdot (\rho X_\ell \mathbf{v}) = 0 ,
\]
(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 \texttt{hydro} unit has a capability to advect mass scalars. Mass scalars are field variables advected with density, similar to species mass fractions,
\[
\frac{\partial \rho \phi_\ell}{\partial t} + \nabla \cdot (\rho \phi_\ell \mathbf{v}) = 0 ,
\]
(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.

\textbf{FLASH3 Transition}

In FLASH2, one specified just the number of mass scalars, not their identity, using the \texttt{NUMMASSSCALARS} keyword in the setup \texttt{Config}. FLASH3 uses the \texttt{MASS_SCALAR} keyword so that each mass scalar can be identified by name. The setup script determines the number of mass scalars by parsing through the \texttt{Config} files.
### 13.1 Gas hydrodynamics

#### 13.1.1 Usage

Two gas hydrodynamic solvers supplied in the release of FLASH3.2 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 \texttt{physics/Hydro/HydroMain/split/PPM} and \texttt{physics/Hydro/HydroMain/unsplit/Hydro_MusclHancock}.

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_MusclHancock.

#### 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 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.

---

**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; \text{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 \text{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>
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_steepeening = .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 FLASH3.2 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) methods. This unsplit hydro solver is a reduced version of the Unsplit Staggered Mesh (USM) MHD solver (see details in Section 13.3.1) that has been available in previous FLASH3 releases.
13.1. GAS HYDRODYNAMICS

Table 13.3: Additional runtime parameters for *Interpolation Schemes* in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro_Unsplit)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>integer</td>
<td>2</td>
<td>Order of method in data reconstruction: 1st order Godunov (FOG), 2nd order MUSCL-Hancock (MH), 3rd order PPM, 5th order WENO. NOTE: WENO requires NGUARD=6, and should include +supportWENO in setup line</td>
</tr>
<tr>
<td>transOrder</td>
<td>integer</td>
<td>3</td>
<td>Interpolation order of taking upwind biased transverse flux derivatives in the unsplit data reconstruction: 1st, 2nd, 3rd.</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>fullyLimit</td>
<td>logical</td>
<td>.false.</td>
<td>On/off full limiting on transverse fluxes (i.e., Donor cell method) instead of using CTU method. When fullyLimited on, CFL limit will be automatically reduced.</td>
</tr>
<tr>
<td>use_steepening</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable contact discontinuity steepening for PPM and WENO</td>
</tr>
<tr>
<td>use_flattening</td>
<td>logical</td>
<td>.false.</td>
<td>Enable/disable flattening (or reducing) numerical oscillations for MH, PPM, and WENO</td>
</tr>
<tr>
<td>use_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>
</tbody>
</table>

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).
Table 13.4: Additional runtime parameters for Riemann Solvers in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro_Unsplit)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>shockInstabilityFix</td>
<td>logical</td>
<td>.false.</td>
<td>On/off carbuncle, odd-even instability fix for the Roe solver</td>
</tr>
<tr>
<td>shockDetect</td>
<td>logical</td>
<td>.false.</td>
<td>On/off detecting strong shocks/rarefactions to gain numerical stability</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>

Some notes on runtime parameters

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.

1. 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 guardcells will require increased number of inter-processor data exchanges, resulting in slower performance during parallel computations.

2. 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.

3. 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.

4. EOSforRiemann = .true. will call (expensive) EOS routines to compute consistent adiabatic indices (e.g., gamc, 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 gamc, 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 gamc, 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.

5. 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.
13.2. RELATIVISTIC HYDRODYNAMICS (RHD)

### 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., $\text{CFL} < 1$). However, in some circumstances where there are strong shocks and rarefactions, `shockDetect=.true.` could be useful to gain more numerical stability by (1) using the more stable Donor cell method, and (2) 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.

---

13.2 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.

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

\[ h = 1 + \frac{\Gamma}{\Gamma - 1} \rho \]

(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, \(U = (D, m, E)\), or primitive, \(V = (\rho, v, p)\), variables. The connection between the two sets is given by

\[ D = \gamma \rho, \quad m = \rho \gamma^2 v, \quad E = \rho h \gamma^2 - p, \]

(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 \(V\) in terms of \(U\), is

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

(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 \(p\):

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

(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 \(U\). 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 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:

\texttt{REQUIRES physics/Eos/EosMain/Gamma/RHD}

For this specific purpose, the current 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 RHD unit is
13.3 Magnetohydrodynamics (MHD)

13.3.1 Description

The FLASH3 code includes two magnetohydrodynamic (MHD) units that represent two different algorithms. The first is the eight-wave model (8Wave) by Powell et al. (1999) that is already present in FLASH2. The second is a newly implemented unsplit staggered mesh algorithm (USM or StaggeredMesh). It should be noted that there are several major differences between the two MHD units. The first difference is how each algorithm enforces the solenoidal constraint of magnetic fields. The eight-wave model basically uses the truncation-error method, which effectively removes the effects of unphysical magnetic monopoles if they are generated during simulations. It does not, however, completely eliminate monopoles that are spurious in a strict physical law. To improve such unphysical effects in simulations, the 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

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0
\]

\[
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v} - \mathbf{B} \mathbf{B}) + \nabla p_* = \rho \mathbf{g} + \nabla \cdot \tau
\]

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

\[
\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{vB} - \mathbf{Bv}) = -\nabla \times (\eta \nabla \times \mathbf{B})
\]

where

\[
p_* = p + \frac{B^2}{2},
\]

\[
E = \frac{1}{2} v^2 + \epsilon + \frac{1}{2} \frac{B^2}{\rho}.
\]

\[
\tau = \mu \left( (\nabla \mathbf{v}) + (\nabla \mathbf{v})^T - \frac{2}{3} (\nabla \cdot \mathbf{v}) \mathbf{I} \right)
\]
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 13.6 and Table 13.7.

13.3.2 Usage

In the current release of FLASH3, 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 \(+\text{usm}\) in a setup script. The default eight-wave unit will be automatically chosen if there is no such specification included.

Note also that a full 3D implementation of the unsplit staggered mesh solver is a "research mode" in this release and the 3D code is available to the users on a collaboration basis. The users can see the Flash website from the Code Support Web Page to get the 3D unsplit staggered mesh code. The public release includes the 1D and 2D implementations of the unsplit staggered mesh solver.

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 \(-\text{gridinterpolation=native}\) is automatically adopted if \(+\text{8wave}\) is specified in setup, otherwise, \(-\text{gridinterpolation=native}\) should be explicitly included in order to use the eight-wave solver without specifying \(+\text{8wave}\). 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 \texttt{FIXEDBLOCKSIZE} and \texttt{NONFIXEDBLOCKSIZE} modes, and the adaptive grid with \texttt{PARAMESH} on Cartesian geometries. When using AMR grids, the eight-wave unit supports both \texttt{PARAMESH 2} and \texttt{PARAMESH 4}, while only \texttt{PARAMESH 4} is supported in the unsplit staggered mesh solver because face-centered variables are only fully supported in \texttt{PARAMESH 4}.

13.3.3 Algorithm: The Eight-wave Solver

The eight-wave magnetohydrodynamic (MHD) unit in the FLASH3 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
Table 13.6: 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 13.7: 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 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>

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 FLASH3 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 $\nabla \cdot \mathbf{B} = 0$ 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 $\nabla \cdot \mathbf{B}$ 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 FLASH3 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 $\nabla \cdot \mathbf{B}$. 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 $\nabla \nabla \cdot \mathbf{B}$ is added to the induction equation, so that the equation becomes

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{vB} - \mathbf{Bv}) = -\mathbf{v} \nabla \cdot \mathbf{B} + \eta_a \nabla \nabla \cdot \mathbf{B}, \quad (13.22)$$

with the artificial diffusion coefficient $\eta_a$ chosen to match that of grid numerical diffusion. In the FLASH code, $\eta_a = \frac{1}{2}\left(\frac{1}{\Delta x} + \frac{1}{\Delta y} + \frac{1}{\Delta z}\right)^{-1}$, where $\lambda$ is the largest characteristic speed in the flow. Since the grid magnetic diffusion Reynolds number is always on the order of unity, this operator locally destroys magnetic monopoles at the rate at which they are created. Recent numerical experiments (Powell et al., 2001) indicate that this approach can very effectively bring the strength of spurious magnetic monopoles to levels that are sufficiently low, so that generated solutions remain physically consistent. The entire $\nabla \cdot \mathbf{B}$ 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 FLASH3 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 FLASH3. 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 $\rho\mathbf{v}$ 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):

- **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 $\nabla \cdot \mathbf{B}$ in the induction equations.
• **An intermediate Riemann problem:** An intermediate set of high-order Godunov fluxes is calculated using the cell interface values obtained from the data reconstruction-evolution step. The resulting fluxes are then used to evolve the normal fields by a half time step in the next procedure.

• **A half time step update for the normal fields:** The normal magnetic fields are evolved by a half time step using the flux-CT method at cell interfaces, ensuring the divergence-free property on a staggered grid. This intermediate update for the normal fields and the half time step data from the data reconstruction-evolution step together provide a second-order accurate MHD states at cell interfaces.

• **Riemann problem:** Using the second-order MHD states calculated from the above procedures, the scheme proceeds to solve the Riemann problem to obtain high-order Godunov fluxes at cell interfaces.

• **Unsplit update of cell-centered variables:** The unsplit time integrations are performed using the high-order Godunov fluxes to update the cell-centered variables for the next time step.

• **Construction of electric fields:** Using the high-order Godunov fluxes, the cell-cornered (edged in 3D) electric fields are constructed. The unsplit staggered mesh scheme computes a new modified electric field construction (MEC) scheme that includes first and second multidimensional derivative terms in Taylor expansions for high-order interpolations. This modified electric field construction provides enhanced accuracy by explicitly adding proper amounts of dissipation as well as spatial gradients in its interpolation scheme.

• **Flux-CT scheme:** The electric fields from the MEC scheme are used to evolve the cell face-centered magnetic fields by solving a set of discrete induction equations. The resulting magnetic fields satisfy the divergence-free constraint up to the accuracy of machine round-off errors.

• **Reconstruct cell-centered magnetic fields:** The cell-centered magnetic fields are reconstructed from the divergence free cell face-centered magnetic fields by taking arithmetic averages of the cell face-centered fields variables.

Note that the procedure required in solving one-dimensional MHD equations is much simpler than solving the multidimensional ones and only involves the first through third and the fifth steps in the above outlined scheme. The choices of TVD slope limiters available in the unsplit staggered mesh scheme (see Table 13.8) includes the minmod limiter as well as the compressible limiters such as van Leer 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):

$$
\begin{align*}
    b_{x,i+1/2,j}^{n+1} &= b_{x,i+1/2,j}^n - \frac{\Delta t}{\Delta y} \left\{ E_{z,i+1/2,j}^{n+1/2} - E_{z,i+1/2,j-1/2}^{n+1/2} \right\}, \\
    b_{y,i,j+1/2}^{n+1} &= b_{y,i,j+1/2}^n - \frac{\Delta t}{\Delta x} \left\{ -E_{z,i+1/2,j}^{n+1/2} + E_{z,i-1/2,j+1/2}^{n+1/2} \right\}.
\end{align*}
$$

(13.23)  (13.24)

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 \( b_{x,i+1/2,j} \) and \( b_{y,j+1/2} \), which are updated via the above induction equations, satisfy the divergence-free constraint locally. The numerical divergence of the magnetic fields is defined as

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

(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 13.8 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:

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

\[
= \frac{1}{4} \left\{ E_{z,i+1/2,j}^{n+1/2} + E_{z,i+1/2,j+1}^{n+1/2} + E_{z,i,j+1/2}^{n+1/2} + E_{z,i,j+1}^{n+1/2} \right\},
\]

(13.26)

where \( F_{B_y} \) and \( G_{B_z} \) 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 \textit{E\_modification} in the unsplit staggered mesh scheme, which takes Taylor series expansions of the cell-cornered electric field \( E_{z,i+1/2,j+1}^{n+1/2} \) 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 \( B_{y,j,i} \) 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 \textit{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. 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 \(\int \partial F_n E \cdot T dl\) along the four edges of the face \(F_{x,i+1/2,j,k}\) gives rise to the negative of the rate of change of the magnetic field flux in \(x\)-direction through the area enclosed by the four edges (e.g., the area of \(F_{x,i+1/2,j,k}\)).

Table 13.8: 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 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>proMethod</td>
<td>string</td>
<td>&quot;INJECTION_PROL&quot;</td>
<td>Use either direct injection method (&quot;INJECTION_PROL&quot;) or Balsara’s method (&quot;BALSARA_PROL&quot;) in prolonging divergence-free magnetic fields stored in face-centered variables</td>
</tr>
<tr>
<td>RiemannSolver</td>
<td>string</td>
<td>&quot;ROE&quot;</td>
<td>&quot;HLLD&quot; is additionally available for MHD, &quot;Hybrid&quot; is NOT available for MHD.</td>
</tr>
</tbody>
</table>
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 (1) using the more stable Donor cell method, and (2) lowering CFL accordingly. 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.

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/magetoHD/.

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 FLASH3.2 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 altogether. 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 FLASH3: 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).

**FLASH3 Transition**

FLASH2 had three functions in the Eos unit interface; eos, eos1d and eos3d. The eos function operated on a single point in space, the eos1d function operated on a vector and eos3d could handle an entire block. Of these, only the single point function, eos, was aware of the optional thermodynamic derivative quantities. FLASH3 has simplified the interface into a primary Eos function that operates on a vector, and a wrapper function, Eos_wrapped, that can convert a block or its section into the format understood by the Eos function and vice-versa. Only the direct Eos function is aware of the optional derived quantities, although the calculation of them is still an optional argument. Because of the optional arguments, both routines should use the interface block file “Eos_interface.F90”, in the calling routines.

FLASH2 passed in the mode of operation with the parameter input; FLASH3 uses the parameter mode. The equivalences are:

<table>
<thead>
<tr>
<th>FLASH2</th>
<th>FLASH3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MODE_DENS_TEMP</td>
</tr>
<tr>
<td>2</td>
<td>MODE_DENS_EI</td>
</tr>
<tr>
<td>3</td>
<td>MODE_DENS_PRESS</td>
</tr>
</tbody>
</table>

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

$$\gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial \rho}.$$ \hspace{1cm} (14.1)

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

$$\gamma_4 = 1 + \frac{P}{\rho e}.$$ \hspace{1cm} (14.2)
These two adiabatic indices are stored as the mesh-based variables `GAMC_VAR` and `GAME_VAR`. All EOS routines must return $\gamma_1$, and $\gamma_4$ is calculated from (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

$$P = (\gamma - 1) \rho \epsilon . \quad (14.3)$$

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

$$P = \frac{N_a k}{\bar{A}} \rho T , \quad (14.4)$$

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

$$\frac{1}{\bar{A}} = \sum_i X_i \frac{A_i}{A_i} , \quad (14.5)$$

where $X_i$ is the mass fraction of the $i$th element. Equating these expressions for pressure yields an expression for the specific internal energy as a function of temperature

$$\epsilon = \frac{1}{\gamma - 1} \frac{N_a k}{\bar{A}} T . \quad (14.6)$$

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

$$\frac{1}{(\gamma - 1)} = \bar{A} \sum_i \frac{1}{(\gamma_i - 1)} X_i . \quad (14.7)$$

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

$$P_{\text{tot}} = P_{\text{rad}} + P_{\text{ion}} + P_{\text{ele}} + P_{\text{pos}} + P_{\text{coul}} \quad (14.8)$$

$$\epsilon_{\text{tot}} = \epsilon_{\text{rad}} + \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{pos}} + \epsilon_{\text{coul}} . \quad (14.9)$$

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

\[
P_{\text{rad}} = \frac{a T^4}{3} \quad (14.10)
\]

\[
\epsilon_{\text{rad}} = \frac{3 P_{\text{rad}}}{\rho} \quad (14.11)
\]

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

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

\[
N_{\text{pos}} = \frac{8\pi \sqrt{2}}{h^3} \, m_e^3 \, c^3 \, \beta^{3/2} \left[ F_{1/2}(-\eta - 2/\beta, \beta) + \beta \, F_{3/2}(-\eta - 2/\beta, \beta) \right] , \quad (14.13)
\]

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

\[
F_k(\eta, \beta) = \int_0^\infty \frac{x^k \left(1 + 0.5 \beta x\right)^{1/2}}{\exp(x - \eta) + 1} \, dx . \quad (14.14)
\]

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

\[
N_{\text{ele, matter}} = \frac{\bar{Z}}{\bar{A}} \rho = \bar{Z} \, N_{\text{ion}} , \quad (14.15)
\]

and charge neutrality requires

\[
N_{\text{ele, matter}} = N_{\text{ele}} - N_{\text{pos}} . \quad (14.16)
\]

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,

\[
F = \epsilon - T \, S \quad (14.17)
\]

\[
dF = -S \, dT + \frac{P}{\rho^2} \, d\rho , \quad (14.18)
\]

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 \(A = Z = 1\) (pure hydrogen), with an electron fraction \(Y_e = 1\). One reason for not including contributions from photons and ions in the table is that these components of the Helmholtz EOS are very simple (Equations 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 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 below the formal accuracy of the hydrodynamics algorithm (Fryxell, et al. 2000; Timmes & Swesty 2000). The physical limits of the Helmholtz EOS are \(10^{-10} < \rho < 10^{11} \text{ (g cm}^{-3}\text{)}\) and \(10^4 < T < 10^{11} \text{ (K)}\). As with the gamma-law EOS, the Helmholtz EOS provides both forward and backward relations. In the case of the forward relation \((\rho, T, \text{ given along with the composition})\) the table lookup scheme and analytic formulae directly provide relevant thermodynamic quantities. In the case of the backward relation \((\rho, \epsilon, \text{ and composition given})\), 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.

14.4 Usage

14.4.1 Initialization

The initialization function of the Eos unit \texttt{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 \texttt{Eos_init} routine is a little more complex. The Helmholtz EOS requires an input file \texttt{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 \texttt{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 \texttt{Eos_init} routine reads in the table data on processor 0 and broadcasts it to all other processors.

14.4.2 Runtime Parameters

Runtime parameters for the Gamma unit require the user to set the thermodynamic properties for the single gas. \texttt{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 \texttt{Config} file indicates the requested species, for example helium and oxygen can be defined as

\begin{verbatim}
SPECIES HE4
SPECIES 016
\end{verbatim}

The properties of the gases are initialized in the file \texttt{Simulation_initSpecies.F90}, for example

\begin{verbatim}
subroutine Simulation_initSpecies()
    use Multispecies_interface, ONLY : Multispecies_setProperty
    implicit none
    #include "Flash.h"
    #include "Multispecies.h"
\end{verbatim}
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_t 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 $\rho$-$T$ 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.

### 14.4.3 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 $\gamma_1$, 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.

### 14.5 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 FLASH3. 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 \texttt{odeStepper} controls which integration method is used in the simulation. The choice \texttt{odeStepper = 1} is the default and invokes a Bader-Deuflhard scheme. The choice \texttt{odeStepper = 2} invokes a Kaps-Rentrop or Rosenbrock scheme. The runtime parameter \texttt{algebra} controls which linear algebra package is used in the simulation. The choice \texttt{algebra = 1} is the default and invokes the sparse matrix MA28 package. The choice \texttt{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

$$X_i = \frac{n_i}{\rho} = \frac{n_i A_i}{(\rho N_A)} ,$$

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

$$Y_i = \frac{X_i}{A_i} = \frac{n_i}{(\rho N_A)} .$$

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 $i(j,k)l$,

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

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 $V_j$ 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 $V_i \equiv 0$ is important for two reasons. First, mass diffusion is often unimportant when compared to other transport processes, such as thermal or viscous diffusion (i.e., large Lewis numbers and/or small Prandtl numbers). Such a situation obtains, for example, in the study of laminar flame fronts propagating through the quiescent interior of a white dwarf. Second, this case permits the decoupling of the reaction network solver from the hydrodynamical solver through the use of operator splitting, greatly simplifying the algorithm. This is the method used by the default FLASH distribution. Setting $V_i \equiv 0$ transforms (15.4) into

$$\frac{dY_i}{dt} = \dot{R}_i, \quad (15.6)$$

which may be written in the more compact, standard form

$$\dot{y} = f(y). \quad (15.7)$$

Stated another way, in the absence of mass diffusion or advection, any changes to the fluid composition are due to local processes.

Because of the highly nonlinear temperature dependence of the nuclear reaction rates and because the abundances themselves often range over several orders of magnitude in value, the values of the coefficients which appear in (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 $\mathbf{J} \equiv \partial \mathbf{f} / \partial \mathbf{y}$ 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

$$\dot{Y}(^{4}\text{He}) = -Y(^{4}\text{He}) Y(^{12}\text{C}) R + \ldots,$$
$$\dot{Y}(^{12}\text{C}) = -Y(^{4}\text{He}) Y(^{12}\text{C}) R + \ldots,$$
$$\dot{Y}(^{16}\text{O}) = +Y(^{4}\text{He}) Y(^{12}\text{C}) R + \ldots, \quad (15.8)$$

where the ellipses indicate additional terms coming from other reaction sequences. The minus signs indicate that helium and carbon are being destroyed, while the plus sign indicates that oxygen is being created. Each of these three expressions contributes two terms to the Jacobian matrix $\mathbf{J} = \partial \mathbf{f} / \partial \mathbf{y}$

$$J(^{4}\text{He},^{4}\text{He}) = -Y(^{12}\text{C}) R + \ldots \quad J(^{4}\text{He},^{12}\text{C}) = -Y(^{4}\text{He}) R + \ldots,$$
$$J(^{12}\text{C},^{4}\text{He}) = -Y(^{12}\text{C}) R + \ldots \quad J(^{12}\text{C},^{12}\text{C}) = -Y(^{4}\text{He}) R + \ldots,$$
$$J(^{16}\text{O},^{4}\text{He}) = +Y(^{12}\text{C}) R + \ldots \quad J(^{16}\text{O},^{12}\text{C}) = +Y(^{4}\text{He}) R + \ldots. \quad (15.9)$$

Entries in the Jacobian matrix represent the flow, in number of nuclei per second, into (positive) or out of (negative) an isotope. All of the temperature and density dependence is included in the reaction rate $R$. 
The Jacobian matrices that arise from nuclear reaction networks are neither positive-definite nor symmetric, since the forward and reverse reaction rates are generally not equal. In addition, the magnitudes of the matrix entries change as the abundances, temperature, or density change with time.

This release of FLASH3 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 $^{54}$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.

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 \(~5.0\times10^7\) 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 reaction network is advanced over a timestep \(h\) according to

\[
y^{n+1} = y^n + \sum_{i=1}^{4} b_i \Delta_i ,
\]

where the four vectors \(\Delta^i\) are found from successively solving the four matrix equations

\[
(\hat{1}/\gamma h - \hat{J}) \cdot \Delta_1 = f(y^n) \tag{15.11}
\]
\[
(\hat{1}/\gamma h - \hat{J}) \cdot \Delta_2 = f(y^n + a_{21} \Delta_1) + c_{21} \Delta_1 / h \tag{15.12}
\]
\[
(\hat{1}/\gamma h - \hat{J}) \cdot \Delta_3 = f(y^n + a_{31} \Delta_1 + a_{32} \Delta_2) + (c_{31} \Delta_1 + c_{32} \Delta_2) / h \tag{15.13}
\]
\[
(\hat{1}/h - \hat{J}) \cdot \Delta_4 = f(y^n + a_{41} \Delta_1 + a_{42} \Delta_2 + a_{43} \Delta_3) + (c_{41} \Delta_1 + c_{42} \Delta_2 + c_{43} \Delta_3) / h . \tag{15.14}
\]

\(b_i, \gamma, a_{ij},\) and \(c_{ij}\) are fixed constants of the method. An estimate of the accuracy of the integration step is made by comparing a third-order solution with a fourth-order solution, which is a significant improvement over the basic Euler method. The minimum cost of this method — which applies for a single timestep that meets or exceeds a specified integration accuracy — is one Jacobian evaluation, three evaluations of the right-hand side, one matrix decomposition, and four backsubstitutions. Note that the four matrix equations represent a staged set of linear equations (\(\Delta_4\) depends on \(\Delta_3\ldots\) depends on \(\Delta_1\)). Not all of the right-hand sides are known in advance. This general feature of higher-order integration methods impacts the optimal choice of a linear algebra package. The fourth-order Kaps-Rentrop routine in FLASH makes use of the routine GRK4T given by Kaps & Rentrop (1979).

Another time integration method used by FLASH for evolving the reaction networks is the variable order Bader-Deuflhard method (\textit{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,

\[
h = H/m
\]
\[
(\hat{1} - \hat{J}) \cdot \Delta_0 = hf(y^n)
\]
\[
y_1 = y^n + \Delta_0 .
\]

Then from \(k = 1, 2, \ldots, m - 1\)

\[
(\hat{1} - \hat{J}) \cdot x = hf(y_k) - \Delta_{k-1}
\]
\[
\Delta_k = \Delta_{k-1} + 2x
\]
\[
y_{k+1} = y_k + \Delta_k , \tag{15.16}
\]

and closure is obtained by the last stage

\[
(\hat{1} - \hat{J}) \cdot \Delta_m = h[f(y_m) - \Delta_{m-1}]
\]
\[
y^{n+1} = y_m + \Delta_m . \tag{15.17}
\]
This staged sequence of matrix equations is executed at least twice with \( m = 2 \) and \( m = 6 \), yielding a fifth-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 \texttt{odeStepper} controls which integration method is used in the simulation. The choice \texttt{odeStepper = 1} is the default and invokes the variable order Bader-Deuflhard scheme. The choice \texttt{odeStepper = 2} invokes the fourth order Kaps-Rentrop / Rosenbrock scheme.

### 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 \texttt{useShockBurn} parameter is set to \texttt{.false.}, this algorithm is used to detect shocks in the Burn unit and to switch off the burning in shocked cells. Currently, the shock detection algorithm supports Cartesian and 2-dimensional cylindrical coordinates. The basic algorithm is to compare the jump in pressure in the direction of compression (determined by looking at the velocity field) with a shock parameter (typically \( 1/3 \)). If the total velocity divergence is negative and the relative pressure jump across the compression front is larger than the shock parameter, then a cell is considered to be within a shock.

This computation is done on a block by block basis. It is important that the velocity and pressure variables have up-to-date guard cells, so a guard cell call is done for the burners only if we are detecting shocks (i.e. \texttt{useShockBurn = .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.
15.2. IONIZATION UNIT

The tabulation of Caughlan & Fowler (1988) is used in FLASH for most of the key nuclear reaction rates. Modern values for some of the reaction rates were taken from the reaction rate library of Hoffman (2001, priv. comm.). A user can choose between two reaction rate evaluations in FLASH. The runtime parameter \texttt{useBurnTable} controls which reaction rate evaluation method is used in the simulation. The choice \texttt{useBurnTable = 0} is the default and evaluates the reaction rates from analytical expressions. The choice \texttt{useBurnTable = 1} evaluates the reactions rates from table interpolation. The reaction rate tables are formed on-the-fly from the analytical expressions. Tests on one-dimensional detonations and hydrostatic burnings suggest that there are no major differences in the abundance levels if tables are used instead of the analytic expressions; we find less than 1% differences at the end of long timescale runs. Table interpolation is about 10 times faster than evaluating the analytic expressions, but the speedup to FLASH is more modest, a few percent at best, since reaction rate evaluation never dominates in a real production run.

Finally, nuclear reaction rate screening effects as formulated by Wallace et al. (1982) and decreases in the energy generation rate $\dot{\epsilon}_{\text{nuc}}$ due to neutrino losses as given by Itoh et al. (1996) are included in FLASH.

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 \texttt{enucDtFactor} controls this ratio. The timestep limiter is calculated as

$$\Delta t_{\text{burn}} = \texttt{enucDtFactor} \cdot \frac{E_{\text{int}}}{E_{\text{nuc}}}$$

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, \texttt{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 \texttt{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

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{15.21}
\]

\[
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla \mathbf{P} = \rho \mathbf{g} \tag{15.22}
\]

\[
\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho E + P \mathbf{v}) = \rho \mathbf{v} \cdot \mathbf{g} \tag{15.23}
\]

\[
\frac{\partial n_i^Z}{\partial t} + \nabla \cdot n_i^Z \mathbf{v} = R_i^Z \quad (i = 1, \ldots, N_{\text{spec}}), \tag{15.24}
\]

where \(\rho\) is the fluid density, \(t\) is the time, \(\mathbf{v}\) is the fluid velocity, \(\mathbf{P}\) is the pressure, \(E\) is the sum of the internal energy and kinetic energy per unit mass, \(\mathbf{g}\) is the acceleration due to gravity, \(n_i^Z\) is the number density of the ion \(i\) of the element \(Z\), \(N_{\text{spec}}\) is the total number of species, and

\[
R_i^Z = N_e [n_{i+1}^Z \alpha_{i+1}^Z + n_{i-1}^Z S_{i-1}^Z - n_i^Z (\alpha_i^Z + S_i^Z)], \tag{15.25}
\]

where \(N_e\) is the electron number density, \(\alpha_i^Z \equiv \alpha(N_e,T)\) are the collisional and dielectronic recombination coefficients, and \(S_i^Z \equiv S(N_e,T)\) are the collisional ionization coefficients of Summers(1974).

### 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

\[
\frac{\partial n_i^Z}{\partial t} = R_i^Z \quad (i = 1, \ldots, N_{\text{spec}}) \tag{15.26}
\]

are integrated. This step incorporates the reactive source terms. Within each grid cell, the above equations can be solved separately with a standard ODE method. Since this system is “stiff”, it is solved using the Bader-Deuflhard time integration solver with the MA28 sparse matrix package. Timmes (1999) has shown that these two algorithms together provide the best balance of accuracy and overall efficiency.

Note that in the present version, the contribution of the ionization and recombination to the energy equation (the bracketed term in (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_den1e8.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 23.6.1)), and Eqi (untested in FLASH3). 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.
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>

15.3 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 $st\text{-}\text{energy}$) divided by the decay time $\tau$ ($st\text{-}\text{decay}$). In the limit that the timestep $\Delta t \to 0$, it is easily seen that the algorithm represents a linearly-weighted summation of the old state with the new Gaussian random variable.

By evolving the phases of the stirring modes in Fourier space, imposing a divergence-free condition is relatively straightforward. At each timestep, the solenoidal component of the velocities is projected out, leaving only the non-compressional modes to add to the velocities.

The velocities are then converted to physical space by a direct Fourier transform – i.e., adding the sin and cos terms explicitly. Since most drivings involve a fairly small number of modes, this is more efficient than an FFT, since the FFT would need large numbers of modes (equal to six times the number of cells in the domain), the vast majority of which would have zero amplitude. The runtime parameters associated with the Stir unit are described in the online [rpi reference] page.
Chapter 16

Diffusive Terms

Figure 16.1: The organizational structure of the Diffuse unit.

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

FLASH3 Transition

Up to FLASH3.2, Diffuse was classified under sourceTerms. The internal organization of the unit has also changed. The standalone implementation of the DiffuseMain subunit is new to FLASH3.3. As of FLASH3.3, the flux-based public routines that used to be part of DiffuseMain have been moved into the new DiffuseFluxBased subunit. If you upgraded from FLASH3.2, you may have to update directory names in your code that were pointing explicitly to the old locations.

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 FLASH3.1 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 FLASH3.1, 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, &
        primaryRightCoord, tempFlx, tempAreaLeft)

    call Diffuse_visc (sweepDir, igeom, blockList(blk), numCells,&
        blkLimits, blkLimitsGC, primaryLeftCoord,primaryRightCoord,&
        tempFlx, tempAreaLeft,secondCoord,thirdCoord)

    ! call Diffuse_species(sweepDir, igeom, blockList(blk), numCells,&
    !    blkLimits, blkLimitsGC, primaryLeftCoord,primaryRightCoord,&
    !    tempFlx, tempFly, tempFlz)
```

To use the Diffuse unit for heat conduction in a FLASH3 simulation, the runtime parameters useDiffuse and useDiffuseTherm must be set to .true. ; and to use the Diffuse unit for viscosity effects in a FLASH3 simulation, the runtime parameters useDiffuse and useDiffuseVisc must be set to .true..
16.1.2 Diffuse Split 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.

However, as of FLASH3.3, only an implementation for thermal conductivity on uniform grids is provided. This subunit has the implicit split implementation for thermal conduction. Second order Crank Nicholson scheme is used for discretization. The diffusion coefficient is time lagged and hence the scheme is fully implicit for constant diffusivity (only). It is directionally split (default: Strang) to integrate with the pencil grid architecture of the GridSolvers/Pfft implementation of the Grid unit.

Salient features,

- It is 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.
- Supports uniform grid (only).

To use the Split implementation of Diffuse unit, a simulation should

- Include the Diffuse unit using an option like \texttt{-with-unit=source/physics/Diffuse/DiffuseMain/Split} on the setup line, or \texttt{REQUIRES physics/Diffuse/DiffuseMain/Split} in a Config file.

- set \texttt{useDiffuse} run time parameter.

The boundary conditions can be set using

- \texttt{diff.XlBoundaryType}, \texttt{diff.XrBoundaryType},
- \texttt{diff.YlBoundaryType}, \texttt{diff.YrBoundaryType},
- \texttt{diff.ZlBoundaryType}, \texttt{diff.ZrBoundaryType}.

Supported boundary conditions:

- Outflow (homogeneous Neumann).
- Dirichlet (homogeneous).
- Periodic.

The scheme is implicit and hence can use larger dt. Run time parameter \texttt{dt.diff_factor} can be modified to increase dt.

Run time parameters \texttt{diff.scaleFactThermSaTime} and \texttt{diff.scaleFactThermFlux} can be used to couple the implicit and explicit diffusion.

<table>
<thead>
<tr>
<th>\texttt{diff.scaleFactThermSaTime}</th>
<th>\texttt{diff.scaleFactThermFlux}</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>Implicit</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

Note: The results of combining the Implicit, Explicit scheme need not necessarily produce equivalent or better results.

The ConductionDelta simulation problem makes use of the diffuse unit. Using default setup \texttt{./setup ConductionDelta -1d -auto} activates the Flux based diffusion. The default behavior is set in Config file of ConductionDelta simulation.

In order to use the Split implementation,

- Override the default by using \texttt{-without-unit=source/physics/Diffuse/DiffuseFluxBased}

- Include split implementation using \texttt{-unit=source/physics/Diffuse/DiffuseMain/Split}

So, the setup call would look like \texttt{./setup ConductionDelta -1d -auto -unit=source/physics/Diffuse/DiffuseMain/Split -without-unit=source/physics/Diffuse/DiffuseFluxBased}
Chapter 17

Gravity Unit

17.1 Introduction

The Gravity unit supplied with FLASH3 computes gravitational source terms for the code. These source terms can take the form of the gravitational potential $\phi(x)$ or the gravitational acceleration $g(x)$,

$$g(x) = -\nabla \phi(x) .$$

(17.1)

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

$$\nabla^2 \phi(x) = 4\pi G \rho(x) ,$$

(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

$$
\begin{align*}
g_{x:ijk} &= \frac{1}{2\Delta x} (\phi_{i-1,j,k} - \phi_{i+1,j,k}) + \mathcal{O}(\Delta x^2) \\
g_{y:ijk} &= \frac{1}{2\Delta y} (\phi_{i,j-1,k} - \phi_{i,j+1,k}) + \mathcal{O}(\Delta y^2) \\
g_{z:ijk} &= \frac{1}{2\Delta z} (\phi_{i,j,k-1} - \phi_{i,j,k+1}) + \mathcal{O}(\Delta z^2) 
\end{align*}
$$

(17.3)

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 8.9.2.1 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 8.9.2.2 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

$$\rho E = \frac{1}{2} \rho v^2 + \rho \epsilon ,$$  \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 $n + 1$

$$\left(\rho v\right)_{i}^{n+1} = \left(\rho v\right)_{i}^{n} + \frac{\Delta t}{2} \left( g_{i}^{n+1} \left( \rho_{i}^{n} + \rho_{i}^{n+1} \right) \right)$$

$$\left(\rho E\right)_{i}^{n+1} = \left(\rho E\right)_{i}^{n} + \frac{\Delta t}{4} \left( g_{i}^{n+1} \left( \rho_{i}^{n} + \rho_{i}^{n+1} \right) \left( v_{i}^{n} + v_{i}^{n+1} \right) \right) .$$ \hspace{1cm} (17.5)

Here $g_{i}^{n+1}$ is obtained by extrapolation from $\phi_{i}^{n-1}$ and $\phi_{i}^{n}$. The Poisson gravity implementation supplies a mesh variable to contain the potential from the previous timestep; future releases of FLASH may permit the storage of several time levels of this quantity for hydrodynamics algorithms that require more steps. Currently, $g$ 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

$$v_{L,i+1/2} \rightarrow v_{L,i+1/2} + \frac{\Delta t}{4} \left( g_{L,i+1/2}^{+} + g_{L,i+1/2}^{-} \right)$$

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

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

```
-with-unit=physics/Gravity
```
on your command line when you configure the code with \texttt{setup}. The default implementation is \texttt{Constant}, which can be overridden by including the entire path to the specific implementation in the command line or \texttt{Config} file. The other available implementations are \texttt{Gravity/GravityMain/Plane}par, \texttt{Gravity/-GravityMain/Pointmass} and \texttt{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 \texttt{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 \texttt{GPOT}\_\texttt{VAR} and \texttt{GPOT}\_\texttt{VAR}, and others as needed, will be automatically defined in \texttt{Flash.h} in those cases. See \texttt{Gravity\_potentialListOfBlocks} for more information.

### 17.5 Unit Tests

There are two unit tests for the gravity unit. \texttt{Poisson3} is essentially the Maclaurin spheroid problem described in Section 23.3.4. Because an analytical solution exists, the accuracy of the gravitational solver can be quantified. The second test, \texttt{Poisson3\_active} is a modification of \texttt{Poisson3} to test the mapping of particles in \texttt{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 \texttt{PoisParticles} discussed in Section 23.4.3. \texttt{PoisParticles} is based on the Huang-Greengard Poisson gravity test described in Section 23.3.3.
Chapter 18

Particles Unit

Figure 18.1: The \texttt{Particles} unit main subunit.

Figure 18.2: The \texttt{Particles} unit with \texttt{ParticlesInitialization} and \texttt{ParticlesMapping} subunits.
CHAPTER 18. PARTICLES UNIT

The support for particles in FLASH3 comes in two flavors, *active* and *passive*. Active particles have mass, and contribute to the dynamics of the simulation, while massless passive particles follow the motion of Lagrangian tracers and make no contribution to the dynamics. Particles are dimensionless objects characterized by positions \( \mathbf{x}_i \), velocities \( \mathbf{v}_i \), 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.

\[
\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i
\]

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

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

may come into play. Here \( \mathbf{a}_i \) 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.

\[
m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{F}_{\text{lr},i} + \mathbf{F}_{\text{sr},i}
\]

Here, \( \mathbf{F}_{\text{lr},i} \) represents the sum of all long-range forces (coupling all particles) acting on the \( i \)th particle and \( \mathbf{F}_{\text{sr},i} \) 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).

---

**FLASH3 Transition**

Please note that the particles routines have not been thoroughly tested with non-Cartesian coordinates; use them at your own risk!

---

**New in FLASH3.1**

In release 3.1 of FLASH, a single simulation can have both active and passive particles defined. FLASH3 and FLASH2 allowed only active or passive particles in a simulation. Because of the added complexity, new `Config` syntax and new `setup` script syntax is necessary for Particles. See Section 5.2 for command line options, Section 6.6 for `Config` syntax, and Section 18.3 below for more details.
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 FLASH3.1 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

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 is 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:
  \[
  x_i^{n+1} = x_i^n + v_i^n \Delta t^n \\
  v_i^{n+1} = v_i^n + a_i^n \Delta t^n.
  \] (18.4)
  Here \( a_i \) is the particle acceleration. Within FLASH3, 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
  \[
  x_i^1 = x_i^0 + v_i^0 \Delta t^0 \\
  v_i^{1/2} = v_i^0 + \frac{1}{2} a_i^0 \Delta t^0 \\
  v_i^{n+1/2} = v_i^{n-1/2} + C_n a_i^n + D_n a_i^{n-1} \\
  x_i^{n+1} = x_i^n + v_i^{n+1/2} \Delta t^n.
  \] (18.5)
  The coefficients \( C_n \) and \( D_n \) are given by
  \[
  C_n = \frac{1}{2} \Delta t^n + \frac{1}{3} \Delta t^{n-1} + \frac{1}{6} \left( \frac{\Delta t^{n-2}}{\Delta t^{n-1}} \right) \\
  D_n = \frac{1}{6} \left( \Delta t^{n-1} - \frac{\Delta t^{n-2}}{\Delta t^{n-1}} \right). \] (18.6)
  By using time-centered velocities and stored accelerations, this method achieves second-order time accuracy even with variable timesteps. Within FLASH3, 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

$$
x_i^1 = x_i^0 + v_i^0 \Delta t^0
$$

$$
v_i^{1/2} = v_i^0 + \frac{1}{2} a_i^0 \Delta t^0
$$

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

$$
x_i^{n+1} = x_i^n + v_i^{n+1/2} \Delta t^n .
$$

Here we define $A \equiv -2 \ddot{a}/a$, 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 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS_PART_PROP</td>
<td>Particle mass</td>
</tr>
<tr>
<td>ACCX_PART_PROP</td>
<td>$x$-component of particle acceleration</td>
</tr>
<tr>
<td>ACCY_PART_PROP</td>
<td>$y$-component of particle acceleration</td>
</tr>
<tr>
<td>ACCZ_PART_PROP</td>
<td>$z$-component of particle acceleration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OACX_PART_PROP</td>
<td>$x$-component of particle acceleration at previous timestep</td>
</tr>
<tr>
<td>OACY_PART_PROP</td>
<td>$y$-component of particle acceleration at previous timestep</td>
</tr>
<tr>
<td>OACZ_PART_PROP</td>
<td>$z$-component of particle acceleration at previous timestep</td>
</tr>
</tbody>
</table>

18.1.2 Passive Particles

Passive particles may be moved using one of several different methods available in FLASH3. With the exception of Midpoint, they are all single-step schemes. The methods are either first-order or second-order
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 cannot be arbitrarily chosen by a particle motion ODE solver. Statements about the order of a method in this context should be understood as referring to the same method if it were applied in a hypothetical simulation where evaluations of velocities $v_i$ could be performed at arbitrary times (and with unlimited accuracy). Note that FLASH3 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 $\text{Particles\_advance}$ is called, the fluid fields have already been updated to $t^{n+1}$, as is the case with the $\text{Driver\_evolveFlash}$ implementations provided with FLASH3. A specific implementation of the passive portion of $\text{Particles\_advance}$ is selected by a setup option such as $-\text{with-unit=Particles/ParticlesMain/passive/Euler}$, or by specifying something like $\text{REQUIRES Particles/ParticlesMain/passive/Euler}$ in a simulation’s $\text{Config}$ file (or by listing the path in the $\text{Units}$ file if not using the $-\text{auto}$ configuration option). Further details are given in Section 18.3 below.

- **Forward Euler ($\text{Particles/ParticlesMain/passive/Euler}$).** Particles are advanced in time from $t^n$ to $t^{n+1} = t^n + \Delta t^n$ using the following difference equation:

$$x_i^{n+1} = x_i^n + v_i^n \Delta t^n .$$

(18.7)

Here $v_i^n$ 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 $v_i^n$ 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.

- **Two-Stage Runge-Kutta ($\text{Particles/ParticlesMain/passive/RungeKutta}$).** This 2-stage Runge-Kutta scheme is the preferred choice in FLASH3. 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:

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

(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; 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 ($\text{Particles/ParticlesMain/passive/Midpoint}$).** 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

$$x_i^{n+1} = x_i^{n-1} + v_i^n (\Delta t^{n-1} + \Delta t^n) .$$

(18.9)

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

---

1Even though it is possible to do so, see $\text{Particles\_computeDt}$, one does not in general wish to let particles integration dictate the time step of the simulation.

2They can be considered “predicted” positions and velocities.
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:

- 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).

Particle advancement follows the equation

$$x_i^{n+1} = x_i^n + \Delta t^n v_i^{\text{comb}},$$

where

$$v_i^{\text{comb}} = c_1 v_i(x_i^{*,n+\frac{1}{2}}, t^n) + c_2 v_i(x_i^{*,n+\frac{3}{2}}, t^{n+1}) + c_3 v_i(x_i^n, t^n) + c_4 v_i(x_i^n, t^{n+1})$$

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

$$c_1 = c_1(\Delta t^{n-1}, \Delta t^n),$$
$$c_2 = c_2(\Delta t^{n-1}, \Delta t^n),$$
$$c_3 = c_3(\Delta t^{n-1}, \Delta t^n),$$
$$c_4 = c_4(\Delta t^{n-1}, \Delta t^n)$$

are chosen dependent on the change in time step so that the method stays second order when $\Delta t^{n-1} \neq \Delta t^n$.

Conditions for the correction can be derived as follows: Let $\Delta t^n = \frac{1}{2} \Delta t^{n-1}$ the estimated half time step used in the scheme, let $t_i^{n+\frac{1}{2}} = t^n + \Delta t^n$ the estimated midpoint time, and $t^{n+\frac{3}{2}} = t^n + \frac{3}{2} \Delta t^n$ 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 $t^{n+\frac{3}{2}}$, and we continue to denote with $x_i^{*,n+\frac{1}{2}}$ the estimated positions reached at the estimated midpoint time $t_i^{n+\frac{1}{2}}$.

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

$$v(x_i^{E,n+\frac{1}{2}}, t^{n+\frac{3}{2}}) = v(x_i^n, t^n) + v_t(x_i^n, t^n) \frac{1}{2} \Delta t^n + (v_i^n \cdot \frac{\partial}{\partial x}) v(x_i^n, t^n) \frac{1}{2} \Delta t^n + O((\frac{1}{2} \Delta t^n)^2)$$

by Taylor expansion. It is known that the propagation scheme $\bar{x}_i^{n+1} = x_i^n + v(x_i^{E,n+\frac{1}{2}}, t^{n+\frac{3}{2}}) \Delta t$ using these velocities is second order (this is known as the modified Euler method).

On the other hand, expansion of (18.11) gives

$$v_i^{\text{comb}} = (c_1 + c_2 + c_3 + c_4) v_i(x_i^n, t^n)$$
$$+ (c_2 + c_4) v_t(x_i^n, t^n) \Delta t + (c_1 + c_2) (v_i^n \cdot \frac{\partial}{\partial x}) v(x_i^n, t^n) \Delta t^n$$
$$+ \text{higher order terms in } \Delta t \text{ and } \Delta t^n.$$
After introducing a time step factor \( f \) defined by \( \Delta t^n_+ = f \Delta t^n \), this becomes
\[
\begin{align*}
\mathbf{v}_{i}^{\text{comb}} - (c_1 + c_2 + c_3 + c_4)\mathbf{v}(x^n_i, t^n) + (c_2 + c_4)\mathbf{v}_i(x^n_i, t^n)\Delta t + (c_1 + c_2)(\mathbf{v}^n_i \cdot \frac{\partial}{\partial x})\mathbf{v}(x^n_i, t^n) f \Delta t
+ O((\Delta t)^2) .
\end{align*}
\]
(18.13)

One can derive conditions for second order accuracy by comparing (18.13) with (18.12) and requiring that
\[
\mathbf{v}_{i}^{\text{comb}} = \mathbf{v}(x^{E,n+\frac{1}{2}}_i, t^{n+\frac{1}{2}}) + O((\Delta t)^2) .
\]
(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 \( O(\Delta t^{n-1}) \) and \( O(\Delta t^{n}) \) error terms:
\[
\begin{align*}
c_1 + c_2 + c_3 + c_4 &= 1 \quad \text{(otherwise the scheme will not even be of first order)}, \\
c_2 + c_4 &= \frac{1}{2} \quad \text{(and thus also } c_1 + c_3 = \frac{1}{2} \text{)}, \\
c_1 + c_2 &= \frac{\Delta t^n}{\Delta t^{n-1}} .
\end{align*}
\]

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_advancPassive 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:
\[
\begin{align*}
\Delta t^n > \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq pt\_dtChangeToleranceUp \times \Delta t^{n-1} \\
\Delta t^n < \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq pt\_dtChangeToleranceDown \times \Delta t^{n-1},
\end{align*}
\]
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 \( x^{E,n+\frac{1}{2}}_i \) and \( v^{E,n+\frac{1}{2}}_i \) between the Particles_advence 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 FLASH3, 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

\[ f_i(x) = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} f(x') \, dx' \]  \hspace{1cm} (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 `Particles_mapFromMesh` and `Particles_mapToMeshOneBlk`.

Specifying the desired mapping method is accomplished by designating the `MAPMETHOD` in the Simulation 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

\[ f(x) = A + B(x - x_i) + C(x - x_i)^2 \] \hspace{1cm} (18.16)

Then integrating gives

\[ f_{i-1} = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B(x - x_i)^2 \right]_{x_i-\frac{1}{2}}^{x_{i-1/2}} + \frac{1}{3} C(x - x_i)^3 \right]_{x_i-\frac{3}{2}}^{x_{i-1/2}} \]
\[ = A - B\Delta x + \frac{13}{12} C\Delta x^2, \] \hspace{1cm} (18.17)

\[ f_i = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B(x - x_i)^2 \right]_{x_i-\frac{1}{2}}^{x_{i+1/2}} + \frac{1}{3} C(x - x_i)^3 \right]_{x_{i-1/2}}^{x_i} \]
\[ = A + \frac{1}{12} C\Delta x^2, \] \hspace{1cm} (18.18)

and

\[ f_{i+1} = \frac{1}{\Delta x} \left[ A + \frac{1}{2} B(x - x_i)^2 \right]_{x_{i+1/2}}^{x_{i+3/2}} + \frac{1}{3} C(x - x_i)^3 \right]_{x_{i-3/2}}^{x_{i-1/2}} \]
\[ = A - B\Delta x + \frac{13}{12} C\Delta x^2, \] \hspace{1cm} (18.19)

We may write these as

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

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

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

In this case, the cell averaged quantities are given by
\[
f_{i,j} (x,y) = \frac{1}{\Delta y} \Delta x \int_{x_{i-1/2}}^{x_{i+1/2}} dx' \int_{y_{j-1/2}}^{y_{j+1/2}} dy' f(x', y') .
\]
(18.23)

Integrating the 9 possible cell averages gives, after some algebra,
\[
\begin{bmatrix}
    f_{i-1,j-1} \\
    f_{i,j} \\
    f_{i+1,j-1} \\
    f_{i-1,j} \\
    f_{i,j} \\
    f_{i+1,j} \\
    f_{i-1,j+1} \\
    f_{i,j+1} \\
    f_{i+1,j+1}
\end{bmatrix} =
\begin{bmatrix}
    1 & -1 & \frac{13}{12} & -1 & \frac{13}{12} & 1 \\
    1 & 0 & \frac{13}{12} & -1 & \frac{13}{12} & 0 \\
    1 & 1 & \frac{13}{12} & -1 & \frac{13}{12} & -1 \\
    1 & -1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\
    1 & 0 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\
    1 & 1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\
    1 & -1 & \frac{13}{12} & 1 & \frac{13}{12} & -1 \\
    1 & 0 & \frac{13}{12} & 1 & \frac{13}{12} & 0 \\
    1 & 1 & \frac{13}{12} & 1 & \frac{13}{12} & 1
\end{bmatrix}
\begin{bmatrix}
    A \\
    B \Delta x \\
    C \Delta x^2 \\
    D \Delta y \\
    E \Delta y^2
\end{bmatrix} .
\]
(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
\[
\begin{bmatrix}
    f_{i,j-1} \\
    f_{i,j} \\
    f_{i,j+1}
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 & \frac{13}{12} & -1 & \frac{13}{12} & 1 \\
    1 & -1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\
    1 & 1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\
    1 & -1 & \frac{13}{12} & 1 & \frac{13}{12} & -1 \\
    1 & 0 & \frac{13}{12} & 1 & \frac{13}{12} & 0 \\
    1 & 1 & \frac{13}{12} & 1 & \frac{13}{12} & 1
\end{bmatrix}
\begin{bmatrix}
    A \\
    B \Delta x \\
    C \Delta x^2 \\
    D \Delta y \\
    E \Delta y^2
\end{bmatrix} .
\]
(18.25)

Inverting gives
\[
\begin{bmatrix}
    A \\
    B \Delta x \\
    C \Delta x^2 \\
    D \Delta y \\
    E \Delta y^2
\end{bmatrix} =
\begin{bmatrix}
    \frac{1}{24} & \frac{1}{24} & \frac{7}{6} & \frac{1}{24} & \frac{1}{24} \\
    0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\
    0 & -\frac{1}{2} & -1 & \frac{1}{2} & 0 \\
    -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\
    \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
    f_{i,j-1} \\
    f_{i,j} \\
    f_{i,j+1}
\end{bmatrix} .
\]
(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
\[
\begin{bmatrix}
    A \\
    B \Delta x \\
    C \Delta x^2 \\
    D \Delta y \\
    E \Delta y^2 \\
    F \Delta z \\
    G \Delta z^2
\end{bmatrix} =
\begin{bmatrix}
    \frac{1}{24} & \frac{1}{24} & \frac{5}{12} & \frac{1}{24} & \frac{1}{24} & \frac{1}{24} & \frac{1}{24} \\
    0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\
    0 & -\frac{1}{2} & -1 & \frac{1}{2} & 0 & 0 & 0 \\
    -\frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\
    \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2} & 0 & 0 \\
    -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\
    \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
    f_{i,j,k-1} \\
    f_{i,j,k} \\
    f_{i,j,k+1}
\end{bmatrix} .
\]
(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
\[
\begin{bmatrix}
A \\
B \Delta r \\
C \Delta r^2 \\
D \Delta z \\
E \Delta z^2
\end{bmatrix} = 
\begin{bmatrix}
-\frac{1}{2\Delta r} & -\frac{h_1-1}{2\Delta r} & \frac{h_1}{2\Delta r} & -\frac{h_1-1}{2\Delta r} & -\frac{1}{2\Delta r} \\
0 & -\frac{1}{2\Delta r} & \frac{h_1}{2\Delta r} & 0 & 0 \\
0 & \frac{h_1}{h_2} & 0 & \frac{h_1}{h_2} & 0 \\
-\frac{1}{2} & 0 & -1 & 0 & 0 \\
0 & \frac{1}{7} & -1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
f_{i,j-1} \\
f_{i-1,j} \\
f_{i,j} \\
f_{i+1,j} \\
f_{i,j+1}
\end{bmatrix}.
\] (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 FLASH3 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 PARTICLETYPE 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 FLASH3. 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 8.8.2. 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 FLASH3 distribution (they are more likely to used with the passive tracer particles):
  - Lattice distributes particles regularly along the axes directions throughout a subsection of the physical grid.
  - WithDensity distributes particles randomly, with particle density being proportional to the grid gas density.

- 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 submit, as the algorithms required vary considerably between the grid implementations. The boundary conditions are also implemented by the GridParticles unit. See Section 8.8 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 FLASH3, 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

```
./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 18.1.2. Add other options as desired.

For unitTest/ParticlesAdvance/HomologousPassive,

```
./setup -auto -1d unitTest/ParticlesAdvance/HomologousPassive +ug -nxb=80
```

is recommended to get started.

When varying the test, the following runtime parameters defined for Simulation/SimulationMain/unitTest/ParticlesAdvance will probably need to be adjusted:

PARAMETER sim_schemeOrder INTEGER 2 — The order of the integration scheme. This should probably always be either 1 or 2.

PARAMETER sim_maxTolCoeff0 REAL 1.0e-8 — Zero-th order error coefficient $C_0$, used for convergence criterion if sim_schemeOrder = 0.

PARAMETER sim_maxTolCoeff1 REAL 0.0001 — First order error coefficient $C_1$, used for convergence criterion if sim_schemeOrder = 1.

PARAMETER sim_maxTolCoeff2 REAL 0.01 — Second order error coefficient $C_2$, used for convergence criterion if sim_schemeOrder = 2.

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

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

where maxError is the maximum absolute error between numerical and analytic solution for any particle that was encountered during a simulation run, and maxActualDt is the maximum time step $\Delta t$ used in the run. The appropriate runtime parameters of various units, in particular Driver, Particles, and Grid, should be used to control the desired simulation run. In particular, it is recommended to vary 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 $\mathbf{x} = \mathbf{r}/a$, where $\mathbf{r}$ is a proper position vector and $a(t)$ is the time-dependent cosmological scale factor. The present epoch is defined to correspond to $a = 1$; in the following discussion we use $t = t_0$ to refer to the age of the Universe at the present epoch. The gas velocity $\mathbf{v}$ is taken to be the comoving peculiar velocity $\mathbf{\dot{x}}$. The comoving gas
density, pressure, temperature, and internal energy are defined to be

\[ \rho \equiv a^3 \tilde{\rho} \]
\[ p \equiv a \tilde{p} \]
\[ T \equiv \frac{T}{a^2} \]
\[ \rho \epsilon \equiv a \tilde{\rho} \tilde{\epsilon} . \]

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

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

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

\[ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0 \]
\[ \frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + p) v] + 2 \frac{\dot{a}}{a} \rho v + \rho \nabla \phi = 0 \]
\[ \frac{\partial \rho}{\partial t} + \nabla \cdot [\rho v] - \dot{v} \cdot \nabla p + \frac{\dot{a}}{a} (3\gamma - 1) \rho \epsilon = 0 . \]

Here \( E \) is the specific total energy, \( \epsilon + \frac{1}{2} v^2 \), and \( \gamma \) is the effective ratio of specific heats. The Cosmology unit applies the terms involving \( \dot{a} \) via the Cosmology_redshiftHydro routine.

The comoving potential \( \phi \) in the above equations is the solution to the Poisson equation in the form

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

where \( \tilde{\rho} \) 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

\[ \tilde{\rho} \equiv \Omega_m \rho_{\text{crit}} \]
\[ \rho_{\text{crit}} = \frac{3H^2}{8\pi G} . \]

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

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

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

\[ \Omega_c \equiv \Omega_m + \Omega_r + \Omega_\Lambda - 1 . \]
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 FLASH3 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 **OmegaMatter**, **OmegaRadiation**, and **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>useCosmology</td>
<td>BOOLEAN</td>
<td>.true.</td>
<td>True if cosmology is to be used in this simulation</td>
</tr>
<tr>
<td>OmegaMatter</td>
<td>REAL</td>
<td>0.3</td>
<td>Ratio of total mass density to critical density at the present epoch ($\Omega_m$)</td>
</tr>
<tr>
<td>OmegaBaryon</td>
<td>REAL</td>
<td>0.05</td>
<td>Ratio of baryonic (gas) mass density to critical density at the present epoch; must be $\leq$ OmegaMatter ($\Omega_b$)</td>
</tr>
<tr>
<td>CosmologicalConstant</td>
<td>REAL</td>
<td>0.7</td>
<td>Ratio of the mass density equivalent in the cosmological constant to the critical density at the present epoch ($\Omega_\Lambda$)</td>
</tr>
<tr>
<td>OmegaRadiation</td>
<td>REAL</td>
<td>$5 \times 10^{-5}$</td>
<td>Ratio of the mass density equivalent in radiation to the critical density at the present epoch ($\Omega_r$)</td>
</tr>
<tr>
<td>HubbleConstant</td>
<td>REAL</td>
<td>$2.1065 \times 10^{-18}$</td>
<td>Value of the Hubble constant $H_0$ in sec$^{-1}$</td>
</tr>
<tr>
<td>MaxScaleChange</td>
<td>REAL</td>
<td>HUGE(1.)</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` directory tree.

**FLASH3 Transition**

In this release, FLASH’s implementation of the material properties units is minimal. For Heat Conductivity and Viscosity, we provide implementations for effects with constant coefficients; these can be used as models for implementing effects that follow other laws. For MassDiffusivity, only no-operation stubs are provided. A routine that calculates constant magnetic resistivity and viscosity is provided in the MagneticResistivity unit and can be used in non-ideal magnetohydrodynamics simulations. Several add-on capabilities are being made available to the users from the Code Support Web Page.
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 FLASH3.2 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 FLASH3 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 $\nu_m = \frac{1}{\mu_0} \eta$ 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.

**FLASH3 Transition**

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 `visc_whichCoefficientIsConst`.

To use viscosity in a FLASH3 simulation, the runtime parameter `useViscosity` must be set to `.true.`

20.4 Mass Diffusivity

The MassDiffusivity unit implements a prescription for calculating a generic mass diffusivity that can be used by the Hydro PPM, and MHD solvers. In this release the unit only provides non-operational stub functionalities.
Part VI

Monitor Units
Chapter 21

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 unfushed 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

221
basenm = "cellular_"
log_file = "cellular.log"

21.1 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:

================================================================================
Number of processors: 2
Dimensionality: 2
Max Number of Blocks/Proc: 1000
Number x zones: 8
Number y zones: 8
Number z zones: 1
Build stamp: Wed Apr 19 16:35:57 2006
System info:
Linux zingiber.uchicago.edu 2.6.12-1.1376_FC3smp #1 SMP Fri Aug 26 23:50:33 EDT
Version: FLASH 3.0.
Build directory: /home/kantypas/FLASH3/trunk/Sod
Setup syntax:
/home/kantypas/FLASH3/trunk/bin/setup.py Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
-objdir=Sod
f compiler flags:
/usr/local/pgi6/bin/pgf90 -I/usr/local/mpich-pg/include -c -r8 -i4 -fast -g
-DMAXBLOCKS=1000 -DNXB=8 -DNYB=8 -DNZB=1 -DN_DIM=2
c compiler flags:
/usr/local/pgi6/bin/pgcc -I/usr/local/hdf5-pg/include -I/usr/local/mpich-pg/include
-c -O2 -DMAXBLOCKS=1000 -DNXB=8 -DNYB=8 -DNZB=1 -DN_DIM=2
===============================================================================

21.2 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
checkpointfileinterval = 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
converttoconsvformeshcalls = F
converttoconsvdimmeshtype = 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>
1 cm 1.0000 cm
2 s 1.0000 s
3 K 1.0000 K
4 g 1.0000 g
5 esu 1.0000 esu
6 m 100.00 cm
7 km 0.10000E+06 cm
8 pc 0.30857E+19 cm

Known physical constants:

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Constant Value</th>
<th>cm</th>
<th>s</th>
<th>g</th>
<th>K</th>
<th>esu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>0.66726E-07</td>
<td>3.00</td>
<td>-2.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>speed of light</td>
<td>0.29979E+11</td>
<td>1.00</td>
<td>-1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Multifluid database contents:

Initially defined values of species:

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
<th>Total</th>
<th>Positive</th>
<th>Neutral</th>
<th>Negative</th>
<th>bind</th>
<th>Ener</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar36</td>
<td>12</td>
<td>3.60E+01</td>
<td>1.80E+01</td>
<td>-9.99E+02</td>
<td>-9.99E+02</td>
<td>3.07E+02</td>
<td>-9.99E+02</td>
<td></td>
</tr>
<tr>
<td>c12</td>
<td>13</td>
<td>1.20E+01</td>
<td>6.00E+00</td>
<td>-9.99E+02</td>
<td>-9.99E+02</td>
<td>9.22E+01</td>
<td>-9.99E+02</td>
<td></td>
</tr>
<tr>
<td>ca40</td>
<td>14</td>
<td>4.00E+01</td>
<td>2.00E+01</td>
<td>-9.99E+02</td>
<td>-9.99E+02</td>
<td>3.42E+02</td>
<td>-9.99E+02</td>
<td></td>
</tr>
</tbody>
</table>

21.3 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_stump 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_stump which includes a data and time stamp along with a logfile message, or Logfile_stumpMessage 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 snippet shows logfile output during the evolution loop of a FLASH run.

[ 04-19-2006 16:40.43 ] [Simulation_init]: initializing Sod problem
[GRID amr_refine_derefine] initiating refinement
[GRID amr_refine_derefine] min blks 0 max blks 1 tot blks 1
[GRID amr_refine_derefine] min leaf blks 0 max leaf blks 1 tot leaf blks 1
[GRID amr_refine_derefine] refinement complete
[ 04-19-2006 16:40.43 ] [GRID gr_expandDomain]: create level=2

[ 04-19-2006 16:40.44 ] [GRID gr_expandDomain]: create level=7
[ 04-19-2006 16:40.44 ] [GRID gr_expandDomain]: create level=7
21.4 Performance Data

Finally, the log file records performance data for the simulation. The Timers unit (see Section 22.1) 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 sec</th>
<th>num calls</th>
<th>secs avg</th>
<th>time pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>1.012</td>
<td>1</td>
<td>1.012</td>
<td>4.366</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.155</td>
<td>17</td>
<td>0.009</td>
<td>0.669</td>
</tr>
<tr>
<td>writeCheckpoint</td>
<td>0.085</td>
<td>1</td>
<td>0.085</td>
<td>0.365</td>
</tr>
<tr>
<td>writePlotfile</td>
<td>0.061</td>
<td>1</td>
<td>0.061</td>
<td>0.264</td>
</tr>
<tr>
<td>evolution</td>
<td>22.176</td>
<td>1</td>
<td>22.176</td>
<td>95.633</td>
</tr>
<tr>
<td>hydro</td>
<td>18.214</td>
<td>40</td>
<td>0.455</td>
<td>78.549</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>2.603</td>
<td>80</td>
<td>0.033</td>
<td>11.227</td>
</tr>
<tr>
<td>sourceTerms</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.002</td>
</tr>
<tr>
<td>particles</td>
<td>0.000</td>
<td>40</td>
<td>0.000</td>
<td>0.001</td>
</tr>
<tr>
<td>Grid_updateRefinement</td>
<td>1.238</td>
<td>20</td>
<td>0.062</td>
<td>5.340</td>
</tr>
<tr>
<td>tree</td>
<td>1.126</td>
<td>10</td>
<td>0.113</td>
<td>4.856</td>
</tr>
<tr>
<td>guardcell tree</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.459</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.338</td>
<td>10</td>
<td>0.034</td>
<td>1.458</td>
</tr>
<tr>
<td>markRefineDerefine</td>
<td>0.339</td>
<td>10</td>
<td>0.034</td>
<td>1.460</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.053</td>
<td>10</td>
<td>0.005</td>
<td>0.230</td>
</tr>
<tr>
<td>amr_refine_derefine</td>
<td>0.003</td>
<td>10</td>
<td>0.000</td>
<td>0.011</td>
</tr>
<tr>
<td>updateData</td>
<td>0.002</td>
<td>10</td>
<td>0.000</td>
<td>0.009</td>
</tr>
<tr>
<td>guardcell</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.453</td>
</tr>
<tr>
<td>guardcell internal</td>
<td>0.337</td>
<td>10</td>
<td>0.034</td>
<td>1.452</td>
</tr>
<tr>
<td>eos</td>
<td>0.111</td>
<td>10</td>
<td>0.011</td>
<td>0.481</td>
</tr>
<tr>
<td>update particle refinemen</td>
<td>0.000</td>
<td>10</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
```
21.5 Example Usage

An example program using the Logfile unit might appear as follows:

```
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 22

Timer and Profiler Units

22.1 Timers

![Diagram of Timers directory tree]

Figure 22.1: The Timers unit directory tree.

22.1.1 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.

FLASH3 Transition

The timers described here reproduce the functionality offered in FLASH2.5 and earlier versions of the perfmon Fortran module. This Fortran module was converted to a full-fledged FLASH3 unit.
The performance routines start or stop a timer at the beginning or end of a section of code to be monitored, and accumulate performance information in dynamically assigned accounting segments. The code also has an interface to write the timing summary to the FLASH logfile. These routines are not recommended for timing very short segments of code due to the overhead in accounting.

There are two ways of using the Timers routines in your code. One mode is to simply pass timer names as strings to the start and stop routines. In this first way, a timer with the given name will be created if it doesn’t exist, or otherwise reference the one already in existence. The second mode of using the timers references them not by name but by an integer key. This technique offers potentially faster access if a timer is to be started and stopped many times (although still not recommended because of the overhead). The integer key is obtained by calling with a string name `Timers_create` which will only create the timer if it doesn’t exist and will return the integer key. This key can then be passed to the start and stop routines.

The typical usage pattern for the timers is implemented in the default Driver implementation. This pattern is: call `Timers_init` once at the beginning of a run, call `Timers_start` and `Timers_stop` around sections of code, and call `Timers_getSummary` at the end of the run to report the timing summary at the end of the logfile. However, it is possible to call `Timers_reset` in the middle of a run to reset all timing information. This could be done along with writing the summary once per-timestep to report code times on a per-timestep basis, which might be relevant, for instance, for certain non-fixed operation count solvers. Since `Timers_reset` does not reset the integer key mappings, it is safe to obtain a key through `Timers_create` once in a saved variable, and continue to use it after calling `Timers_reset`.

Two runtime parameters control the Timer unit and are described below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eachProcWritesSummary</td>
<td>LOGICAL</td>
<td>TRUE</td>
<td>Should each process write its summary to its own file? If true, each process will write its summary to a file named <code>timer_summary.&lt;process id&gt;</code></td>
</tr>
<tr>
<td>writeStatSummary</td>
<td>LOGICAL</td>
<td>TRUE</td>
<td>Should timers write the max/min/avg values for timers to the logfile?</td>
</tr>
</tbody>
</table>

monitors/Timers/TimersMain/MPINative writes two summaries to the logfile: the first gives the timer execution of the master processor, and the second gives the statistics of max, min, and avg times for timers on all processors. The secondary max, min, and avg times will not be written if some process executed timers differently than another. For example, this anomaly happens if not all processors contain at least one block. In this case, the Hydro timers only execute on the processors that possess blocks. See Section 21.4 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.

### 22.1.2 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/](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 22.1.1) 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.

22.2 Profiler

![Diagram of Profiler unit directory tree]

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 23

The Supplied Test Problems

Figure 23.1: The Simulation unit directory tree. Only some of the provided simulation implementations are shown. Users are expected to add their own simulations to the tree.

To verify that FLASH works as expected and to debug changes in the code, we have created a suite of standard test problems. Many of these problems have analytical solutions that can be used to test the accuracy of the code. Most of the problems that do not have analytical solutions produce well-defined flow features that have been verified by experiments and are stringent tests of the code. For the remaining problems, converged solutions, which can be used to test the accuracy of lower resolution simulations, are easy to obtain. The test suite configuration code is included with the FLASH source tree (in the Simulation/ directory), so it is easy to configure and run FLASH with any of these problems ‘out of the box.’ Sample runtime parameter files are also included.

23.1 Hydrodynamics Test Problems

These problems are primarily designed to test the functioning of the hydrodynamics solvers within FLASH3.
23.1.1 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

$$
\begin{align*}
\rho_L &= 1.0 & \rho_R &= 0.125 \\
p_L &= 1.0 & p_R &= 0.1
\end{align*}
$$

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 23.1 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 23.1 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 23.2 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 23.3 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.
23.1. HYDRODYNAMICS TEST PROBLEMS

Figure 23.2: 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 23.3: 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.
23.1. HYDRODYNAMICS TEST PROBLEMS

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 23.4 presents a colormap plot of the density at $t = 0.2$ in the diagonal solution together with the block structure of the AMR grid. Note that regions surrounding the discontinuities are maximally refined, while behind the shock and contact discontinuity, the grid has de-refined, because the second derivative of the density has decreased in magnitude. Because zero-gradient outflow boundaries were used for this test, some reflections are present at the upper left and lower right corners, but at $t = 0.2$ these have not yet propagated to the center of the grid.

![Figure 23.4: 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).](image)

23.1.2 Variants of the Sod Problem in Curvilinear Geometries

Variants of the Sod problems can be set up in in various geometries in order to test the handling of non-Cartesian geometries.

- An axisymmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with `./setup Sod -auto -2d -geometry=cylindrical` and using runtime parameters that include `geometry = "cylindrical"`. Use `sim_angle = 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 axi-symmetric 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.

### 23.1.3 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

\[ p_L = 1000, \quad p_M = 0.01, \quad p_R = 100 . \]  

The equation of state is that of a perfect gas with \( \gamma = 1.4 \).

Figure 23.5 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 23.5 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 23.6 shows the self-convergence of density and pressure when FLASH is run on this problem. We compare the density, pressure, and total specific energy at \( t = 0.038 \) obtained using FLASH with ten levels of refinement to solutions using several different maximum refinement levels. This figure plots the L1 error norm for each variable \( u \), defined using

\[ E(N_{\text{ref}}; u) \equiv \frac{1}{N(N_{\text{ref}})} \sum_{i=1}^{N(N_{\text{ref}})} \left| \frac{u_i(N_{\text{ref}}) - Au_i(10)}{u_i(10)} \right| , \tag{23.3} \]

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 23.5: 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 23.2. 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.
23.1. HYDRODYNAMICS TEST PROBLEMS

Figure 23.6: Self-convergence of the density, pressure, and total specific energy in the Blast2 test problem.

Table 23.2: 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 (ρ_L)</td>
</tr>
<tr>
<td>rho_mid</td>
<td>real</td>
<td>1</td>
<td>Initial density between the two interfaces (ρ_M)</td>
</tr>
<tr>
<td>rho_right</td>
<td>real</td>
<td>1</td>
<td>Initial density to the right of the right interface (ρ_R)</td>
</tr>
<tr>
<td>p_left</td>
<td>real</td>
<td>1000</td>
<td>Initial pressure to the left (p_L)</td>
</tr>
<tr>
<td>p_mid</td>
<td>real</td>
<td>0.01</td>
<td>Initial pressure in the middle (p_M)</td>
</tr>
<tr>
<td>p_right</td>
<td>real</td>
<td>100</td>
<td>Initial pressure to the right (p_R)</td>
</tr>
<tr>
<td>u_left</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the left (u_L)</td>
</tr>
<tr>
<td>u_mid</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) in the middle (u_M)</td>
</tr>
<tr>
<td>u_right</td>
<td>real</td>
<td>0</td>
<td>Initial velocity (perpendicular to interface) to the right (u_R)</td>
</tr>
<tr>
<td>xangle</td>
<td>real</td>
<td>0</td>
<td>Angle made by interface normal with the x-axis (degrees)</td>
</tr>
<tr>
<td>yangle</td>
<td>real</td>
<td>90</td>
<td>Angle made by interface normal with the y-axis (degrees)</td>
</tr>
<tr>
<td>posnL</td>
<td>real</td>
<td>0.1</td>
<td>Point of intersection between the left interface plane and the x-axis</td>
</tr>
<tr>
<td>posnR</td>
<td>real</td>
<td>0.9</td>
<td>Point of intersection between the right interface plane and the x-axis</td>
</tr>
</tbody>
</table>
23.1.4 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

$$p'_0 = \frac{3(\gamma - 1)E}{(\nu + 1)\pi \delta r^\nu},$$

(23.4)

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

$$R(t) = C_\nu(\gamma) \left( \frac{Et^2}{\rho_0} \right)^{1/(\nu+2)}.$$  

(23.5)

Here $C_\nu$ is a dimensionless constant depending only on $\nu$ and $\gamma$; for $\gamma = 1.4$, $C_2 \approx C_3 \approx 1$ to within a few percent. Just behind the shock front at $\xi = 1$ the analytical solution is

$$\begin{align*}
\rho &= \rho_1 = \frac{\gamma + 1}{\gamma - 1} \rho_0 \\
p &= p_1 = \frac{2}{\gamma + 1} \rho_0 u^2 \\
v &= v_1 = \frac{2}{\gamma + 1} u,
\end{align*}$$

(23.6)

where $u \equiv dR/dt$ is the speed of the shock wave. Near the center of the grid,

$$\begin{align*}
\rho(\xi)/\rho_1 &\propto \xi^{\nu/(\gamma - 1)} \\
p(\xi)/p_1 &= \text{constant} \\
v(\xi)/v_1 &\propto \xi.
\end{align*}$$

(23.7)

Figure 23.7 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.
23.1. HYDRODYNAMICS TEST PROBLEMS

Figure 23.7: 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 23.8 shows the pressure field in the 8-level calculation at \( t = 0.05 \) together with the block refinement pattern. Note that a relatively small fraction of the grid is maximally refined in this problem. Although the pressure gradient at the center of the grid is small, this region is refined because of the large temperature gradient there. This illustrates the ability of PARAMESH to refine grids using several different variables at once.

![Image](image_url)

Figure 23.8: Pressure field in the 2D Sedov explosion problem with 8 levels of refinement at \( t = 0.05 \). The outlines of the AMR blocks are overlaid on the pressure colormap.

We have also run FLASH on the spherically symmetric Sedov problem in order to verify the code’s performance in three dimensions. The results at \( t = 0.05 \) using five levels of grid refinement are shown in Figure 23.9. In this figure we have plotted the average values as well as the root-mean-square (RMS) deviations from the averages. As in the two-dimensional runs, the shock is spread over about two cells at the finest AMR resolution in this run. The width of the pressure peak in the analytical solution is about 1 1/2 cells at this time, so the maximum pressure is not captured in the numerical solution. Behind the shock the numerical solution average tracks the analytical solution quite well, although the Cartesian grid geometry produces RMS deviations of up to 40% in the density and velocity in the de-refined region well behind the shock. This behavior is similar to that exhibited in the two-dimensional problem at comparable resolution.

The additional runtime parameters supplied with the Sedov problem are listed in Table 23.3. This problem is configured to use the perfect-gas equation of state (\( \Gamma \)) with \( \gamma \) set to 1.4. It is simulated in a unit-sized box.

### 23.1.4.1 Sedov Self-Gravity

Another variant of the Sedov problem is included in the release which runs with spherical coordinates in one dimension. The Sedov Self-Gravity problem also allows the effects of gravitational acceleration where the gravitational potential is calculated using the multipole solver. Figure 23.10 and 23.11 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 23.9: 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 23.3: 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 23.10: Snapshots of Sedov Self-gravity density profile and gravitational potential at time $t=0.5$ sec.

Figure 23.11: Snapshots of Sedov Self-gravity density profile and gravitational potential at time $t=1.0$ sec.
23.1.5 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 vortices 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 \texttt{Grid} runtime parameters \texttt{xmin}, \texttt{xmax}, \texttt{ymin}, and \texttt{ymax}) 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

\[
\begin{align*}
T &= \frac{T_\infty}{T^*_\infty}, \\
\rho &= \rho_\infty \left( \frac{T}{T_\infty} \right)^{\frac{1}{\gamma}}.
\end{align*}
\]

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 \texttt{nx_subint} \times \texttt{ny_subint} subintervals. The runtime parameters for the isentropic vortex problem are listed in Table 23.4.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient pressure ((p_\infty))</td>
</tr>
<tr>
<td>rho_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient density ((\rho_\infty))</td>
</tr>
<tr>
<td>u_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient (x)-velocity ((u_\infty))</td>
</tr>
<tr>
<td>v_ambient</td>
<td>real</td>
<td>1.0</td>
<td>Initial ambient (y)-velocity ((v_\infty))</td>
</tr>
<tr>
<td>vortex_strength</td>
<td>real</td>
<td>5.0</td>
<td>Non-dimensional vortex strength</td>
</tr>
<tr>
<td>xctr</td>
<td>real</td>
<td>0.0</td>
<td>(x)-coordinate of vortex center</td>
</tr>
<tr>
<td>yctr</td>
<td>real</td>
<td>0.0</td>
<td>(y)-coordinate of vortex center</td>
</tr>
<tr>
<td>nx_subint</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (x)-direction</td>
</tr>
<tr>
<td>ny_subint</td>
<td>integer</td>
<td>10</td>
<td>number of subintervals in (y)-direction</td>
</tr>
</tbody>
</table>
Figure 23.12 shows the exact density distribution represented on a $40 \times 40$ uniform grid with $-5.0 \leq x, y \leq 5.0$. The borders of each grid block ($8 \times 8$ cells) are superimposed. In addition to the shaded representation, contour lines are shown for $\rho = 0.95, 0.85, 0.75,$ and $0.65$. The density distribution is radially symmetric, and the minimum density is $\rho_{\text{min}} = 0.510287$. Because the exact solution of the isentropic vortex problem is the initial solution shifted by $(u_\infty t, v_\infty t)$, numerical phase (dispersion) and amplitude (dissipation) errors are easy to identify. Dispersive errors distort the shape of the vortex, breaking its symmetry. Dissipative errors smooth the solution and flatten extrema; for the vortex, the minimum in density at the vortex core will increase.

A numerical simulation using the PPM scheme was run to illustrate such errors. The simulation used the same grid shown in Figure 23.12 with the same contour levels and color values. The grid is intentionally coarse and the evolution time long to make numerical errors visible. The vortex is represented by approximately 8 grid points in each coordinate direction and is advected diagonally with respect to the grid. At solution times of $t = 10.0, 20.0, \ldots$, the vortex should be back at its initial location.

Figure 23.13 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 23.14), and the minimum density is now $0.601786$.

Figure 23.15 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.
23.1. HYDRODYNAMICS TEST PROBLEMS

Figure 23.13: Density at $t = 50.0$ for the isentropic vortex problem.

Figure 23.14: Density at $t = 100.0$ for the isentropic vortex problem.
Figure 23.15: Representative density profiles for the isentropic vortex near $y = 0.0$ at $t = 0.0$ (black), $t = 50.0$ (red), and $t = 100.0$ (blue).

23.1.6 Wind Tunnel With a Step

The problem of a wind tunnel containing a step, WindTunnel, was first described by Emery (1968), who used it to compare several hydrodynamical methods. Woodward and Colella (1984) later used it to compare several more advanced methods, including PPM. Although it has no analytical solution, this problem is useful because it exercises a code’s ability to handle unsteady shock interactions in multiple dimensions. It also provides an example of the use of FLASH to solve problems with irregular boundaries.

The problem uses a two-dimensional rectangular domain three units wide and one unit high. Between $x = 0.6$ and $x = 3$ along the $x$-axis is a step 0.2 units high. The step is treated as a reflecting boundary, as are the lower and upper boundaries in the $y$-direction. For the right-hand $x$-boundary, we use an outflow (zero gradient) boundary condition, while on the left-hand side we use an inflow boundary. In the inflow boundary cells, we set the density to $\rho_0$, the pressure to $p_0$, and the velocity to $u_0$, with the latter directed parallel to the $x$-axis. The domain itself is also initialized with these values. We use

$$\rho_0 = 1.4, \quad p_0 = 1, \quad u_0 = 3, \quad \gamma = 1.4,$$

which corresponds to a Mach 3 flow. Because the outflow is supersonic throughout the calculation, we do not expect reflections from the right-hand boundary.

The additional runtime parameters supplied with the WindTunnel problem are listed in Table 23.5. This problem is configured to use the perfect-gas equation of state (Gamma) with $\gamma$ set to 1.4. We also set $\text{xmax} = 3$, $\text{ymax} = 1$, $\text{Nblockx} = 15$, and $\text{Nblocky} = 5$ in order to create a grid with the correct dimensions. The version of Simulation\_defineDomain supplied with this problem removes all but the first three top-level blocks along the lower edge of the grid to generate the step, and gives \text{REFLECTING} boundaries to the obstacle blocks. Finally, we use $\text{xl\_boundary\_type} = \text{"user"}$ (USER\_DEFINED condition) and $\text{xr\_boundary\_type} = \text{"outflow"}$ (OUTFLOW boundary) to instruct FLASH to use the correct boundary conditions in the $x$-direction. Boundaries in the $y$-direction are reflecting (REFLECTING).

Until $t = 12$, the flow is unsteady, exhibiting multiple shock reflections and interactions between different types of discontinuities. Figure 23.16 shows the evolution of density and velocity between $t = 0$ and $t = 4$. 
Figure 23.16: 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 23.16: Density and velocity in the Emery wind tunnel test problem (continued).
Table 23.5: 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 23.17 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 23.18 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 coarse 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 23.17: Density at $t = 4$ in the Emery wind tunnel test problem, as computed with FLASH using several different levels of refinement.
23.1. HYDRODYNAMICS TEST PROBLEMS

23.1.7 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 Table 23.6 shows values the runtime parameters values to control the amount of driving, and the Figures Figure 23.19 and Figure 23.20 show the density and x velocity profile of an xy plane in the center of the domain.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_stirmax</td>
<td>real</td>
<td>25.1327</td>
<td>maximum stirring wavenumber</td>
</tr>
<tr>
<td>st_stirmin</td>
<td>real</td>
<td>6.2832</td>
<td>minimum stirring wavenumber</td>
</tr>
<tr>
<td>st_energy</td>
<td>real</td>
<td>5.0E-6</td>
<td>energy input per mode</td>
</tr>
<tr>
<td>st_decay</td>
<td>real</td>
<td>0.5</td>
<td>correlation time for driving</td>
</tr>
<tr>
<td>st_freq</td>
<td>integer</td>
<td>1</td>
<td>frequency of stirring</td>
</tr>
</tbody>
</table>

Figure 23.18: Detail of the Kelvin-Helmholtz instability seen at $t = 3$ in the Emery wind tunnel test problem for several different levels of refinement.
Figure 23.19: Density profile for the \texttt{StirTurb} driven turbulence problem.

Figure 23.20: Velocity along X dimension for the \texttt{StirTurb} driven turbulence problem.
23.1.8 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:

\[
\begin{align*}
\rho_L &= 10 \\
\rho_R &= 1.0 \\
\rho_L &= 40/3 \\
\rho_R &= 2/3 \times 10^{-6}
\end{align*}
\]

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 23.7:

<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>

Figure 23.21, Figure 23.22, and Figure 23.23 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 23.21).
Figure 23.21: Density of numerical solution to the relativistic Sod problem at time $t = 0.36$.

Figure 23.22: Pressure of numerical solution to the relativistic Sod problem at time $t = 0.36$. 
23.1. HYDRODYNAMICS TEST PROBLEMS

23.1.9 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:

\[
(\rho, p, v_x, v_y) = \begin{cases} 
(0.5,1.0,0.0,0.0) & -1.0 \leq x < 0.0, \ -1.0 \leq y < 0.0 \\
(0.1,1.0,0.0,0.99) & 0.0 \leq x \leq 1.0, \ -1.0 \leq y < 0.0 \\
(0.1,1.0,0.99,0.0) & -1.0 \leq x < 0.0, \ 0.0 \leq y \leq 1.0 \\
(p_1,p_1,0.0,0.0) & 0.0 \leq x \leq 1.0, \ 0.0 \leq y \leq 1.0 
\end{cases},
\]

where \(p_1 = 5.477875 \times 10^{-3}\) 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 23.24 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 23.23: Normal velocity of numerical solution to the relativistic Sod problem at time \(t = 0.36\).
CHAPTER 23. THE SUPPLIED TEST PROBLEMS

Figure 23.24: 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

23.2 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 

```bash
./setup magnetoHD/BrioWu -auto -1d
```

23.2.1 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 23.1.1). 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 23.25, Figure 23.26, Figure 23.27, Figure 23.28, and Figure 23.29. 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 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 \texttt{StaggeredMesh} scheme because the currently released version of the FLASH3 code does not truly support physically correct boundary conditions for this rotated shock geometry.

Figure 23.25: Density profile for the Brio-Wu shock tube problem.

Figure 23.26: Pressure profile for the Brio-Wu shock tube problem.
Figure 23.27: Tangential magnetic field profile for the Brio-Wu shock tube problem.

Figure 23.28: Normal velocity profile for the Brio-Wu shock tube problem.
Figure 23.29: Tangential velocity profile for the Brio-Wu shock tube problem.
23.2.2 Orszag-Tang MHD Vortex

The Orszag-Tang MHD vortex problem (Orszag and Tang, 1979), magentoHD/OrszagTang, is a simple two-dimensional problem that has become a classic test for MHD codes. In this problem a simple, non-random initial condition is imposed at time $t = 0$

$$\mathbf{V} = V_0 (-\sin(2\pi y), \sin(2\pi x), 0), \quad \mathbf{B} = B_0 (-\sin(2\pi y), \sin(4\pi x), 0), \quad (x, y) \in [0, 1]^2,$$

(23.16)

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 23.30 and Figure 23.31 shows density and magnetic field contours at $t = 0.5$. As one can observe, the flow pattern at this time is already quite complicated. A number of strong waves have formed and passed through each other, creating turbulent flow features at all spatial scales.

The results were obtained using the directionally splitting SWave MHD solver for this Orszag-Tang problem.

![Density contours in the Orszag-Tang MHD vortex problem at t = 0.5.](image)

Figure 23.30: Density contours in the Orszag-Tang MHD vortex problem at $t = 0.5$. 
Figure 23.31: Magnetic field contours in the Oszag-Tang MHD vortex problem at $t = 0.5$. 
23.2.3 MHD Rotor

The two-dimensional MHD rotor problem (Balsara and Spicer, 1999), *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

$$\rho(x, y) = \begin{cases} 10 & r \leq r_0 \\ 1 + 9f(r) & r_0 < r < r_1 \\ 1 & r \geq r_1 \end{cases}$$

(23.17)

$$u(x, y) = \begin{cases} -f(r)u_0(y - 0.5)/r_0 & r \leq r_0 \\ -f(r)u_0(y - 0.5)/r & r_0 < r < r_1 \\ 0 & r \geq r_1 \end{cases}$$

(23.18)

$$v(x, y) = \begin{cases} f(r)u_0(x - 0.5)/r_0 & r \leq r_0 \\ f(r)u_0(x - 0.5)/r & r_0 < r < r_1 \\ 0 & r \geq r_1 \end{cases}$$

(23.19)

$$p(x, y) = 1$$

(23.20)

$$B_x(x, y) = \frac{5}{\sqrt{4\pi}}$$

(23.21)

$$B_y(x, y) = 0,$$

(23.22)

where $r_0 = 0.1, r_1 = 0.115, r = \sqrt{(x - 0.5)^2 + (y - 0.5)^2}, w = B_z = 0$ and a taper function $f(r) = (r_1 - r)/(r - r_0)$. The value $\gamma = 1.4$ is used. The initial set-up is occupied by a dense rotating disk at the center of the domain, surrounded by the ambient flow at rest with uniform density and pressure. The rapidly spinning rotor is not in an equilibrium state due to the centrifugal forces. As the rotor spins with the given initial rotating velocity, the initially uniform magnetic field in $x$-direction will wind up the rotor. The rotor will be wrapped around by the magnetic field, and hence start launching torsional Alfvén waves into the ambient fluid. The angular momentum of the rotor will be diminished in later times as the evolution time increases. The circular rotor will be progressively compressed into an oval shape by the build-up of the magnetic pressure around the rotor. The results shown in Figure 23.32 were obtained using the *StaggeredMesh* MHD solver using 6 refinement levels. The divergence free evolution of the magnetic fields are well preserved as illustrated in Figure 23.33.
Figure 23.32: The Rotor problem at $t = 0.15$ (a) Density (b) Pressure (c) Mach number (d) Magnetic pressure.
Figure 23.33: Divergence of magnetic fields using the \texttt{StaggeredMesh} solver at $t = 0.15$ for the Rotor problem.
23.2.4 MHD Current Sheet

The two-dimensional current sheet problem, \texttt{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:

$$B_y = \begin{cases} 
B_0 & 0.0 \leq x < 0.5 \\
-B_0 & 0.5 \leq x < 1.5 \\
B_0 & 1.5 \leq x \leq 2.0 
\end{cases} \quad (23.23)$$

where $B_0 = 1$. The other magnetic field components $B_x, B_z$ are set to be zeros. The $x$ component of the velocity is given by $u = u_0 \sin 2\pi y$ with $u_0 = 0.1$, and all the other velocity components are initialized with zeros. The density is unity and the gas pressure $p = 0.1$.

The changes of the magnetic fields seed the magnetic reconnection and develop formations of magnetic islands along the two current sheets. The small islands are then merged into the bigger islands by continuously shifting up and down along the current sheets until there is one big island left in each current sheet.

The temporal evolution of the magnetic field lines from $t = 0.0$ to $t = 5.0$ are shown in Figure 23.34(a) – 23.34(f) on a $256 \times 256$ uniform grid. In Figure 23.35 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 \texttt{StaggeredMesh} solver was used for this problem.
Figure 23.34: 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 23.35: Current density at $t = 4.0$ using the StaggeredMesh solver for the MHD CurrentSheet problem.
23.2.5 Field Loop

The 2D and 3D field loop advection problems (magnetoHD/FieldLoop) are known to be stringent test cases in multidimensional MHD. In this test problem we consider a 2D advection of a weakly magnetized field loop traversing the computational domain diagonally. Details of the problem have been described in Gardiner and Stone (2005).

The computational domain is $[-1,1] \times [-0.5,0.5]$, with a grid resolution $256 \times 148$, and doubly-periodic boundary conditions. With this rectangular grid cell, the flow is not symmetric in $x$ and $y$ directions because the field loop does not advect across each grid cell diagonally and hence the resulting fluxes are different in $x$ and $y$ directions. The density and pressure are unity everywhere and $\gamma = 5/3$. The velocity fields are defined as,

$$\mathbf{U} = u_0(\cos\theta, \sin\theta, 1)\quad (23.24)$$

with the advection angle $\theta$, given by $\theta = \tan^{-1}(0.5) \approx 26.57^\circ$. For the choice of the initial advection velocity we set $u_0 = \sqrt{5}$. The size of domain and other parameters were chosen such that the weakly magnetized field loop makes one complete cycle by $t = 1$. It is important to initialize the magnetic fields to satisfy $\nabla \cdot \mathbf{B} = 0$ numerically in order to avoid any initial nonzero error in $\nabla \cdot \mathbf{B}$. As suggested in Gardiner and Stone (2005), the magnetic field components are initialized by taking the numerical curl of the $z$-component of the magnetic vector potential $A_z$,

$$B_x = \frac{\partial A_z}{\partial y}, \quad B_y = -\frac{\partial A_z}{\partial x}, \quad (23.25)$$

where

$$A_z = \begin{cases} A_0(R - r) & r \leq R \\ 0 & \text{otherwise}. \end{cases}\quad (23.26)$$

By using this initialization process, divergence-free magnetic fields are constructed with a maximum value of $\nabla \cdot \mathbf{B}$ in the order of $10^{-16}$ at the chosen resolution. The parameters in (23.26) 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 $\mathbf{A}_{B_x}$ and $\mathbf{A}_{B_y}$, are explicitly included in the data reconstruction-evolution algorithm in the USM scheme (see Lee and Deane, 2008). During the advection a good numerical scheme should maintain: (a) the circular symmetry of the loop at all time: a numerical scheme that lacks proper numerical dissipation results in spurious oscillations at the loop, breaking the circular symmetry; (b) $B_z = 0$ during the simulation: $B_z$ will grow proportional to $w\nabla \cdot \mathbf{B} \Delta t$ if a numerical scheme does not properly include multidimensional MHD terms.

From the results in Figure 23.36, 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.

23.2.6 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 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
23.3 Gravity Test Problems

23.3.1 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 \left[ 1 + \delta \cos(k \cdot x) \right] \\
p(x) &= \rho_0 \left[ 1 + \gamma \delta \cos(k \cdot x) \right] \\
v(x) &= 0 ,
\end{align*}

where the perturbation amplitude $\delta \ll 1$. The stability of the perturbation is determined by the relationship between the wavenumber $k \equiv |k|$ and the Jeans wavenumber $k_J$, where $k_J$ is given by

\begin{equation}
k_J \equiv \frac{\sqrt{4\pi G \rho_0}}{c_0} ,
\end{equation}

and where $c_0$ is the unperturbed adiabatic sound speed

\begin{equation}
c_0 = \sqrt{\frac{\gamma \rho_0}{\rho_0}}
\end{equation}

Figure 23.36: The field loop advection problem using the StaggeredMesh solver at time $t = 2$ with the Roe Riemann solver. (a) $B_p$ with the MEC at $t = 2$. The color scheme between $2.32 \times 10^{-25}$ and $7.16 \times 10^{-7}$ was used. (b) Magnetic field lines with the MEC at $t = 2$. 20 contour lines of $A_z$ between $-2.16 \times 10^{-6}$ and $2.7 \times 10^{-4}$ are shown.

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 23.37, the explosion now becomes highly anisotropic as shown in the pressure plot in Figure 23.37. 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.
If \( k > k_J \), the perturbation is stable and oscillates with frequency

\[
\omega = \sqrt{c_s^2 k^2 - 4\pi G \rho_0} ;
\]

(23.30) otherwise, it grows exponentially, with a characteristic timescale given by \( \tau = (i\omega)^{-1} \).

We checked the dispersion relation (23.30) 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

\[
L = \frac{1}{2} \sqrt{\frac{\pi \gamma p_0}{G \rho_0^2}} .
\]

(23.31) This prevents roundoff errors at wavenumbers less than \( k_J \) from being amplified by the physical Jeans instability. We used wavevectors \( \mathbf{k} \) parallel to and at 45 degrees to the \( x \)-axis. Each test calculation used the multigrid Poisson solver together with its default settings.

The resulting kinetic, thermal, and potential energies as functions of time for one choice of \( \mathbf{k} \) are shown in Figure 23.38 together with the analytic solution, which is given in two dimensions by

\[
T(t) = \frac{\rho_0 \delta^2 |\omega|^2 L^2}{8k^2} [1 - \cos(2\omega t)]
\]

\[
U(t) - U(0) = -\frac{1}{8} \rho_0 c_s^2 \delta^2 L^2 [1 - \cos(2\omega t)]
\]

\[
W(t) = -\frac{\pi G \rho_0^2 \delta^2 L^2}{2k^2} [1 + \cos(2\omega t)] .
\]

(23.32)
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 23.39 compares the resulting dispersion relation to (23.30). It can be seen from this plot that FLASH correctly reproduces (23.30). 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 23.38: 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 23.8. 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 23.39: Computed versus expected Jeans dispersion relation (for stable modes) found using FLASH 1.62 with a six-level uniform grid.

Table 23.8: 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>lmbdax</td>
<td>real</td>
<td>0.572055</td>
<td>Perturbation wavelength in $x$ direction ($\lambda_x = 2\pi/k_x$)</td>
</tr>
<tr>
<td>lmbday</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>lmbdaz</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</td>
</tr>
</tbody>
</table>

$$\max_{\text{block}} \left\{ \left| \frac{\rho_{ijk}}{\rho_{\text{ref}}} - 1 \right| \right\}$$
23.3.2 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

\[
p_0 \ll \frac{4\pi G}{\gamma} \rho_0^2 r_0^2. \tag{23.33}
\]

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

\[
\left( \frac{8\pi G}{3} \rho_0 \right)^{1/2} t = \left( 1 - \frac{r(t)}{r_0} \right)^{1/2} \left( \frac{r(t)}{r_0} \right)^{1/2} + \sin^{-1} \left( 1 - \frac{r(t)}{r_0} \right)^{1/2} \tag{23.34}
\]

(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 23.40, 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 23.41a, 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 \( 7.30 \times 10^8 \text{ cm s}^{-1} \), respectively. In Figure 23.41b 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 (23.34). Overall our results, including the numerical effects, agree well with those of Mönchmeyer and Müller (1989).

This problem is configured to use the perfect-gas equation of state (gamma) with gamma set to 1.67 and is run in a three-dimensional box. The problem uses the specialized refinement marking routine supplied under the Grid interface of Grid_markRefineSpecialized which refines blocks containing the center of the cloud.
CHAPTER 23. THE SUPPLIED TEST PROBLEMS

Figure 23.40: 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 23.41: 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.
23.3.3 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

$$\rho(x,y) = \sum_{i=1}^{13} e^{-\sigma_i[(x-x_i)^2 + (y-y_i)^2]} ,$$  \hspace{1cm} (23.35)

where the constants $\sigma_i$, $x_i$, and $y_i$ are given in Table 23.9. 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 23.42.

Figure 23.42: 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.

<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>

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 8.9.2.2.
23.3.4 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 FLASH3 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 FLASH3 \texttt{smlrho} is used). The spheroid is motionless and in hydrostatic equilibrium. The gravitational potential of such object is analytically calculable, and is:

\[
\phi(x) = \pi G \rho \left[ 2A_1 a_1^2 - A_1 (x^2 + y^2) + A_3 (a_3^2 - z^2) \right],
\]

for a point inside the spheroid. Here

\[
A_1 = \frac{\sqrt{1 - e^2}}{e^3} \sin^{-1} e - \frac{1 - e^2}{e^2},
\]

\[
A_3 = \frac{2}{e^2} - \frac{2 \sqrt{1 - e^2}}{e^3} \sin^{-1} e,
\]

where \(e\) is the ellipticity of a spheroid:

\[
e = \sqrt{1 - \left( \frac{a_3}{a_1} \right)^2}.
\]

For a point outside the spheroid, potential is:

\[
\phi(x) = \frac{2 a_3}{e^2} \pi G \rho \left[ a_1 e \tan^{-1} h - \frac{1}{2} \left( (x^2 + y^2) \left( \tan^{-1} h - \frac{h}{1 + h^2} \right) + 2z^2 (h - \tan^{-1} h) \right) \right],
\]

where

\[
h = \frac{a_1 e}{\sqrt{a_3^2 + \lambda}},
\]

and \(\lambda\) is the positive root of the equation

\[
\frac{x^2}{a_1^2 + \lambda} + \frac{y^2}{a_2^2 + \lambda} + \frac{z^2}{a_3^2 + \lambda} = 1.
\]

This test is also useful because the spheroid has spherical symmetry in the X–Y plane, but also lack of such symmetry in X–Z and Y–Z planes. The density distribution of the spheroid is shown in Equation 23.3.4. 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 \texttt{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:

\[
\epsilon = \left| \frac{\phi_{\text{analytical}} - \phi_{\text{FLASH}}}{\phi_{\text{analytical}}} \right|
\]

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 \(l_{\max}\), the cutoff \(l\) in (8.15). In Figure 23.44–23.44 the gravitational potential for the Maclaurin spheroid, the FLASH3 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 23.44, indicating that the last dipole term in the multipole expansion does not contribute to the accuracy of the solution but does increase computational
23.3. GRAVITY TEST PROBLEMS

Figure 23.43: Density of the MacLaurin spheroid (left X–Y plane, right Y–Z plane) with ellipticity \( e = 0.9 \). The FLASH3 block structure is shown on top.

Table 23.10: Minimal and maximal relative error in all zones of the simulation, calculated using (23.43). 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^2_{\text{in}} ) norm</th>
<th>( L^2_{\text{out}} ) norm</th>
<th>approx. time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 4.5 \times 10^{-6} )</td>
<td>0.182</td>
<td>( 7.1 \times 10^{-2} )</td>
<td>( 6.8 \times 10^{-2} )</td>
<td>9.8</td>
</tr>
<tr>
<td>1</td>
<td>( 4.5 \times 10^{-6} )</td>
<td>0.182</td>
<td>( 7.1 \times 10^{-2} )</td>
<td>( 6.8 \times 10^{-2} )</td>
<td>14.5</td>
</tr>
<tr>
<td>2</td>
<td>( 9.8 \times 10^{-6} )</td>
<td>0.062</td>
<td>( 1.4 \times 10^{-2} )</td>
<td>( 1.7 \times 10^{-2} )</td>
<td>34.7</td>
</tr>
<tr>
<td>4</td>
<td>( 1.0 \times 10^{-8} )</td>
<td>0.026</td>
<td>( 4.0 \times 10^{-3} )</td>
<td>( 5.0 \times 10^{-3} )</td>
<td>55.4</td>
</tr>
<tr>
<td>6</td>
<td>( 6.1 \times 10^{-9} )</td>
<td>0.013</td>
<td>( 1.6 \times 10^{-3} )</td>
<td>( 2.5 \times 10^{-3} )</td>
<td>134.9</td>
</tr>
<tr>
<td>8</td>
<td>( 7.8 \times 10^{-9} )</td>
<td>0.007</td>
<td>( 8.7 \times 10^{-4} )</td>
<td>( 1.2 \times 10^{-3} )</td>
<td>210.2</td>
</tr>
<tr>
<td>10</td>
<td>( 6.7 \times 10^{-9} )</td>
<td>0.004</td>
<td>( 5.5 \times 10^{-4} )</td>
<td>( 7.0 \times 10^{-4} )</td>
<td>609.7</td>
</tr>
</tbody>
</table>

Because gravity sources are all of the same sign, and the symmetry of the problem, all odd-\( l \) moments are zero: reasonable, physically motivated values for setting \( \text{mpole}_l \text{max} \) should be an even number.

In the X–Y plane, where the solution is radially symmetric, the first monopole term is enough to qualitatively capture the correct potential. As expected, the error is the biggest on the spheroid boundary, and decreases both outwards and inwards. Increasing the maximum included moment reduces errors. However, in other non-symmetric planes, truncating the potential to certain \( l_{\text{max}} \) leads to an error whose leading term will be the spherical harmonic of order \( l_{\text{max}} + 2 \), as can be nicely seen in the lower right sections of Figure 23.44 – 23.46. 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 23.44: Maclaurin spheroid: $l_{max} = 0$, 6 refinement levels. Left column is X–Y plane, cut through $z=0.5$, right column is Y–Z plane cut through $x=0.5$. From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.
Figure 23.45: 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 23.46: 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.
23.4 Particles Test Problems

These problems are primarily designed to test the functioning of the particle tracking routines within FLASH3.

23.4.1 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

\[
|v| = \sqrt{\frac{2GM}{r_0}},
\]

if a constant gravitational field due to a point mass \( M \) at \((0.5L_x, 0.5L_y, 0.5L_z)\) is employed, or

\[
|v| = \frac{1}{2} \sqrt{\frac{2G}{r_0}},
\]

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 23.47 shows a typical pair of particle trajectories for this problem.

![Particle trajectories in the Orbit test problem for a 3D grid at a fixed refinement level of 2. There is no motion along the z-axis.](image)

No specific gravity unit is required by the problem configuration file, because the problem is intended to be run with either a fixed external field or the particles’ own field. If the particles are to orbit in an external field (\texttt{ext\_field = .true.}), the field is assumed to be a central point-mass field (\texttt{physics/Gravity/GravityMain/PointMass}), and the parameters for that unit should be assigned appropriate values. If the particles are self-gravitating (\texttt{ext\_field = .false.}), the \texttt{physics/Gravity/GravityMain/Poisson}
unit should be included in the code, and a Poisson solver that supports isolated boundary conditions should be used (grav_boundary_type = "isolated").

In either case, long-range forces for the particles must be turned on, or else they will not experience any accelerations at all. This can be done using the particle-mesh method by including the unit Particles/ParticlesMain/active/longRange/gravity/ParticleMesh.

FLASH3 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 FLASH3, 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 23.11 in addition to the regular ones supplied with the code.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>separation</td>
<td>real</td>
<td>0.5</td>
<td>Initial particle separation ($r_0$)</td>
</tr>
<tr>
<td>ext_field</td>
<td>logical</td>
<td>.false.</td>
<td>Whether to make the particles self-gravitating or to have them orbit in an external potential. In the former case GravityMain/Poisson should be used; in the latter, GravityMain/PointMass.</td>
</tr>
</tbody>
</table>

23.4.2 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
23.4. PARTICLES TEST PROBLEMS

$k$ which collapses to a caustic at redshift $z_c < z$ has comoving density and velocity given by

$$\rho(x_c; z) = \tilde{\rho} \left[ 1 + \frac{1 + z_c}{1 + z} \cos(kx_\ell) \right]^{-1}$$  \hspace{1cm} (23.46)

$$v(x_c; z) = - H_0 (1 + z)^{1/2} \frac{1 + z_c}{1 + z} \frac{\sin(kx_\ell)}{k}$$  \hspace{1cm} (23.47)

where $\tilde{\rho}$ is the comoving mean density. Here $x_c$ is the distance of a point from the pancake midplane, and $x_\ell$ is the corresponding Lagrangian coordinate, found by iteratively solving

$$x_c = x_\ell - \frac{1 + z_c}{1 + z} \frac{\sin(kx_\ell)}{k}.$$  \hspace{1cm} (23.48)

The temperature solution is determined from the density under the assumption that the gas is adiabatic with ratio of specific heats $\gamma$:

$$T(x_c; z) = (1 + z)^2 \bar{T}_{\text{fid}} \left( \frac{1 + z_{\text{fid}}}{1 + z} \right)^3 \rho(x_c; z_{\text{fid}}) \rho(x_c; z)^{\gamma^{-1}}.$$  \hspace{1cm} (23.49)

The mean temperature $\bar{T}_{\text{fid}}$ is specified at a redshift $z_{\text{fid}}$.

Dark matter particles are initialized using the same solution as the gas. The Lagrangian coordinates $x_\ell$ are assigned to lie on a uniform grid. The corresponding perturbed coordinates $x_c$ are computed using (23.48). Particle velocities are assigned using (23.47).

At caustic formation ($z = z_c$), planar shock waves form in the gas on either side of the pancake midplane and begin to propagate outward. A small region at the midplane is left unshocked. Immediately behind the shocks, the comoving density and temperature vary approximately as

$$\rho(x_c; z) \approx \tilde{\rho} \frac{18}{(kx_\ell)^2} \frac{(1 + z_c)^3}{(1 + z)^3}$$  \hspace{1cm} (23.50)

$$T(x_c; z) \approx \frac{\mu H_0^2}{6 k_B k^2} \frac{(1 + z_c) (1 + z)^2}{(kx_\ell)^2}.$$  

At the midplane, which undergoes adiabatic compression, the comoving density and temperature are approximately

$$\rho_{\text{center}} \approx \tilde{\rho} \frac{1 + z_{\text{fid}}}{1 + z} \left[ \frac{3 H_0^2 \mu}{k_B \bar{T}_{\text{fid}} k^2} \frac{(1 + z_c)^4}{1 + z_{\text{fid}}} \right]^{\gamma^{-1}}$$  \hspace{1cm} (23.51)

$$T_{\text{center}} \approx \frac{3 H_0^2 \mu}{k_B k^2} \frac{(1 + z)^2}{(1 + z_c)^4} \frac{\tilde{\rho}}{\rho_{\text{center}}}.$$  

An example FLASH calculation of the post-caustic gas solution appears in Figure 23.48.

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 23.49.

The 2D pancake problem in FLASH3 uses the runtime parameters listed in Table 23.12 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 23.48: Example solution for the gas in a mixed particle/gas Zel’dovich Pancake problem. A comoving wavelength $\lambda = 10$ Mpc, caustic redshift $z_c = 5$, fiducial redshift $z_{\text{fid}} = 200$, and fiducial temperature $T_{\text{fid}} = 550$ 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 23.49: Example solution for the dark matter in a mixed particle/gas Zel’dovich pancake. Perturbation and cosmological parameters were the same as in Figure 23.48. 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 23.12: 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 $(T_{\text{fid}})$</td>
</tr>
<tr>
<td>zfiducial</td>
<td>real</td>
<td>200.</td>
<td>Redshift at which gas temperature is $T_{\text{fid}}$ $(z_{\text{fid}})$</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>
23.4.3 Modified Huang-Greengard Poisson Test

The PoisParticles problem is designed to generate a refined grid from the distribution of particles in the computational domain. In other words, create a grid which is more refined in places where there is a clustering of particles. It is a modified form of the PoisTest simulation described in Section 23.3.3. Recall that the PoisTest problem involves the creation of a highly refined grid, which is used to test grid refinement.

In the PoisParticles problem, the density stored in the grid is used as an indicator of where to create new particles. Here, the number of particles created in each region of the grid is proportional to the grid density, i.e., more particles are created in regions where there is a high density. Each new particle is assigned a mass, which is taken from the density in the grid, so that mass is conserved.

The flash.par parameters shown in Table 23.13 specify that the grid should refine until at most 5 particles exist per block. This creates a refined grid similar to the PoisTest problem.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refine_on_particle_count</td>
<td>logical</td>
<td>.true.</td>
<td>On/Off flag for refining the grid according to particle count.</td>
</tr>
<tr>
<td>max_particles_per_blk</td>
<td>integer</td>
<td>5</td>
<td>Grid refinement criterion which specifies maximum number of particles per block.</td>
</tr>
</tbody>
</table>

23.5 Burn Test Problem

23.5.1 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 $\alpha$-chain plus heavy-ion reaction network is used in the calculations. A definition of what we mean by an $\alpha$-chain reaction network is prudent. A strict $\alpha$-chain reaction network is only composed of $(\alpha, \gamma)$ and $(\gamma, \alpha)$ links among the 13 isotopes $^4$He, $^{12}$C, $^{16}$O, $^{20}$Ne, $^{24}$Mg, $^{28}$Si, $^{32}$S, $^{36}$Ar, $^{40}$Ca, $^{44}$Ti, $^{48}$Cr, $^{52}$Fe, and $^{56}$Ni. It is essential, however, to include $(\alpha, p)(p, \gamma)$ and $(\gamma, p)(p, \alpha)$ links in order to obtain reasonably accurate energy generation rates and abundance levels when the temperature exceeds $\sim 2.5 \times 10^9$ K. At these elevated temperatures the flows through the $(\alpha, p)(p, \gamma)$ sequences are faster than the flows through the $(\alpha, \gamma)$ channels. An $(\alpha, p)(p, \gamma)$ sequence is, effectively, an $(\alpha, \gamma)$ reaction through an intermediate isotope. In our $\alpha$-chain reaction network, we include 8 $(\alpha, p)(p, \gamma)$ sequences plus the corresponding inverse sequences through the intermediate isotopes $^{27}$Al, $^{31}$P, $^{35}$Cl, $^{39}$K, $^{43}$Sc, $^{47}$V, $^{51}$Mn, and $^{55}$Co by assuming steady state proton flows.

The two-dimensional calculations are performed in a planar geometry of size 256.0 cm by 25.0 cm. The initial conditions consist of a constant density of $10^7$ g cm$^{-3}$, temperature of $2 \times 10^8$ K, composition of pure carbon $X(^{12}$C)=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 $4.423 \times 10^9$ 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 23.14.

Table 23.14: Runtime parameters used with the Cellular test problem.

<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 23.50 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 23.51 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 23.50: Steady-state conditions of the Cellular nuclear burn problem.
Figure 23.51: Close-up of the detonation front in steady-state for the **Cellular** nuclear burn problem.
23.6 Other Test Problems

23.6.1 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)).

![Image](image.png)

Figure 23.52: Temperature profile assumed for the test.

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 23.52). 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

$$\left[n^Z_i\right]_{eq} = \left[n^Z_{i+1}\right]_{eq} \alpha_{i+1}^Z \quad (i = 1, \ldots, l_Z - 1) \quad (23.52)$$

$$\sum_{i=1}^{l_Z} \left[n^Z_i\right]_{eq} = A_Z n_p \quad (23.53)$$

The presence of a temperature jump causes a strong pressure difference, which in turn should cause significant plasma motions. Since the purpose is to test the NEI module, it is imposed that the pressure difference does not induce any plasma motion and, to this end, the hydro variables (namely, $T$, $\rho$, $v$) are not updated. In practice, the continuity equations are solved with uniform density and velocity, while the momentum and energy equations are ignored.

Figure 23.53 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 23.53: 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 23.53: ... continued ...
Figure 23.54: As in Figure 23.53 for the solutions of the FLASH code.
Figure 23.54: ... 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 23.54 shows the population fraction vs. space after 700 s.

### 23.6.2 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

$$T(x, t_{ini}) = Q\delta(x),$$  \hspace{1cm} (23.54)

then

$$T(x, t) = \frac{Q}{(4\pi\chi(t-t_{ini}))^{1/2}}e^{-x^2/4\chi(t-t_{ini})}$$  \hspace{1cm} (23.55)

(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.

### 23.6.3 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.
Figure 23.55: Temperature profile of the Delta-Function Heat Conduction Problem at two times. $t_{\text{ini}} = -1\text{ ms}$, top: $t = 0\text{ ms}$, bottom: $t = 2.46\text{ ms}$. 

-time = 0.002 ps
-number of blocks = 14
-AMR levels = 3

-time = 0.002 a
-number of blocks = 14
-AMR levels = 3
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.

### 23.7 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/](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/](http://flash.uchicago.edu/website/codesupport/visit/). Full instructions are also available at that site.
Chapter 24

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.

24.1 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.

24.2 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:
"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 25

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 FLASH3. 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.

25.1 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.

<table>
<thead>
<tr>
<th>FLASH3 Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that FLASH3 no longer supports fidlr2 routines. Also, the names of the basic environment variables and routines have been changed to require the user to update their system configuration and old habits. For example, the environment variable was called XFLASH_DIR, now it is called XFLASH3_DIR. Similarly, the basic routine xflash has been changed to xflash3 to avoid confusion. The suffix 3.0 on the fidlr3.0 directory has been added to emphasize that these routines work with FLASH3.</td>
</tr>
</tbody>
</table>

307
25.1.1 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.

25.1.2 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
```

25.2 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 25.1.

```
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 FLASH3. These sections are explained below.
25.2. **XFLASH3: A WIDGET INTERFACE TO PLOTTING FLASH DATASETS**

![Figure 25.1: The main xflash3 widget.](image)

**25.2.1 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.* `hdf5chk`) 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.

**25.2.1.1 File/Open prototype...**

The Open prototype... menu option will bring up the file selection dialog box (see Figure 25.2). 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 25.3).

*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.

**25.2.2 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.

25.2.3 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.

25.2.4 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.

25.2.5 Plotting options available from the GUI

Various options are available on the xflash3 user interface to change the appearance of the plots.

25.2.5.1 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.
25.2.5.2 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.

25.2.5.3 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.

25.2.5.4 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
Table 25.1: \textit{xflash3} options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>Plot the base-10 log of the variable.</td>
</tr>
<tr>
<td>max</td>
<td>When a sequence of files is defined in the file options, plot the maximum of the variable in each cell over all the files.</td>
</tr>
<tr>
<td>annotate</td>
<td>Toggle the title and time information on the plot.</td>
</tr>
<tr>
<td>show ticks</td>
<td>Show the axis tick marks on the plot.</td>
</tr>
<tr>
<td>abs. value</td>
<td>Plot the absolute value of the dataset. This operation is performed before taking the log.</td>
</tr>
<tr>
<td>show blocks</td>
<td>Draw the zone boundaries on the plot.</td>
</tr>
<tr>
<td>colorbar</td>
<td>Plot the colorbar legend for the data range.</td>
</tr>
</tbody>
</table>

derived variables should look at how the total velocity ($\text{tot.vel}$) is computed.

25.2.5.5 Options

The options block allows you to toggle various options on/off. Table Table 25.1 lists the various options available.

25.2.5.6 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 \textit{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.

25.2.5.7 Slice Plane

The slice plane group is only active for 3-d datasets. This option allows you to select a plane for plotting in ($x$-$y$, $x$-$z$, $y$-$z$).

25.2.5.8 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 \textit{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 \textit{reset} button will reset the zoom limits.

25.2.6 Plotting buttons

Several buttons are located at the bottom of the \textit{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.

25.2.6.1 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 25.4). 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.
25.2. **XFLASH3: A WIDGET INTERFACE TO PLOTTING FLASH DATASETS**

25.2.6.2 **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 25.5). 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 which to scale the vectors, and *minimum vector* and *maximum vector* specify the range of vectors for which to plot. *zskip* and *yskip* allow you to thin out the arrows. This option is available in 2-d only.

25.2.6.3 **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 FLASH3, this option is disabled in xflash3.

25.2.6.4 **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.

25.2.6.5 **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.

25.2.6.6 **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.

25.2.6.7 **Histogram Button**

Create a histogram of the data. This option is disabled in this release of FLASH3.
25.2.6.8 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 25.6). Use the Close button to dismiss the results.

25.2.6.9 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.

25.3 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.

FLASH3 Transition

The `diff3` command can be used to compare two-dimensional output from FLASH2 simulations to FLASH3 simulations, since the `fidlr3.0` routines are backwards-compatible. The physical structure of the simulations must be identical.
Figure 25.6: The xflash3 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
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
Lee, D. 2006, Ph.D. Dissertation, Univ. of Maryland
25.3. COMPARING TWO DATASETS

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
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, 182, 185
- enucDfFactor, 187
- odeStepper, 182, 186
- useBurnTable, 187
- useShockBurn, 186

Conductivity
- useConductivity, 218

Cosmology
- CosmologicalConstant, 215
- MaxScaleChange, 215
- OmegaMatter, 215
- OmegaRadiation, 215

Diffuse
- DiffuseFluxBased
  - diff_scaleFactThermFlux, 193
- DiffuseMain
  - diff_scaleFactThermFlux, 193
  - diff_scaleFactThermSaTime, 193
  - diff_XIBoundaryType, 193
  - diff_XrBoundaryType, 193
  - diff_YIBoundaryType, 193
  - diff_YrBoundaryType, 193
  - diff_ZIBoundaryType, 193
  - diff_ZrBoundaryType, 193
  - dt_diff_factor, 193
- useDiffuse, 192, 193
- useDiffuseTherm, 192
- useDiffuseVisc, 192

Driver
- dtmax, 211
- nend, 121
- restart, 122
- tmax, 121

Eos
- eintSwitch, 156
- eos_coulumbMult, 178
- eos_maxNewton, 178
- eos_singleSpeciesA, 177
- eos_singleSpeciesZ, 177
- eos_tolerance, 178
- gamma, 177, 234, 244, 250

Gravity
- grav_boundary_type, 79

Grid
- convertToConsVdForMeshCalls, 50, 51, 87
- convertToConsVdInMeshInterp, 50, 51, 86, 87
- derefine_cutoff#, 88
- flux_correct, 82
- geometry, 111
- gr_lrefineMaxRedDoByLogR, 89
- gr_lrefineMaxRedDoByTime, 89
- gr_lrefineMaxRedLogBase, 89
- gr_lrefineMaxRedRadiusFact, 89
- gr_lrefineMaxRedTimeScale, 89
- gr_lrefineMaxRedTRef, 89
- iGridSize, 83
- interpol_order, 86
- jGridSize, 83
- kGridSize, 83
- lrefine_max, 89
- max_particles_per_blk, 88
- mpole_lmax, 280, 281
- mpole_subSample, 103, 104
- Nblockx, 78, 250
- nblockx, 77
- Nblocky, 78, 250
- nblocky, 77
- Nblockz, 78
- nblockz, 77
- pfft_setupOnce, 100
- refine_cutoff#, 88
- refine_filter#, 88
- refine_on_particle_count, 88
- refine_var#, 87, 88
- smlrho, 280
- xl_boundary_type, 78, 79, 250
- xmax, 77, 247, 250
- xmin, 77, 247
- xr_boundary_type, 250
- ymax, 77, 247, 250
- ymin, 77, 247
- zmax, 77
- zmin, 77
Hydro

hybrid_riemann, 158
ppm_modifystates, 158
use_cma_flattening, 158
use_steepening, 158

IO

alwaysComputeUserVars, 136
alwaysRestrictCheckpoint, 136
checkpointFileIntervalStep, 140
checkpointFileNumber, 122
chkGuardCellsInput, 130
chkGuardCellsOutput, 130
outputSplitNum, 135
particleFileNumber, 123
plot_grid_var_1, 128
plot_grid_var_2, 128
plot_var_1, 123
plot_var_2, 123
plotFileNumber, 142
plotfileNumber, 123

Ioniz

dneimax, 189
dneimin, 189
tneimax, 189
tneimin, 189

Logfile

log_file, 221

Particles

particle_attribute_1, 210
particle_attribute_2, 210
pt_dtChangeToleranceDown, 205
pt_dtChangeToleranceUp, 205
pt_dtFactor, 210
pt_initialXMin, 210
pt_numX, 210
pt_numX, 140
pt_numY, 140
useParticles, 210

PhysicalConstants

cp_unitsBase, 150

Simulation

sim_maxTolCoeff0, 211
sim_maxTolCoeff1, 211
sim_maxTolCoeff2, 211
sim_schemeOrder, 211
sim_smlRho, 279

Stir

st_decay, 189
st_energy, 189

Timers

eachProcWritesSummary, 228
writeStatSummary, 228

Viscosity

useViscosity, 218
visc_whichCoefficientIsConst, 218
FLASH3 API

Driver
- Driver_abortFlash, 69, 128, 141, 224
- Driver_abortFlashC, 69
- Driver_evolveFlash, 33, 36, 68, 192, 193, 203
- Driver_finalizeFlash, 69
- Driver_getDt, 69
- Driver_get_elapsedWCTime, 69
- Driver_getNStep, 69
- Driver_getSimTime, 69
- Driver_initFlash, 34, 121, 140, 150
- Driver_sendOutputData, 127
- Driver_evolveFlash, 193
- Driver_initFlash, 19

Grid
- Grid_applyBCEdge, 80
- Grid_applyBCEdgeAllUnkVars, 80
- Grid_bcApplyToRegion, 80
- Grid_bcApplyToRegionSpecialized, 80
- Grid_fillGuardCells, 32, 81, 84, 85, 91, 95
- Grid_getBlkBoundBox, 85
- Grid_getBlkCenterCoords, 85, 86, 112
- Grid_getBlkCenterID, 85
- Grid_getBlkData, 32, 90, 112
- Grid_getBlkIndexLimits, 23, 91
- Grid_getBlkPhysicalSize, 86, 112
- Grid_getBlkPtr, 90
- Grid_getBlkRefineLevel, 85
- Grid_getBlkType, 85
- Grid_getCellCoords, 23, 32, 86, 111
- Grid_getDeltas, 112
- Grid_getGeometry, 111
- Grid_getListOfBlocks, 33, 90
- Grid_getPlaneData, 90, 112
- Grid_getPointData, 90, 112
- Grid_getRowData, 90, 112
- Grid_getSingleCellVol, 112
- Grid_init, 77, 90
- Grid_initDomain, 77, 90
- Grid_MapParticlesToMesh, 198
- Grid_mapParticlesToMesh, 209
- Grid_markRefineDerefine, 17, 89
- Grid_markRefineSpecialized, 89
- Grid_pfft, 100
- Grid_pfftFinalize, 100
- Grid_pfftInit, 100
- Grid_putBlkData, 32, 90
- Grid_sendOutputData, 91, 127
- Grid_solvePoisson, 108
- Grid_updateRefinement, 32, 33, 84, 91
- Grid_getBlkPhysicalSize, 85

IO
- IO_getPrevScalar, 128
- IO_getScalar, 127
- IO_init, 121, 141
- IO_output, 121, 123, 136
- IO_readUserArray, 128
- IO_setScalar, 127, 131
- IO_writeCheckpoint, 121, 136
- IO_writeIntegralQuantities, 125, 210
- IO_writeParticles, 124
- IO_writePlotfile, 123, 136
- IO_writeUserArray, 128

monitors
- Logfile
  - Logfile_init, 221
  - Logfile_stamp, 224
  - Logfile_stampMessage, 224
  - Logfile_writeSummary, 225

Timers
- Timers_create, 228
- Timers_getSummary, 228
- Timers_init, 228
- Timers_reset, 228
- Timers_start, 228
- Timers_stop, 228

Multispecies
- Multispecies_getAvg, 145
- Multispecies_getProperty, 145
- Multispecies_getSum, 145
- Multispecies_getSumFrac, 146
- Multispecies_getSumInv, 146
- Multispecies_getSumSqr, 146
- Multispecies_list, 146
- Multispecies_setProperty, 145
- Multispecies_unitTest, 147
Particles
- Particles_advance, 201, 203, 211
- Particles_computeDt, 203
- Particles_dump, 210
- Particles_init, 210
- Particles_initPositions, 209
- Particles_mapToMeshOneBlk, 209
- ParticlesSpecifyMethods, 57, 208
- Particles_unitTest, 210
- Particles_updateAttributes, 210

PhysicalConstants
- PhysicalConstants_get, 150
- PhysicalConstants_init, 150
- PhysicalConstants_list, 150
- PhysicalConstants_listUnits, 150
- PhysicalConstants_unitTest, 150, 151

physics
- Cosmology
  - Cosmology_cdmPowerSpectrum, 215
  - Cosmology_computeDeltaCrit, 216
  - Cosmology_computeDt, 215
  - Cosmology_computeVariance, 216
  - Cosmology_getRedshift, 214
  - Cosmology_massToLength, 216
  - Cosmology_redshiftToTime, 216
  - Cosmology_solveFriedmannEqn, 215
- Diffuse
  - Diffuse, 192, 193
  - Diffuse_species, 192
  - Diffuse_therm, 192, 298
  - Diffuse_visc, 192
- Eos, 174
  - Eos, 51, 90, 173, 178
  - Eos_getData, 51
  - Eos_init, 177
  - Eos_putData, 51
  - Eos_wrapped, 51, 52, 173, 174, 178
- Gravity
  - Gravity_init, 79
  - Gravity_potentialListOfBlocks, 198

RuntimeParameters
- RuntimeParameter_get, 141
- RuntimeParameters_get, 140, 141
- RuntimeParameters_getPrev, 127
- RuntimeParameters_mapStrToInt, 78, 79
- RuntimeParameters_set, 141

Simulation
- Simulation_defineDomain, 77, 80, 237, 250
- Simulation_getVarnameType, 77
- Simulation_init, 19
- Simulation_initBlock, 90
- Simulation_initSpecies, 144, 145, 177, 181
Index

.dump_checkpoint, see IO
.dump_restart, see IO
subunit, 34
  ptherimary, 34

adaptive mesh, 83
adiabatic indices, 174
angular coordinates, 77

block, 75
boundary conditions, 78

computational domain, 77
Config, 17, 18

double precision, 23

equation of state, 173, 174
  Eos, 178
  Eos_wrapped, 178
  gamma-law, 174, 175
  Helmholtz, 174, 175
  multi-gamma, 174, 175
  rhd-gamma, 174

fidlr
  xflash3, 12
FIXEDBLOCKSIZE mode, 60, 64, 82
Flash.h, 59
flash.par, 11, 25, 123, 222
flux conservation, 85

geometry, see mesh
gmake
  gmake clean, 10
  Makefile.h, 10
  parallel build, 10
gmake
  gmake realclean, 10
grid
  boundary, see boundary conditions
data averaging, 84
guardcells, see guardcells
interpolation, 82, 84, 86, 110, 115
  order, 86

prolongation, 86, 115
restriction, 86, 115
xmin/xmax, 77
ymin/ymax, 77
zmin/zmax, 77
Grid_getCellCoords, 23
guardcells, 75

HDF5, see IO

I/O, 118
IDL, 9

IO
  .dump_checkpoint, 121
  .dump_particle_file, 124
  .dump_plotfile, 123
  .dump_restart, 121
  checkpoint file, 121
  forced Plotfile, 123
  HDF5, 129
  output file names, 128
  particle file, 124
  plotfile, 123

kernel, 35

Makefile.h, 39
MAXBLOCKS, 41, 84
memory, 83, 84

mesh
  geometry, 110
    CARTESIAN, 111
    Cartesian, 113
    CYLINDRICAL, 111
cylindrical, 113
    POLAR, 111
    polar, 114
    SPHERICAL, 111
    spherical, 114
  refinement criteria, 21

NONFIXEDBLOCKSIZE mode, 60, 82, 83, 91

PARAMESH, 83
paramesh, 76
Physical Constants, 149
  CGS,CGS, 149
  MKS,MKS, 149

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

setup shortcuts, 129, 135
setup_params, 27

setups
  example, 17
  sedov, 10

shortcuts, 120
Simulation_data, 19
Simulation_init, 19
Simulation_initBlock, 17, 22
  creating new, 17

unit, 31

variable type
  conserved, 50
  mass-specific, 50
  variable type, 50

XFLASH3_DIR, 12