

# FLASH User's Guide

Version 4.8

May 2024 (last updated May 6, 2024)



Flash Center for Computational Science  
University of Rochester



# 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 developed in part by the DOE NNSA- and DOE Office of Science-supported Flash Center for Computational Science at the University of Chicago and the University of Rochester.”

The users should visit the bibliography hosted at [flash.rochester.edu/site/publications/flash\\_pubs.shtml](http://flash.rochester.edu/site/publications/flash_pubs.shtml) to find the relevant papers to cite in their work.

## 0.2 Full License Agreement

This document constitutes a binding license agreement (the "License Agreement") between the University of Rochester, through its Flash Center for Computational Science (hereafter, "we", "us" or the "Center") and users outside the Center ("users") regarding the dissemination of the "FLASH Code" by the Center and use of the FLASH Code by users. Users are responsible for following all applicable rules described below.

1. **Public Release.** Publicly released versions of the FLASH Code are available via the Center's website. We expect to include any external contributions to the Code in public releases that occur after the end of negotiated testing period with users.
2. **Decision Process.** Release of the FLASH Code to users not located at the University of Chicago or at the University of Rochester is governed solely by the Center's Director and Management Committee at the University of Rochester; decisions related to public release of the FLASH Code will be made in the same manner.
3. **License and Distribution Rights.** The University of Chicago owns the copyright to the FLASH Code developed by the members of the Flash Center. The University of Rochester serves as the exclusive operator of the Flash Center for Computational Science and holds a perpetual, worldwide, fully transferable, fully sublicensable and fully enforceable license of the FLASH Code. External contributors to the FLASH Code may choose to be included in the Copyright Assertion. Users of the FLASH Code pursuant to this License Agreement agree that the FLASH Code, or any part of the code, can only be released and distributed by the Flash Center; individual users of the FLASH Code are not free to re-distribute the FLASH Code, or any of its components, outside the Center. All users of the FLASH Code must accept the online version of this License Agreement at <https://flash.rochester.edu>. Distribution of the FLASH Code can only occur once the user accepts the License Agreement. For those users who accept this License Agreement and who are granted access to the FLASH Code, the Center grants to those users a non-exclusive, non-transferable, non-sublicensable right to use the FLASH Code, subject to the terms of this License Agreement.

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

- (a) The software must carry prominent notices stating that the user changed specified portions of the FLASH Code. This will also assist us in clearly identifying the portions of the FLASH Code that the user has contributed.

- (b) The software must display the following acknowledgement: “This product includes software developed by and/or derived from the Flash Center for Computational Science (<https://flash.rochester.edu>) to which the U.S. Government retains certain rights.”

- (c) The FLASH Code header section, which describes the origins of the FLASH Code and of its components, must remain intact, and should be included in all modified versions of the code.

Furthermore, all publications resulting from the use of the FLASH Code, or any modified version or portion of the FLASH Code, must acknowledge the Flash Center for Computational Science; addition of the following text to the paper acknowledgments will be sufficient:

“The software used in this work was developed in part by the DOE NNSA- and DOE Office of Science-supported Flash Center for Computational Science at the University of Chicago and the University of Rochester.”

The Code header provides information on software that has been utilized as part of the FLASH development effort. The Center website includes a list of key scientific journal references for the FLASH Code. We request that such references be included in the reference section of any papers based on the FLASH Code.

5. **Commercial Use.** All users interested in commercial use of the FLASH Code must obtain prior written approval from the Director of the Center. Use of the FLASH Code, or any modification thereof, for commercial purposes is not permitted otherwise.
6. **Bug Fixes and New Releases.** As part of the FLASH Code dissemination process, the Center has set up and will maintain as part of its website mechanisms for announcing new FLASH Code releases, collecting requests for FLASH Code use and collecting and disseminating relevant documentation. We support the user community through mailing lists and by providing a bug report facility.
7. **User Feedback.** The Center requests that all users of the FLASH Code notify the Center about all publications that incorporate results based on the use of the code, or modified versions of the code or its components. All such information can be sent to [infoflash.rochester.edu](mailto:infoflash.rochester.edu).
8. **No Assignment.** User may not assign or otherwise transfer any of its rights or obligations under this Agreement.
9. **Governing Law.** This Agreement will be interpreted, construed, and enforced in all respects in accordance with the local laws of the State of New York.
10. **Disclaimer.** The FLASH Code was prepared, in part, as an account of work sponsored by an agency of the United States Government.



THE FLASH CODE IS PROVIDED "AS IS" AND THE UNITED STATES, THE UNIVERSITY OF CHICAGO, THE UNIVERSITY OF ROCHESTER, THE CENTER, ANY CONTRIBUTORS TO THE FLASH CODE, OR ANY OF THEIR EMPLOYEES OR CONTRACTORS, MAKE NO WARRANTIES, EXPRESS OR IMPLIED (INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE), OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF THE FLASH CODE OR THAT THE FLASH CODE WILL BE FREE FROM DEFECTS.

IN NO EVENT WILL THE UNIVERSITY OF CHICAGO OR THE UNIVERSITY OF ROCHESTER OR THE CENTER OR ANY CONTRIBUTORS TO THE FLASH CODE OR ANY OF THEIR TRUSTEES, EMPLOYEES, FACULTY, AGENTS, OR OTHER REPRESENTATIVES BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THIS AGREEMENT OR THE USE OF THE SOFTWARE. TO THE FULLEST EXTENT PERMITTED BY LAW, UNDER NO CIRCUMSTANCES WILL THE UNIVERSITY CHICAGO OR THE UNIVERSITY ROCHESTER OR THE CENTER HAVE ANY LIABILITY OF ANY KIND (MONETARY OR OTHERWISE) ARISING OUT OF OR RELATED TO THIS AGREEMENT, REGARDLESS OF THE FORUM AND REGARDLESS OF WHETHER ANY ACTION OR CLAIM IS BASED ON CONTRACT, TORT, OR OTHERWISE.

## Acknowledgments

The Flash Center for Computational Science at the University of Rochester is supported by the U.S. DOE NNSA, the U.S. DOE Office of Science, the U.S. DOE ARPA-E, the NSF, the U.S. DOD, Los Alamos National Laboratory, Lawrence Livermore National Laboratory, and the Laboratory for Laser Energetics. Some of the test calculations described here were performed on machines at LLNL, LANL, San Diego Supercomputing Center, and ANL. The current contributors to the code from the Center include:

Abigail Armstrong, Tristan Bachmann, Jonathan Carroll-Nellenback, Pericles Farmakis, Milad Fatenejad, Edward Hansen, BaoWei Liu, Mary McMullan, David Michta, Kassie Moczulski, Ananya Mohapatra, Adam Reyes, Rich Sarkis, Joshua P. Sauppe, Anthony Scopatz, Victor Tranchant, Petros Tzeferacos, Vipul Vaidya, and Amy Yangyuxin Zou.

Considerable external and past contributors include:

Marissa Adams, Katie Antypas, John Bachan, Robi Banerjee, Alemayehu Bogale, Edward Brown, Peter Brune, Alvaro Caceres, Alan Calder, Alberto Cattaneo, Emmanouil Chatzopoulos, Sean Couch, Christopher Daley, Scott Davidson, Anshu Dubey, Jonathan Dursi, Lee Ellison, Christoph Federrath, Scott Feister, Robert Fisher, Norbert Flocke, Bruce Fryxell, J. Brad Gallagher, Murali Ganapathy Krishna, Fernando Garcia-Rubio Nathan Goldbaum, Shravan K. Gopal, William Gray, Carlo Graziani, Nathan Hearn, Mats Holmström, George Calhoun Jordan IV, Benjamin Khier, Jake King, JT Laune, Yousef Lawrence, Dongwook Lee, Mokin Lee, Timur Linde, Yingchao Lu, Zarija Lukic, Elisabeth Medina, Andrea Mignone, Joshua Miller, Prateeti Mohapatra, Samuel Olofin, Kevin Olson, Salvatore Orlando, Tomek Plewa, Kim Robinson, Lynn Reid, Paul Rich, Paul Ricker, Katherine Riley, Chalence Safranek-Shrader, Daniel Sheeler, Andrew Siegel, Noel Taylor, Frank Timmes, Dean Townsley, Niels Vanderloo, Marcos Vanella, Natalia Vladimirova, Christopher Walker, Klaus Weide, Greg Weirs, Richard Wunsch, Mike Zingale, and John ZuHone.

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

[https://web.archive.org/web/20160421055017/http://www.physics.drexel.edu/~olson/paramesh-doc/Users\\_manual/amr.html](https://web.archive.org/web/20160421055017/http://www.physics.drexel.edu/~olson/paramesh-doc/Users_manual/amr.html).

# Contents

<b>License</b>	<b>i</b>
0.1 Acknowledgments in Publication	i
0.2 Full License Agreement	i
<b>1 Introduction</b>	<b>1</b>
1.1 What's New in FLASH4	2
1.2 External Contributions	11
1.3 Known Issues in This Release	11
1.4 About the User's Guide	13
<b>I Getting Started</b>	<b>15</b>
<b>2 Quick Start</b>	<b>17</b>
2.1 System requirements	17
2.2 Unpacking and configuring FLASH for quick start	18
2.3 Alternative Configuration With a Docker Image	20
2.4 Running FLASH	21
<b>3 Setting Up New Problems</b>	<b>27</b>
3.1 Creating a Config file	28
3.2 Creating a Makefile	28
3.3 Creating a Simulation_data.F90	29
3.4 Creating a Simulation_init.F90	29
3.5 Creating a Simulation_initBlock.F90	31
3.6 Creating a Simulation_freeUserArrays.F90	35
3.7 The runtime parameter file (flash.par)	36
3.8 Running your simulation	39
<b>II The FLASH Software System</b>	<b>41</b>
<b>4 Overview of FLASH architecture</b>	<b>43</b>
4.1 FLASH Inheritance	44
4.2 Unit Architecture	44
4.2.1 Stub Implementations	45
4.2.2 Subunits	46
4.2.3 Unit Data Modules, <code>_init</code> , and <code>_finalize</code> routines	46
4.2.4 Private Routines: kernels and helpers	47
4.3 Unit Test Framework	48

<b>5</b>	<b>The FLASH configuration script (setup)</b>	<b>49</b>
5.1	Setup Arguments	50
5.2	Comprehensive List of Setup Arguments	51
5.3	Using Shortcuts	65
5.4	Setup Variables and Preprocessing Config Files	66
5.5	Config Files	68
5.5.1	Configuration file syntax	68
5.5.2	Configuration directives	69
5.6	Creating a Site-specific Makefile	78
5.7	Files Created During the setup Process	79
5.7.1	Informational files	79
5.7.2	Code generated by the setup call	79
5.7.3	Makefiles generated by setup	80
5.8	Setup a hybrid MPI+OpenMP FLASH application	81
5.9	Setup a FLASH+Chombo application	83
5.9.1	Overview	83
5.9.2	Build procedure	83
5.10	Setup a FLASH application with cmake	86
<b>6</b>	<b>The Flash.h file</b>	<b>89</b>
6.1	UNK, FACE(XYZ) Dimensions	89
6.2	Property Variables, Species and Mass Scalars	90
6.3	Fluxes	91
6.4	Scratch Vars	91
6.5	Fluid Variables Example	92
6.6	Particles	93
6.6.1	Particles Types	93
6.6.2	Particles Properties	93
6.7	Non-Replicated Variable Arrays	94
6.7.1	Per-Array Macros	94
6.7.2	Array Partitioning Macros	94
6.7.3	Example	95
6.8	Other Preprocessor Symbols	96
<b>III</b>	<b>Driver Unit</b>	<b>97</b>
<b>7</b>	<b>Driver Unit</b>	<b>99</b>
7.1	Driver Routines	99
7.1.1	Driver_initFlash	100
7.1.2	Driver_evolveFlash	100
7.1.3	Driver_finalizeFlash	102
7.1.4	Driver accessor functions	102
7.1.5	Time Step Limiting	103
<b>IV</b>	<b>Infrastructure Units</b>	<b>105</b>
<b>8</b>	<b>Grid Unit</b>	<b>107</b>
8.1	Overview	109
8.2	GridMain Data Structures	111
8.3	Computational Domain	112
8.4	Boundary Conditions	113
8.4.1	Boundary Condition Types	113
8.4.2	Boundary Conditions at Obstacles	114

8.4.3	Implementing Boundary Conditions	115
8.5	Uniform Grid	117
8.5.1	FIXEDBLOCKSIZE Mode	117
8.5.2	NONFIXEDBLOCKSIZE mode	118
8.6	Adaptive Mesh Refinement (AMR) Grid with Paramesh	118
8.6.1	Additional Data Structures	120
8.6.2	Grid Interpolation	121
8.6.3	Refinement	122
8.7	Chombo	125
8.7.1	Using Chombo in a UG configuration	125
8.7.2	Using Chombo in an AMR configuration	125
8.8	GridMain Usage	127
8.9	GridParticles	128
8.9.1	GridParticlesMove	129
8.9.2	GridParticlesMapToMesh	131
8.10	GridSolvers	135
8.10.1	Pfft	135
8.10.2	Poisson equation	138
8.10.3	Using the Poisson solvers	151
8.10.4	HYPRE	156
8.11	Grid Geometry	162
8.11.1	Understanding 1D, 2D, Curvilinear	162
8.11.2	Choosing a Geometry	164
8.11.3	Geometry Information in Code	164
8.11.4	Available Geometries	165
8.11.5	Conservative Prolongation/Restriction on Non-Cartesian Grids	168
8.12	Unit Test	168
<b>9</b>	<b>IO Unit</b>	<b>171</b>
9.1	IO Implementations	173
9.2	Output Files	175
9.2.1	Checkpoint files - Restarting a Simulation	175
9.2.2	Plotfiles	177
9.2.3	Particle files	179
9.2.4	Integrated Grid Quantities – flash.dat	180
9.2.5	General Runtime Parameters	180
9.3	Restarts and Runtime Parameters	181
9.4	Output Scalars	181
9.5	Output User-defined Arrays	181
9.6	Output Scratch Variables	182
9.7	Face-Centered Data	182
9.8	Output Filenames	182
9.9	Output Formats	183
9.9.1	HDF5	183
9.9.2	Parallel-NetCDF	189
9.9.3	Direct IO	189
9.9.4	Output Side Effects	189
9.10	Working with Output Files	190
9.11	Unit Test	190
9.12	Chombo	191
9.13	Derived data type I/O	191

<b>10 Runtime Parameters Unit</b>	<b>193</b>
10.1 Defining Runtime Parameters	193
10.2 Identifying Valid Runtime Parameters	193
10.3 Routine Descriptions	194
10.4 Example Usage	195
<b>11 Multispecies Unit</b>	<b>197</b>
11.1 Defining Species	197
11.2 Initializing Species Information in <code>Simulation_initSpecies</code>	198
11.3 Specifying Constituent Elements of a Species	200
11.4 Alternative Method for Defining Species	200
11.5 Routine Descriptions	201
11.6 Example Usage	203
11.7 Unit Test	203
<b>12 Physical Constants Unit</b>	<b>205</b>
12.1 Available Constants and Units	206
12.2 Applicable Runtime Parameters	206
12.3 Routine Descriptions	206
12.4 Unit Test	207
<b>13 Cpp &amp; Python units</b>	<b>209</b>
13.1 Cpp	209
13.2 Python	210
13.2.1 Setup Generated Files	210
<b>V Physics Units</b>	<b>211</b>
<b>14 3T Capabilities for Simulation of HEDP Experiments</b>	<b>213</b>
<b>15 Hydrodynamics Units</b>	<b>217</b>
15.1 Gas hydrodynamics	219
15.1.1 Usage	219
15.1.2 The piecewise-parabolic method (PPM)	219
15.1.3 The unsplit hydro solver	220
15.1.4 The Volume-of-Fluid Solver	224
15.1.5 Multitemperature extension for Hydro	225
15.1.6 Chombo compatible Hydro	228
15.2 Relativistic hydrodynamics (RHD)	228
15.2.1 Overview	228
15.2.2 Equations	229
15.2.3 Relativistic Equation of State	229
15.2.4 Additional Runtime Parameter	230
15.3 Magnetohydrodynamics (MHD)	230
15.3.1 Description	230
15.3.2 Usage	231
15.3.3 Algorithm: The Unsplit Staggered Mesh Solver	232
15.3.4 Algorithm: The Eight-wave Solver	237
15.3.5 Extended MHD	238
15.4 Radiation-Flux-Limiter Aware Hydrodynamics	245
<b>16 Incompressible Navier-Stokes Unit</b>	<b>247</b>

<b>17 Equation of State Unit</b>	<b>249</b>
17.1 Introduction	249
17.2 Gamma Law and Multigamma	250
17.2.1 Ideal Gamma Law for Relativistic Hydrodynamics	251
17.3 Helmholtz	251
17.4 Multitemperature extension for Eos	253
17.4.1 Gamma	253
17.4.2 Multigamma	254
17.4.3 Tabulated	255
17.4.4 Multitype	258
17.5 MultiFluid	259
17.6 Usage	259
17.6.1 Initialization	259
17.6.2 Runtime Parameters	259
17.6.3 Direct and Wrapped Calls	260
17.7 Unit Test	261
<b>18 Local Source Terms</b>	<b>263</b>
18.1 Burn Unit	264
18.1.1 Algorithms	264
18.1.2 Reaction networks	265
18.1.3 Detecting shocks	269
18.1.4 Energy generation rates and reaction rates	269
18.1.5 Temperature-based timestep limiting	270
18.2 Ionization Unit	270
18.2.1 Algorithms	271
18.2.2 Usage	271
18.3 Stir Unit	272
18.3.1 Stir Unit: Generate Implementation	272
18.3.2 Stir Unit: FromFile Implementation	272
18.3.3 Using the StirFromFile Unit	274
18.3.4 Stirring Unit Test	276
18.4 Energy Deposition Unit	276
18.4.1 Ray Tracing in the Geometric Optics Limit	276
18.4.2 Laser Power Deposition	277
18.4.3 Laser Energy Density	279
18.4.4 Algorithmic Implementations of the Ray Tracing	279
18.4.5 Setting up the Laser Pulse	283
18.4.6 Setting up the Laser Beam	283
18.4.7 Setting up the Rays	286
18.4.8 3D Laser Ray Tracing in 2D Cylindrical Symmetry	291
18.4.9 Ray Tracing in 3D Cylindrical Domains	297
18.4.10 Synchronous and Asynchronous Ray Tracing	302
18.4.11 Usage	306
18.4.12 Unit Tests for 3D/2D Cartesian Domain Geometries	311
18.4.13 Unit Tests for 3D Cylindrical Domain Geometries	318
18.5 Heatexchange	320
18.5.1 Spitzer Heat Exchange	321
18.5.2 LeeMore Heat Exchange	323
18.6 Flame	324
18.6.1 Reaction-Diffusion Forms	324
18.6.2 Unit Structure	325
18.7 Turbulence Measurement	326
18.8 Circuit	326

18.8.1	Constant	326
18.8.2	FileInput	326
18.8.3	McBride	326
18.8.4	CESZAR	327
<b>19</b>	<b>Diffusive Terms</b>	<b>329</b>
19.1	Diffuse Unit	329
19.1.1	Diffuse Flux-Based implementations	330
19.1.2	General Implicit Diffusion Solver	331
19.1.3	Flux Limiters	334
19.1.4	Isotropic Thermal Diffusion	335
19.1.5	Anisotropic Thermal Diffusion	336
19.1.6	Magnetic Diffusion	337
<b>20</b>	<b>Gravity Unit</b>	<b>341</b>
20.1	Introduction	341
20.2	Externally Applied Fields	342
20.2.1	Constant Gravitational Field	342
20.2.2	Plane-parallel Gravitational field	342
20.2.3	Gravitational Field of a Point Mass	342
20.2.4	User-Defined Gravitational Field	342
20.3	Self-gravity	342
20.3.1	Coupling Gravity with Hydrodynamics	343
20.3.2	Tree Gravity	344
20.4	Usage	345
20.4.1	Tree Gravity Unit Usage	345
20.5	Unit Tests	347
<b>21</b>	<b>Particles Unit</b>	<b>349</b>
21.1	Time Integration	351
21.1.1	Active Particles (Massive)	351
21.1.2	Charged Particles - Hybrid PIC	353
21.1.3	Passive Particles	355
21.2	Mesh/Particle Mapping	359
21.2.1	Quadratic Mesh Mapping	360
21.2.2	Cloud in Cell Mapping	362
21.3	Using the Particles Unit	362
21.3.1	Particles Runtime Parameters	364
21.3.2	Particle Attributes	364
21.3.3	Particle I/O	365
21.3.4	Unit Tests	365
21.4	Sink Particles	366
21.4.1	Basics of Sink Particles	366
21.4.2	Using the Sink Particle Unit	366
21.4.3	The Sink Particle Method	367
21.4.4	Sink Particle Unit Test	368
<b>22</b>	<b>Cosmology Unit</b>	<b>371</b>
22.1	Algorithms and Equations	371
22.2	Using the Cosmology unit	373
22.3	Unit Test	374



<b>23 Material Properties Units</b>	<b>375</b>
23.1 Thermal Conductivity . . . . .	376
23.1.1 Anisotropic Thermal Conductivity . . . . .	377
23.2 Magnetic Resistivity . . . . .	377
23.2.1 Constant resistivity . . . . .	378
23.2.2 Spitzer HighZ resistivity . . . . .	378
23.2.3 DaviesWen resistivity . . . . .	378
23.2.4 Vacuum resistivity . . . . .	378
23.3 Viscosity . . . . .	379
23.3.1 Constant Viscosity . . . . .	379
23.3.2 Spitzer Viscosity . . . . .	379
23.4 Thermoelectric Coefficients . . . . .	379
23.4.1 Constant thermoelectric coefficients . . . . .	380
23.4.2 DaviesWen thermoelectric coefficients . . . . .	380
23.5 Opacity . . . . .	380
23.5.1 Constant Implementation . . . . .	380
23.5.2 Constcm2g Implementation . . . . .	381
23.5.3 BremsstrahlungAndThomson Implementation . . . . .	381
23.5.4 OPAL Implementation . . . . .	381
23.5.5 Multispecies Implementation . . . . .	381
23.5.6 The IONMIX EOS/Opacity Format . . . . .	384
23.6 Mass Diffusivity . . . . .	386
<b>24 Physics Utilities</b>	<b>387</b>
24.1 PlasmaState . . . . .	387
<b>25 Radiative Transfer Unit</b>	<b>389</b>
25.1 Multigroup Diffusion . . . . .	390
25.1.1 Using Multigroup Radiation Diffusion . . . . .	390
25.1.2 Using Mesh Replication with MGD . . . . .	391
25.1.3 Specifying Initial Conditions . . . . .	392
25.1.4 Altering the Radiation Spectrum . . . . .	394
<b>VI Monitor Units</b>	<b>397</b>
<b>26 Logfile Unit</b>	<b>399</b>
26.1 Meta Data . . . . .	400
26.2 Runtime Parameters, Physical Constants, and Multispecies Data . . . . .	400
26.3 Accessor Functions and Timestep Data . . . . .	402
26.4 Performance Data . . . . .	403
26.5 Example Usage . . . . .	404
<b>27 Timer and Profiler Units</b>	<b>405</b>
27.1 Timers . . . . .	405
27.1.1 MPINative . . . . .	405
27.1.2 Tau . . . . .	406
27.2 Profiler . . . . .	407
<b>VII Diagnostic Units</b>	<b>409</b>
<b>28 Proton Imaging Unit</b>	<b>411</b>
28.0.1 Proton Deflection by Lorentz Force . . . . .	412
28.0.2 Setting up the Proton Beam . . . . .	415

28.0.3	Creating the Protons	415
28.0.4	Setting up the Detector Screens	417
28.0.5	Time Resolved Proton Imaging	418
28.0.6	Usage	418
28.0.7	Unit Test	421
<b>29</b>	<b>Proton Emission Unit</b>	<b>429</b>
29.0.1	Proton Generation	429
29.0.2	Proton Detector Screens	431
29.0.3	Proton Emission Boxes	431
29.0.4	Usage	432
<b>30</b>	<b>Thomson Scattering Unit</b>	<b>435</b>
30.1	Thomson Scattering including Ray Tracing and Deflection	435
30.1.1	Identifying the Interaction Region	436
30.1.2	Assembling the Thomson Scattering Spectra	436
30.1.3	Usage	438
30.2	Simple Thomson Scattering without Ray Tracing	443
30.2.1	Ray Tracing in the Geometric Optics Limit	443
30.2.2	Laser Power Deposition	443
30.2.3	Laser Energy Density	443
30.2.4	Algorithmic Implementations of the Ray Tracing	444
30.2.5	Setting up the Probe Laser Pulse	444
30.2.6	Setting up the Laser Beam	444
30.2.7	Creating the Rays	444
30.2.8	Setting up the Detectors	445
30.2.9	Usage	446
30.2.10	Example Setup	447
<b>31</b>	<b>X-ray Imaging Unit</b>	<b>449</b>
31.0.1	X-ray Paths through Domain	449
31.0.2	Creating and Tracing the X-rays	450
31.0.3	Setting up and Recording on the X-ray Detector Screens	450
31.0.4	Usage	451
31.0.5	Unit Test	453
<b>VIII</b>	<b>Numerical Tools Units</b>	<b>455</b>
<b>32</b>	<b>Interpolate Unit</b>	<b>457</b>
32.1	Introduction	457
32.2	Piecewise Cubic Interpolation	457
32.3	Usage	459
<b>33</b>	<b>Roots Unit</b>	<b>461</b>
33.1	Introduction	461
33.2	Roots of Quadratic Polynomials	461
33.3	Roots of Cubic Polynomials	462
33.4	Roots of Quartic Polynomials	462
33.5	Usage	462
33.6	Unit Tests	462
33.6.1	Quadratic Polynomials Root Test	463
33.6.2	Cubic Polynomials Root Test	464
33.6.3	Quartic Polynomials Root Test	465

<b>34 RungeKutta Unit</b>	<b>467</b>
34.1 Introduction	467
34.2 Runge Kutta Integration	467
34.3 Usage	470
34.4 Unit Tests	471
34.4.1 Runge Kutta FLASH Test for a 2D Elliptical Path	471
34.4.2 Runge Kutta FLASH Test for Binomial Homogeneous Higher Order ODE	476
 <b>IX Simulation Units</b>	 <b>477</b>
<b>35 The Supplied Test Problems</b>	<b>479</b>
35.1 Hydrodynamics Test Problems	479
35.1.1 Sod Shock-Tube	480
35.1.2 Variants of the Sod Problem in Curvilinear Geometries	483
35.1.3 Interacting Blast-Wave <b>Blast2</b>	484
35.1.4 Sedov Explosion	488
35.1.5 Isentropic Vortex	493
35.1.6 The double Mach reflection problem	494
35.1.7 Wind Tunnel With a Step	498
35.1.8 The Shu-Osher problem	501
35.1.9 Driven Turbulence <b>StirTurb</b>	504
35.1.10 Relativistic Sod Shock-Tube	504
35.1.11 Relativistic Two-dimensional Riemann	511
35.1.12 Flow Interactions with Stationary Rigid Body	512
35.2 Magnetohydrodynamics Test Problems	514
35.2.1 Brio-Wu MHD Shock Tube	514
35.2.2 Orszag-Tang MHD Vortex	516
35.2.3 Magnetized Accretion Torus	523
35.2.4 Magnetized Noh Z-pinch	524
35.2.5 MHD Rotor	524
35.2.6 MHD Current Sheet	525
35.2.7 Field Loop	530
35.2.8 3D MHD Blast	530
35.3 Gravity Test Problems	532
35.3.1 Jeans Instability	532
35.3.2 Homologous Dust Collapse	536
35.3.3 Huang-Greengard Poisson Test	538
35.3.4 MacLaurin	539
35.4 Particles Test Problems	544
35.4.1 Two-particle <b>Orbit</b>	544
35.4.2 Zel'dovich Pancake	545
35.4.3 Modified Huang-Greengard Poisson Test	548
35.5 Burn Test Problem	548
35.5.1 Cellular Nuclear Burning	548
35.6 RadTrans Test Problems	552
35.6.1 Infinite Medium, Energy Equilibration	552
35.6.2 Radiation Step Test	552
35.7 Other Test Problems	554
35.7.1 The non-equilibrium ionization test problem	554
35.7.2 The Delta-Function Heat Conduction Problem	559
35.7.3 The HydroStatic Test Problem	559
35.7.4 Hybrid-PIC Test Problems	559
35.7.5 Full-physics Laser Driven Simulation	561

35.7.6 Laser Driven Simulation with Thomson Scattering Diagnostics . . . . .	570
35.7.7 Z-pinch Simulation . . . . .	570
35.8 3T Shock Simulations . . . . .	573
35.8.1 Shafranov Shock . . . . .	573
35.8.2 Non-Equilibrium Radiative Shock . . . . .	576
35.8.3 Blast Wave with Thermal Conduction . . . . .	576
35.9 Matter+Radiation Simulations . . . . .	579
35.9.1 Radiation-Inhibited Bondi Accretion . . . . .	579
35.9.2 Radiation Blast Wave . . . . .	579
35.9.3 1D Supernova . . . . .	579
 <b>X Tools</b>	 <b>581</b>
<b>36 VisIt</b>	<b>585</b>
<b>37 Serial FLASH Output Comparison Utility (sfocu)</b>	<b>587</b>
37.1 Building sfocu . . . . .	587
37.2 Using sfocu . . . . .	587
<b>38 Drift</b>	<b>589</b>
38.1 Introduction . . . . .	589
38.2 Enabling drift . . . . .	590
38.3 Typical workflow . . . . .	590
38.4 Caveats and Annoyances . . . . .	591
<b>39 FLASH IDL Routines (fidlr3.0)</b>	<b>593</b>
39.1 Installing and Running fidlr3.0 . . . . .	593
39.1.1 Setting Up fidlr3.0Environment Variables . . . . .	593
39.1.2 Running IDL . . . . .	594
39.2 xflash3: A Widget Interface to Plotting FLASH Datasets . . . . .	594
39.2.1 File Menu . . . . .	594
39.2.2 Defaults Menu . . . . .	595
39.2.3 Colormap Menu . . . . .	596
39.2.4 X/Y plot count Menu . . . . .	596
39.2.5 Plotting options available from the GUI . . . . .	596
39.2.6 Plotting buttons . . . . .	598
39.3 Comparing two datasets . . . . .	600
<b>40 convertspec3d</b>	<b>603</b>
40.1 Installation . . . . .	603
40.2 Usage . . . . .	604
 <b>XI Going Further with FLASH</b>	 <b>605</b>
<b>41 Adding new solvers</b>	<b>609</b>
<b>42 Porting FLASH to other machines</b>	<b>611</b>
42.1 Writing a Makefile.h . . . . .	611

<b>43 Multithreaded FLASH</b>	<b>615</b>
43.1 Overview	615
43.2 Threading strategies	615
43.3 Running multithreaded FLASH	616
43.3.1 OpenMP variables	616
43.3.2 FLASH variables	616
43.3.3 FLASH constants	617
43.4 Verifying correctness	617
43.5 Performance results	617
43.5.1 Multipole solver	617
43.5.2 Helmholtz EOS	618
43.5.3 Sedov	619
43.5.4 LaserSlab	620
43.6 Conclusion	621
<b>References</b>	<b>625</b>
<b>Runtime Parameters</b>	<b>633</b>
<b>API Index</b>	<b>639</b>
<b>Index</b>	<b>641</b>



# Chapter 1

## Introduction

The FLASH code is a modular, parallel multiphysics simulation code capable of handling general compressible flow problems found in many plasma physics experiments and astrophysical environments. It is a set of independent code units put together with a Python language setup tool to form various applications. The code is written in FORTRAN90 and C. It uses the Message-Passing Interface (MPI) library for inter-processor communication and the HDF5 or Parallel-NetCDF library for parallel I/O to achieve portability and scalability on a variety of different parallel computers. FLASH4 has three interchangeable discretization grids: a Uniform Grid, and a block-structured oct-tree based adaptive grid using the **PARAMESH** library, and a block-structured patch based adaptive grid using **Chombo**. Both **PARAMESH** and **Chombo** place resolution elements only where they are needed most.<sup>1</sup> 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.

For the past 20 years, the Flash Center for Computational Science has led the development of the FLASH code under the directorship (1997-2001) of Robert Rosner, William E. Wrather Distinguished Service Professor of the Department of Astronomy and Astrophysics, the Enrico Fermi Institute, and the Harris School of Public Policy at the University; under the directorship (2002-2018) of Donald Q. Lamb, Robert A. Millikan Distinguished Service Professor Emeritus of the Department of Astronomy and Astrophysics, the Enrico Fermi Institute, and the Harris School of Public Policy at the University; and under the directorship (2018-present) of Petros Tzeferacos, Associate Professor at the Department of Physics and Astronomy and Senior Scientist at the Laboratory for Laser Energetics (LLE) at the University of Rochester.

The development of the FLASH Code has been primarily funded by the U.S. DOE NNSA, initially (FLASH Code versions 2 and 3, 1998-2010) through the Academic Partnership Academic Strategic Alliances Program (ASAP) of the NNSA Accelerated Strategic Computing Initiative (ASCI) Program. The scientific goal of the Center then was to address several problems related to thermonuclear flashes on the surface of compact stars (neutron stars and white dwarfs), in particular Type Ia supernovae, and novae.

In 2010, NNSA mandated the Flash Center director to make the FLASH Code into a publicly available academic code for laser-driven High Energy Density Physics (HEDP) and to support the academic HEDP community, in particular the academic research teams accessing the Omega Laser Facility at LLE and the National Ignition Facility at Lawrence Livermore National Laboratory (LLNL). FLASH Code version 4 (2011-present) was then developed by the Flash Center, in collaboration with Los Alamos National Laboratory (LANL, beginning 2010), LLNL (beginning 2010), the Naval Research Laboratory (NRL, beginning 2010), and LLE (beginning 2012). The continued expansion of FLASH's physics and algorithmic capabilities by the Flash Center is now supported primarily by the U.S. DOE NNSA, LANL, LLNL, LLE, U.S. DOE Advanced Research Projects Agency-Energy (ARPA-E), as well as the U.S. DOE Office of Science Fusion Energy Sciences (FES), the National Science Foundation (NSF), and the Department of Defense (DOD). This expansion is in turn extending the code's applicability into different plasma regimes relevant to plasma astrophysics, HEDP, and fusion energy.

The FLASH code has become a key hydrodynamics application used to test and debug new machine

---

<sup>1</sup>The **Chombo** grid in FLASH has had limited testing, and supports only a limited set of physics units. At this time the use of Chombo within FLASH for production is not recommended.

architectures because of its modular structure, portability, scalability and dependence on parallel I/O libraries. It has a growing user base and has rapidly become a shared code for the astrophysics community and beyond, with hundreds of active users who customize the code for their own research.

As of October 25, 2021, the Flash Center for Computational Science is part of at the Department of Physics and Astronomy at the University of Rochester.

## 1.1 What’s New in FLASH4

This Guide describes the release version 4.8 of FLASH4. FLASH4 includes all the well tested capabilities of FLASH3. There were a few modules in the official releases of FLASH2 which were added and tested by local users, but did not have standardized setups that could be used to test them after the migration to FLASH3. Those modules are not included in the official releases of FLASH3 or FLASH4, however, they are being made available to download ”as is” from the Flash Center’s website. We have ensured that they have been imported into FLASH4 to the extent that they conform to the architecture and compile. We cannot guarantee that they work correctly; they are meant to be useful starting points for users who need their functionality. We also welcome setups contributed by the users that can meaningfully test these units. If such setups become available to us, the units will be released in future.

In terms of the code architecture, FLASH4 closely follows FLASH3. The major changes from FLASH3 are several new capabilities in both physics solvers and infrastructure. Major effort went into the design of the FLASH3 architecture to ensure that the code can be easily modified and extended by internal as well as external developers. Each code unit in FLASH4, like in FLASH3 has a well defined interface and follows the rules for inheritance and encapsulation defined in FLASH3. One of the largest achievements of FLASH3 was the separation of the discretized ‘grid’ architecture from the actual physics. This untangling required changes in the deepest levels of the code, but has demonstrated its worth by allowing us to import a new AMR package **Chombo** into the code.

Because of the increasing importance of software verification and validation, the Flash code group has developed a test-suite application for FLASH3. The application is called FlashTest and can be used to setup, compile, execute, and test a series of FLASH code simulations on a regular basis. FlashTest is available without a license and can be downloaded from the [Code Support Web Page](#). There is also a more general open-source version of FlashTest which can be used to test any software in which an application is configured and then executed under a variety of different conditions. The results of the tests can then be visualized in a browser with FlashTestView, a companion to FlashTest that is also open-source.

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

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

- 3T capabilities in the split and unsplit Hydro solvers. There is support for non-cartesian geometry and the unsplit solver also supports stationary rigid body.
- Upwind biased constrained transport (CT) scheme in the unsplit staggered mesh MHD solver
- Full corner transport upwind (CTU) algorithm in the unsplit hydro/MHD solver
- Cylindrical geometry support in the unsplit staggered mesh MHD solver on UG and AMR. A couple of MHD simulation setups using cylindrical geometry.
- Units for radiation diffusion, conduction, and heat exchange.
- Equation-of state unit includes table based multi-material multi-temperature implementation.
- The Opacities unit with the ability to use hot and cold opacities.



- The laser drive with threading for performance
- Ability to replicate mesh for multigroup diffusion or other similar applications.
- Several important solvers have been threaded at both coarse-grain (one block per thread) and fine-grain (threads within a block) levels.
- Several new HEDP simulation setups.
- A new multipole solver
- Ability to add particles during evolution

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

- The HLLD Riemann solver has been improved to handle MHD degeneracy.
- PARAMESH's handling for face-centered variables in order to ensure divergence-free magnetic fields evolution on AMR now uses `gr_pmrpDivergenceFree=.true.` and `gr_pmrpForceConsistency=.true.` by default.
- The HEDP capabilities of the code have been exercised and are therefore more robust.
- Laser 3D in 2D ray tracing has been added. The code traces rays in a real 3D cylindrical domain using a computational 2D cylindrical domain and is based on a polygon approximation to the angular part.
- In non-fixedblocksize mode, restart with particles did not work when starting with a different processor count. This bug has now been fixed.
- All I/O implementations now support reading/writing 0 blocks and 0 particles.
- There is support for particles and face variables in PnetCDF
- Initialization of the computation domain has been optimized by eliminating unnecessary invocations of PARAMESH's "digital orrery" algorithm at simulation startup. It is possible to run the orrery in a reduced communicator in order to speed up FLASH initialization.
- The custom region code and corresponding Grid API routines have been removed.
- PARAMESH4DEV is now the default PARAMESH implementation.

The new capabilities in FLASH4.2 ... FLASH4.2.2 since FLASH4.0.1 include:

- New Core-Collapse Super Nova (CCSN) physics, with complete nuclear EOS routines, local neutrino heating/cooling and multispecies neutrino leakage.
- New unsplit Hydro and MHD implementations, highly optimized for performance. These implementations are now the default option. We have retained the old implementations as an `unsplit_old` alternative for compatibility reasons.
- New support for 3T magnetohydrodynamics, designed for HEDP problems.
- A new magnetic resistivity implementation, **SpitzerHighZ**, for HEDP problems. We have also extended the support for resistivity in cylindrical geometry in the unsplit solver.
- New threading capabilities for unsplit MHD, compatible with all threading strategies followed by the code.
- New, improved multipole Poisson solver, implementing the algorithmic refinements described in <http://dx.doi.org/10.1088/0004-637X/778/2/181> and <http://arxiv.org/abs/1307.3135>.
- Reorganization of the **EnergyDeposition** unit. A new feature has been included that allows Energy-Deposition to be called once every  $n$  time steps.

The new capabilities in FLASH4.3 since FLASH4.2.2 include:

- The sink particles implementation now has support for particles to remain active when leaving the grid domain (in case of outflow boundary conditions).
- New Proton Imaging unit: The new unit is a simulated diagnostic of the Proton Radiography used in HEDP experiments.
- Flux-limited-diffusion for radiation (implemented in RadTransMain/MGD) is now available for astrophysical problem setups:
  - MatRad3 (matter+rad [2T] stored in three components) implementations for several Eos types: **Gamma**, **Multigamma**, and (experimentally) **Helmholtz/SpeciesBased**.
  - Implemented additional terms in FLD Rad-Hydro equations to handle streaming and transition-to-streaming regimes better - including radiation pressure. This is currently available as a variant of the unsplit Hydro solver code, under HydroMain/unsplit\_rad . We call this RADFLAH - Radiation Flux-Limiter Aware Hydro. Setup with shortcut **+uhd3tR** instead of **+uhd3t** . This has had limited testing, mostly in 1D spherical geometry.
  - New test setups under **Simulation/SimulationMain/radflaHD**: **BondiAccretion**, **RadBlastWave**
  - Various fixes in Eos implementations.
  - New "outstream" diffusion solver boundary condition for streaming limit. (currently 1D spherical only)
  - Added Levermore-Pomraning flux limiter.
  - More flexible setup combinations are now easily possible - can combine, *e.g.*, species declared on setup command line with SPECIES in Config files and initialized with Simulation\_initSpecies, by setup with **ManualSpeciesDirectives=True**.
  - Created an "Immediate" HeatExchange implementation.
  - EXPERIMENTAL: ExpRelax variant of RadTrans diffusion solver, implements the algorithm described in Gittings et al (2008) for the RAGE code, good for handling strong matter-radiation coupling; for one group (grey) only.
  - EXPERIMENTAL: Unified variant of RadTrans diffusion solver, for handling several coupled scalar equations with HYPRE.
  - EXPERIMENTAL: More accurate implementation of flux limiting (and evaluation of diffusion coeffs): apply limiter to face values, not cell centered values.
- Gravity can now be used in 3T simulations.
- Laser Energy Deposition: New ray tracing options added based on cubic interpolation techniques. Two variants: 1) Piecewise Parabolic Ray Tracing (PPRT) and 2) Runge Kutta (RK) ray tracing.
- Introduction of new numerical tool units: 1) Interpolate: currently contains the routines to set up and perform cubic interpolations on rectangular 1D,2D,3D grids, 2) Roots: (will) contain all routines that solve  $f(x) = 0$  (currently contains quadratic, cubic and quartic polynomial root solvers, 3) Runge Kutta: sets up and performs Runge Kutta integration of arbitrary functions (passed as arguments).
- Unsplit Hydro/MHD: Local CFL factor using CFL\_VAR. (Declare a "VARIABLE cfl" and initialize it appropriately.)
- Unsplit Hydro/MHD: Significant reorganization.
  - reorganized definition and use of scratch data. Memory savings.
  - use **hy\_memAllocScratch** and friends.
  - **hy\_fullRiemannStateArrays** (instead of FLASH\_UHD\_NEED\_SCRATCHVARS)

- New runtime parameter `hy_fullSpecMsFluxHandling`, default `TRUE`. resulting in flux-corrected handling for species and mass scalars, including USM.
- Use `shockLowerCFL` instead of `shockDetect` runtime parameter.
- Revived `EOSforRiemann` option.
- More accurate handling of geometric effects close to the origin in 1D spherical geometry.

Important changes in FLASH4.4 since FLASH4.3 include:

- The default Hydro implementation has changed from split PPM to unsplit Hydro. A new shortcut `+splitHydro` can be used to request a split Hydro implementation.
- Updated values of many physical constants to 2014 CODATA values. This may cause differences from previously obtained results. The previous values of constants provided by the `PhysicalConstants` unit can be restored by replacing the file `PhysicalConstants/init.F90` with an older version; the version from FLASH4.3 is included as `PhysicalConstants_init.F90.flash43`. This should only be done to reproduce previous simulation results to bit accuracy.
- An improved Newton-Raphson search in the 3T Multi-type Eos implementation (MTMMMT, including Eos based on IONMIX tables) can prevent some cases of convergence failure by bounding the search. This implementation follows original improvements made to the `Helmholtz Eos` implementation by Dean Townsley.
- Added new Poisson solvers (Martin-Cartwright Geometric Multigrid and BiPCGStab, which uses multigrid as preconditioner). Combinations of homogeneous Dirichlet, Neumann, and periodic boundary conditions are supported (although not yet “isolated” boundaries for self-gravity).
- Added the `IncompNS` physics unit, which provides a solver for incompressible flow problems on rectangular domains. Multistep and Runge-Kutta explicit projection schemes are used for time integration. Implementations on staggered grid arrangement for both uniform grid (UG) and adaptive mesh refinement (AMR) are provided. The new Poisson solvers are employed for AMR cases, whereas the homogeneous trigonometric solver + PFFT can be used in UG. Typical velocity boundary conditions for this problem are implemented.
- The ProtonImaging diagnostics code has been improved. Time resolved proton imaging is now possible, where protons are traced through the domain during several time steps. The original version (tracing of protons during one time step with fixed domain) is still available.
- The code for Radiation-Fluxlimiter-Aware Hydro has been updated. Smoothing of the flux-limiter function within the enhanced Hydro implementation has been implemented and has been shown effective in increasing stability in 1D simulations.
- New Opacity implementations: `BremsstrahlungAndThomson` and `OPAL`. These are for gray opacities.
- In addition to the FLASH4.4 release, the publicly available Python module `opacplot2` has received significant development (credit to JT Laune). It can assist in handling EoS/opacity tables, and includes command line tools to convert various table formats to IONMIX and to compare between different tables. More information can be found in the Flash Center’s GitHub repository at <https://github.com/flash-center/opacplot2>.

New additions in FLASH4.5 since FLASH4.4 include:

- New implementation for synthetic Thomson scattering simulated diagnostic.
- A new 1D simulation of a 2-temperature Supernova evolution using the flux-limiter-aware hydro treatment of radiation, and related code changes.

- A new timestep limiter, which ensures that a given list of variables remains positive-definite during hydrodynamic advection. This can be used as an alternative to lowering the Hydro unit's CFL runtime parameter, and has been shown useful for 3T simulations in particular, where non-conservative equations are considered. This will obviate negative temperature errors when using 3T Hydro and MHD. See [Section 7.1.5.1](#) in the Driver chapter for more information.
- Expanded the Sedov simulation to be initialized from a quasi-analytical profile, and added diagnostic variables to keep track of various simulation error measures.
- New flux limiter "**vanLeer1.5**" for unsplit Hydro and MHD.
- Fixed an issue in which PARAMESH block distribution failed to settle down because of message delay, reported for some machines. This is done by inserting `sleep(1.)` calls in extreme cases.
- Implement saving of laser irradiation (to "lase" variable) for 2DCyl3D ray tracing, too. Previously was 3D only.
- We have included detailed instructions that outline how to easily install FLASH on Windows, Mac OS, and Linux distributions, using Docker. See [Section 2.3](#) in the "Quick Start" chapter.
- **FLASH Interface** is a new graphical interface (GUI) for managing parameter files. and is now available for download. The code was developed by Christopher Walker (cnwalker@uchicago.edu). It currently supports:
  - Configuring and editing existing `flash.par` files;
  - Generating new `flash.par` files by choosing among the full set of possible parameters;
  - Filtering the full set of possible FLASH parameters (found in `setup_params`) via categories;
  - Searching for FLASH params by name, and viewing full descriptions of each parameter's role and function.

The source code, along with download and build instructions can be found at <https://github.com/cnwalker/FLASH-Interface>. Compiled executables for a number of operating systems can be found at <http://flash.rochester.edu/site/flashcode/FLASH-Interface>

- The **PRaLine** code for Proton Radiography Linear reconstruction is now included with the FLASH code tarball and is available for download at <https://github.com/flash-center/PRaLine>. This standalone Python code was originally developed for the magnetic field reconstruction method using proton radiography, discussed in Graziani et al. 2017 (in press), <https://arxiv.org/abs/1603.08617>. Alemayehu Solomon Bogale (alemsolobog@rochester.edu) restructured the code and made it suitable for public release. For more information see the README files in `tools/protonRad` and `tools/protonRad/PRaLine`.

New additions in FLASH4.6 since FLASH4.5 include:

- Implementation of new nonideal MHD effects: Hall term (Cartesian 1D/2D/3D and cylindrical 2D), and Biermann battery (Cartesian 2D/3D and cylindrical 2D) as a source term and a flux-based formulation for 3T.
- Corrections on SpitzerHighZ magnetic resistivity. Now using the NRL plasma formulary equation.
- New high-order reconstruction methods in unsplit Hydro / MHD solvers:
  - PCM: Piecewise Cubic Method, see Lee et al, JCP 341, 230, 2017
  - GP: Gaussian Process, see Reyes et al., JCP 381, 189, 2019; Reyes et al., JSC 76, 443, 2018
- Added support for anisotropic thermal conduction. Anisotropic conductivities depend on the direction of a magnetic field, so anisotropic conduction is meant to be used together with MHD. Currently only available with the uniform Grid (UG) implementation.

- New implementation for simulated Thomson scattering experimental diagnostic using ray tracing. (source/diagnostics/ThomsonScattering)
- Implementation of X-ray imaging simulated experimental diagnostic using ray tracing. (source/diagnostics/XrayImaging)
- Implementation of proton emission simulated experimental diagnostic using ray tracing. (source/diagnostics/ProtonEmission)
- Enabled support for 1D cylindrical laser energy deposition.
- Applied all known patches for working with HDF5 versions 1.10.x.
- Compatibility updates to the setup utility so it works with Python2 and Python3 versions.
- Updated various Config files for Python3 compatibility.

These are the changes from FLASH4.6 to FLASH4.6.1:

- Corrected Ohmic heating term for nonideal MHD in `hy_uhd_unsplitUpdate`: erroneous multiplication by density
- New runtime parameter for optionally specifying laser beam intensities in 1D and 2D Cartesian simulations in a more natural way by determining the beam power profile via beam power, not power per length or area unit.
- Enabled and updated XrayImaging unit, included new unit tests.
- Fixed occasional generation of invalid refinement pattern by PARAMESH in 2D and 3D spherical coordinates.
- Disabled `PM_OPTIMIZE_MORTONBND_FETCHLIST` to avoid rare runtime error in guard cell communication.
- Added modifications to make setup work better with Python3, including removing 8bit characters and TAB character consistency. We have also added some extra features in the setup command.
- Improved detection of case-insensitive file system by setup to avoid unnecessary recompilation by make in some Docker environments.
- Disabled use of 1D profile in default configuration of Sedov example.

These are the changes from FLASH4.6.1 to FLASH4.6.2:

- For unsplit `Hydro` with curvilinear coordinates we have improved the computation of velocity divergence used in determining the strength of artificial viscosity (when `use_avis` is `TRUE`). The effects of artificial viscosity can be expected to change, particularly in cells close to the coordinate axis in 2D cylindrical coordinates.
- Corrected wrong order of operations in an MHD code path when using flux correction: energy corrections (proper accounting of magnetic energy in cells) could be applied too late in interior cells. The problem only occurred in 1D MHD configurations (with the `MHD_StaggeredMesh` code), which could then lead to errors in results of 3T simulations.
- Corrected wrong `Eos` mode in `Eos_wrapped` calls, as well as energy adjustment, in the Biermann battery source term code.
- `LaserRayTrace` and `ThomsonScatteringMain/WithRayTracing`: Improved ray tracing in situations where rays enter the computational domain in locations that have significant electron density by adding ray speed adjustment at domain entry.
- Fixed some problems in `Multitype` EOS that showed up in tests with non-fixed-blocksize uniform grid mode. Convergence of root search should now be improved in some EOS modes.

- Slightly improved time step computation for anisotropic diffusion (Uniform Grid).
- FLASH now requires being compiled against MPI version 2 or higher. Some setups continue to work when compiled against an MPI-1 library, but this will not be true in general.

These are the changes from FLASH4.6.2 to FLASH4.7:

- `LaserRayTrace` and `ThomsonScatteringMain/WithRayTracing`: 3D-cylindrical ray tracing; other ray tracing changes. (experimental)
- Boundary condition aware `LaserRaysCreate` routines that allow rays to enter from periodic/reflecting boundaries with appropriate symmetry transformations.
- Support for 3D cylindrical geometry: split and unsplit Hydro, (experimental)
- Reorganize flux correction for the staggered mesh MHD implementation so that `MagneticResistivity` can be used with `conserveAngField` disabled in cylindrical coordinates.
- Enhancements in implicit solves:
  - Anisotropic thermal conduction now supports; AMR grids with `PARAMESH`, as well as slope limiting to avoid unphysical temperature changes.
  - Periodic boundary conditions are now supported in implicit HYPRE solvers.
  - We have added an operator split HYPRE solver for implicit magnetic diffusion on uniform grids.
- We have add the full Braginskii extended-MHD terms in the generalized OHM's law for the unsplit staggered mesh hydro solver, including:
  - Full Anisotropic Magnetic Resistivity
    - \* `UG +AMR`
    - \*  $\eta_{\parallel}$
    - \*  $\eta_{\perp}$
    - \*  $\eta_{\wedge}$
  - Hall
  - Nernst
  - Seebeck
  - cross-field term
  - Biermann
- The Biermann solver now has a symmetrized use of `SHOKVAR` (previously left-biased)
- New `materialProperties` implementations of transport coefficients, including:
  - Ji & Held - thermal conduction
  - Davies & Wen - thermal electric & resistivity
  - Spitzer Viscosity
  - New multi-species API for calculating  $Z^2$
  - Fixed bug in calculation of  $\tau$  for Epperlein & Haines
  - Consolidate older implementations to use  $Z_{\text{eff}}$
- We have implemented a Current Drive Unit that implements a magnetic field boundary condition for modeling Z-pinch experiments using a variety of models for the current drive.
  - This includes a circuit model that can model systems like the Z-machine and CESZAR.

- We have added native readers in FLASH for tabulated data, including the LANL SESAME format for EoS and the LANL TOPS format for opacities.
- The unsplit hydrodynamics solver has been updated with the option for a volume-of-fluid interface capturing scheme for maintaining sharp material interfaces. Available for 1D/2D uniform grids in Cartesian geometry.
- The I/O routines have been updated for compatibility with Open-MPI 3.
- The python scripts, including `setup`, have been updated to be fully compatible with Python3.

These are the changes from FLASH4.7 to FLASH4.7.1:

- We have added the missing `FL_NONE` flux limiter for time step calculations in extended MHD that use the Nernst term.
- We have removed the deprecated local log-lambda subroutine from Spitzer magnetic resistivity unit that could cause some compile issues.
- We have corrected calls to `PlasmaState_tau` in `hy_uhd_computeDtExtMHD`.
- We have modified the ion viscous heating in `hy_uhd_ragelike` to now use the correct scratch variables in hydro & MHD.
- We have fixed improper memory deallocations in `eos_idealgamma3T` when set up with multitype EoS in non-fixed block size uniform grids.
- We have fixed a bug in implicit magnetic diffusion HYPRE solver that only allowed diffusion of MAGX.
- We have updated the SESAME EoS table reader to use geometric spacing in the interpolated density/temperature grids as well as a species runtime parameter to specify the double format of the table (defaults to E22.15).
- We have made consistent all the output of the Circuit unit (Circuit.dat) so all data uses the same time centering.
- We have fixed a bug in Thomson Scattering diagnostic that occurred when compiled in debug mode.
- We have modified the fast magnetosonic speed calculation in timestep computation to avoid negative numbers.

These are the changes from FLASH4.7.1 to FLASH4.8

- Added python bindings to a subset of flash units using the pybind11 library
- notable routines that may be written in python and coupled to a FLASH simulation include:
  - Simulation initialization & `adjustEvolution`
  - user-defined boundary conditions
  - python function as a runtime parameter file
- Examples are provided in the `source/Simulation/SimulationMain/python` directory
- Added support for CMake builds with the `"-cmake"` setup argument. Site configurations may be provided with a `"site.cmake"` in the corresponding directory in `"sites"`
- Introducing a new unified interface to the HYPRE library that may be used solve generic parabolic PDEs in FLASH in `"source/Grid/GridSolvers/HYPRE/Unified"`. The interface is now fully compatible with AMR and only requires users to specify coefficient tensors and an UNK index for transport coefficients.

- Introducing a new fully implicit operator split viscosity solver built on the Unified HYPRE interface in both UG & AMR grids
- The implicit magnetic resistivity solver has been moved to the Unified HYPRE interface and now supports AMR grids.
- vector diffusion requires HYPRE versions  $\geq 2.24$
- The solver evolves the cell-centered fields and there are two update modes for coupling to the staggered face fields
- In the staggered update, a full divergence preserving constrained-transport update is performed
- In the other, the cell-centered fields are averaged to the faces
- The `*fullState` interfaces for MagneticResistivity & Conductivity are now the only interface
- Those that used `(xdens, xtemp)` as input have been removed
- The PlasmaState unit now calculates the ion-electron equilibration time. There are no longer separate calculations in the Leemore/spitzer heat exchange implementations, and the same calculation can now be used in other material property units.
- The circuit unit has been reorganized.
- Added an interface for setting Dirichlet boundary conditions in the RadTrans MGD solver to be more easily specified from a fixed radiation temperature. See the "RadSlab" provided test problem. This enables radiative ablation problems.
- BiermannSource + 3D now will abort, since the source term not compatible with the staggered mesh update
- 3T MHD simulations using the flux-based Biermann battery solver (`biermannSource=.false.`) no longer require `useBiermann3T` so that Biermann may be used without turning on Hall MHD. This will allow larger timesteps in certain problems.
- Bug fixes:
- Multi-material property TOPS opacity tables are now supported
- A bug in the usage of TOPS tables with the `op.useLogTables` switch has been fixed
- The SESAME reader has been made more robust to edge cases in the file format

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

- New Laser - Async communication (experimental).
- Electron-Entropy Advection in Hydro for non-ideal Eos.
- New cubic and quartic equation solvers have been added and are ready to be used. They return only real cubic and quartic roots. The routines can be found in the `flashUtilites/general` section
- An alternative setup tool "`setup_alt`" intended to be a compatible replacement with a cleaner structure.



## 1.2 External Contributions

Here we list some major contributions to FLASH4 from outside the Flash Center that are included in the distribution. For completeness, we also list such contributions in FLASH3 which have long been included in the release.

- Huang-Greengard based multigrid solver, contributed by Paul Ricker. This contribution was first distributed in FLASH3. Reference: <http://adsabs.harvard.edu/abs/2008ApJS..176..293R>
- Direct solvers for Uniform Grid contributed by Marcos Vanella. The solvers have been in the release since FLASH3. Reference: <http://dx.doi.org/10.1002/cpe.2821>
- Additional Poisson solvers (Martin-Cartwright Geometric Multigrid ported from FLASH2, and new BiPCGStab), and Incompressible Navier-Stokes solver unit, from Marcos Vanella; added to the release code in FLASH4.4.
- Hybrid-PIC code, contributed by Mats Holmström. The contribution has been in the distribution since FLASH4-alpha. Reference: <http://adsabs.harvard.edu/abs/2011arXiv1104.1440H>
- Primordial Chemistry contributed by William Gray. This contribution was added in FLASH4.0. Reference: <http://iopscience.iop.org/0004-637X/718/1/417/>.
- Barnes Hut tree gravity solver contributed by Richard Wunsch. This contribution has been further extended in FLASH 4.2.2 and in the current release and has been developed in collaboration with Frantisek Dinnbier (responsible for periodic and mixed boundary conditions) and Stefanie Walch.
- Sink Particles contributed by Christoph Federrath et al. This contribution has received significant updates over several release. Please refer to <http://iopscience.iop.org/0004-637X/713/1/269/> for details.
- Since FLASH4.2.2, there is a new 'FromFile' implementation of the Stir unit, contributed by Christoph Federrath. The new implementations is sitting beside the older 'Generate' implementation.
- New Flame and Turb units contributed by Dean Townsley, with code developed by Aaron Jackson and Alan Calder. A corresponding paper (Jackson, Townsley, & Calder 2014) on modeling turbulent flames has been published, see <http://stacks.iop.org/0004-637X/784/i=2/a=174>. More information can be found in [Section 18.6](#) and [Section 18.7](#).

## 1.3 Known Issues in This Release

- FLASH assumes by default that it is being compiled against MPI version 2 or higher. If you must compile against an MPI-1 library, define the preprocessor symbol `FLASH_MPI1` for your Fortran compiler, but note that not all code units support this.
- The outflow boundary condition for face-centered variables in the use of solenoidal magnetic field evolution on AMR fails to ensure the solenoidal constraint at the physical outflow boundaries. However, numerical solutions with respect to this error are still physically correct away from the outflow boundaries. This issue may be resolved in future releases.
- The upwind-biased electric field implementation (i.e., `E_upwind=.true.`) for the unsplit staggered mesh solver in some cases fails to satisfy divergence-free magnetic field evolutions at restart. If this is the case for your runs, please use `E_upwind=.false.`
- The new multipole solver is missing the ability to treat a non-zero minimal radius for spherical geometries, and the ability to specify a point mass contribution to the potential.
- The "Split" implementation of the diffusion solver is essentially meant for testing purposes. Moreover, it has not been exercised with PARAMESH.

- Some configurations of hydrodynamic test problems with Chombo grid show worse than expected mass and total energy conservation. Please see the Chombo section in the Hydro chapter of this Guide for details.
- We have experienced the following abort when running IsentropicVortex problem with Chombo Grid: "MayDay: TreeIntVectSet.cpp:1995: Assertion 'bxNumPts != 0' failed. !!!" We have been in contact with the Chombo team to resolve this issue.
- The unsplit MHD solver doesn't support the mode "use\_GravPotUpdate=.true." for 1D when self-gravity is utilized. The solver will still work if it is set to be .false. In this case the usual reconstruction schemes will be used in computing the gravitational accelerations at the predictor step (i.e., at the  $n+1/2$  step) rather than calling the Poisson solver to compute them.
- Time limiting due to burning, even though has an implementation, is turned off in most simulations by keeping the value of parameter `enucDtFactor` very high. The implementation is therefore not well tested and should be used with care.
- Mesh replication is only supported for parallel HDF5 and the experimental derived data type Parallel-NetCDF (`+pnetTypeIO`) I/O implementations. Flash will fail at runtime if multiple meshes are in use without using one of these I/O implementations.
- The unsplit staggered mesh MHD solver shows slight differences (of relative magnitudes of order  $10^{-12}$ ) in restart comparisons (e.g., sfocu comparison), when there are non-zero values of face-centered magnetic fields. With PARAMESH4DEV, the current default Grid implementation, we have observed this problem only with the 'force\_consistency' flag (see Runtime Parameter `gr_pmrpForceConsistency`) turned on.
- In some cases with the default refinement criteria implementation, the refinement pattern at a given point in time of a PARAMESH AMR simulation may be slightly different depending on how often plotfiles and checkpoints are written; with resulting small changes in simulation results. The effect is expected to also be present in previous FLASH versions. This is a side effect of `Grid_restrictAllLevels` calls that happen in IO to prepare all grid blocks for being dumped to file. We have determined that this can only impact how quickly coarser blocks next to a refinement boundary are allowed to further derefine when their better resolved neighbors also derefine, in cases where second-derivative criteria applied to the block itself would allow such derefinement. Users who are concerned with this effect may want to replace the call to `amr_restrict` in `Grid_updateRefinement` with a call to `Grid_restrictAllLevels`, at the cost of a slight increase in runtime.
- The PG compiler fails to compile source files which contain OpenMP parallel regions that reference threadprivate data. This h/4.7 appens in the threaded versions of the Multipole solver and the within block threaded version of split hydro. A workaround is to remove "default(none)" from the OpenMP parallel region.
- Absoft compiler (gcc 4.4.4/Absoft Pro fortran 11.1.x86\_64 with mpich2 1.4.1p1) generates incorrectly behaving code with some files when used with any optimization. More specifically, we have seen this behavior with `gr_markRefineDerefine.F90`, but other files may be vulnerable too. An older version (Absoft Fortran 95 9.0 EP/gcc 4.5.1 with mpich-1.2.7p1) works.
- The `-index-reorder` setup flag does not work in all the configurations. If you wish to use it please contact the FLASH team.
- The `-noclobber` setup option will not force a rebuild of all necessary source files in a FLASH application with derived data type I/O (`+hdf5TypeIO` or `+pnetTypeIO`). Do not use `-noclobber` with derived data type I/O.

## 1.4 About the User's Guide

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

Laney, C. B. *Computational Gasdynamics* (Cambridge UP, 1998)

LeVeque, R. J., Mihalas, D., Dorfi, E. A., and Müller, E., eds. *Computational Methods for Astrophysical Fluid Flow* (Springer, 1998)

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

Toro, E. F. *Riemann Solvers and Numerical Methods for Fluid Dynamics, 2nd Edition* (Springer, 1997)

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

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



**Part I**

**Getting Started**



# Chapter 2

## Quick Start

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

### 2.1 System requirements

You should verify that you have the following:

- A copy of the FLASH source code distribution (as a Unix tar file). To request a copy of the distribution, click on the “Code Request” link on the [FLASH Center web site](#). You will be asked to fill out a short form before receiving download instructions. Please remember the username and password you use to download the code; you will need these to get bug fixes and updates to FLASH.
- A F90 (Fortran 90) compiler and a C compiler. Most of FLASH is written in F90. Information available at the [Fortran Company web site](#) can help you select an F90 compiler for your system. FLASH has been tested with many Fortran compilers. For details of compilers and libraries, see the RELEASE-NOTES available in the FLASH home directory.
- An installed copy of the Message-Passing Interface (MPI) library. A freely available implementation of MPI called MPICH is available from [Argonne National Laboratory](#).
- To use the Hierarchical Data Format (HDF) for output files, you will need an installed copy of the freely available HDF library. The serial version of HDF5 is the current default FLASH format. HDF5 is available from the HDF Group (<http://www.hdfgroup.org/>) of the National Center for Supercomputing Applications (NCSA) at <http://www.ncsa.illinois.edu>. The contents of HDF5 output files produced by the FLASH units are described in detail in [Section 9.1](#).
- To use the Parallel NetCDF format for output files, you will need an installed copy of the freely available PnetCDF library. PnetCDF is available from Argonne National Lab at <http://www.mcs.anl.gov/parallel-netcdf/>. For details of this format, see [Section 9.1](#).
- To use Chombo as the (Adaptive Mesh Refinement) AMR option, you will need an installed copy of the library, available freely from Lawrence Berkeley National Lab at <https://seesar.lbl.gov/anag/chombo>. The use of Chombo is described in [Section 8.7](#).
- To use the Diffuse unit with HYPRE solvers, you will need to have an installed copy of HYPRE, available for free from Lawrence Livermore National Lab at <https://computation.llnl.gov/casc/hypre/software.html>. Versions of HYPRE from 2.7.0b to 2.13 should all work with FLASH4.5.

HYPRE is required for using several critical HEDP capabilities including multigroup radiation diffusion and thermal conduction. Please make sure you have HYPRE installed if you want these capabilities.

- To use the output analysis tools described in this section, you will need a copy of the IDL language from [ITT Visual Information Solutions](#). IDL is a commercial product. It is not required for the analysis of FLASH output, but the `fidlr3.0` tools described in this section require it. (FLASH output formats are described in [Section 9.9](#). If IDL is not available, another visual analysis option is ViSit, described in [Chapter 36](#).) The newest IDL routines, those contained in `fidlr3.0`, were written and tested with IDL 6.1 and above. You are encouraged to upgrade if you are using an earlier version. Also, to use the HDF5 version 1.6.2, analysis tools included in IDL require IDL 6.1 and above. New versions of IDL come out frequently, and sometimes break backwards compatibility, but every effort will be made to support them.
- The GNU make utility, `gmake`. This utility is freely available and has been ported to a wide variety of different systems. For more information, see the entry for `make` in the development software listing at <http://www.gnu.org/>. On some systems `make` is an alias for `gmake`. GNU make is required because FLASH uses macro concatenation when constructing Makefiles.
- A copy of the Python language, version 2.2 or later is required to run the `setup` script. Python can be downloaded from <http://www.python.org>.

## 2.2 Unpacking and configuring FLASH for quick start

To begin, unpack the FLASH source code distribution.

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

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

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

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

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

```
cd object
make
```

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

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

If compilation and linking were not successful, here are a few common suggestions to diagnose the problem:



- 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

basenm  = "sedov_"

lrefine_max = 5
refine_var_1 = "dens"
refine_var_2 = "pres"

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"

sim_profFileName = "/dev/null"
```

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

This example first instructs FLASH to name output files appropriately (through the `basenm` parameter). FLASH is then instructed to use up to five levels of adaptive mesh refinement (AMR) (through the `lrefine_max` parameter), with the actual refinement adapted based on the density and pressure of the solution (parameters `refine_var_1` and `refine_var_2`). 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 Alternative Configuration With a Docker Image

The FLASH code runs natively on Unix environments. Especially if you will be using a personal machine such as a laptop computers to run FLASH, it can be time-consuming and nontrivial to set up the required environment and install the required libraries as described in the previous sections, and in particular to satisfy the system requirements listed above.

As an alternative way to quickly get to a working environment for using FLASH, we use the Docker platform and make a predefined image available. If you want to follow this path, the remainder of this sections gives some quick start instructions.

Download Docker for your operating system and install it following the online instructions at <https://docs.docker.com/get-started>. The image we will be using contains a lightweight Unix distro with everything pre-compiled, tagged as `icksa/flash4-deps`. From the terminal/command line, you will need to download the image with:

```
> docker pull icksa/flash4-deps
```

The container will then be initialized interactively while also loading a shared volume so as to access your file system and the folder where you will unpack the FLASH distribution. Make sure that the volume or folder you want to be shared between the container and the host filesystem is listed under **Preferences => File Sharing**. The Docker command would look something like this in MacOS, with the contents of the shared folder `/Users/petros` residing inside the `/mnt/petros` directory of the container:

```
> docker run -v /Users/petros:/mnt/petros -it icksa/flash4-deps
```

For more information on Docker refer to the above-mentioned link and the Docker User's Guide, which can be found online at <https://docs.docker.com/engine/userguide>.

To visualize the FLASH simulation results we will be primarily using VisIt, an interactive, parallel visualization and graphical analysis tool for viewing scientific data. It can be downloaded for your platform from <https://wci.llnl.gov/simulation/computer-codes/visit/downloads>; see also [Chapter 36](#). While the VisIt GUI is pretty intuitive for simple plots, you can find information on more advanced data analysis methods at <http://www.visitusers.org>.

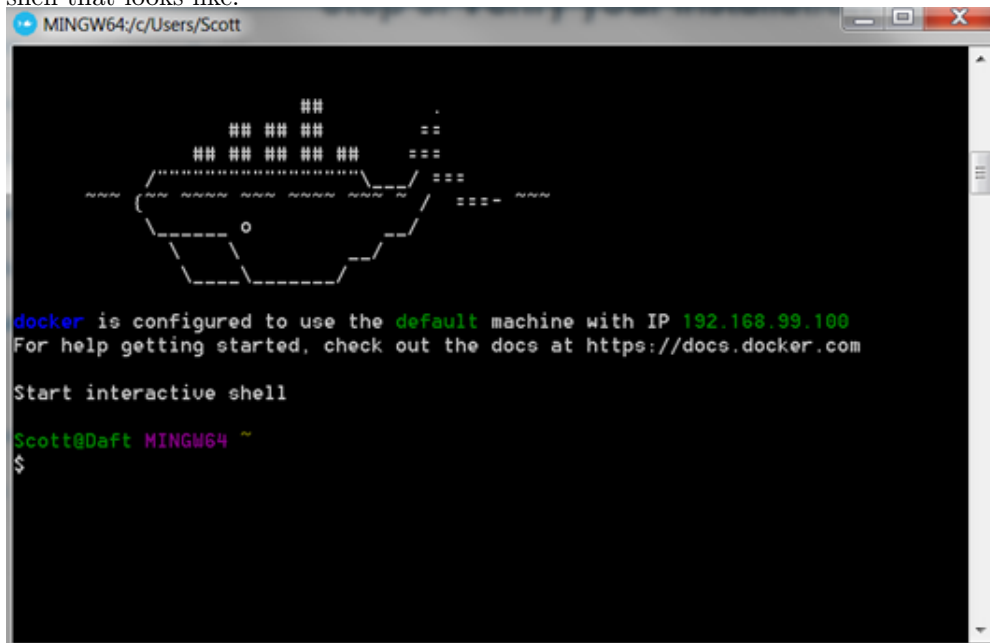
You will need a text editor to modify the code files. The Docker image comes with vim installed but, if you prefer to edit the files on your native platform, feel free to choose whichever software you like. Some suggestions in no particular order:

```
https://www.sublimetext.com
https://notepad-plus-plus.org
https://www.activestate.com/komodo-ide/downloads/edit
https://atom.io
```

The following clarifications are for users of certain older version of Microsoft Windows and can be ignored by other users. Docker for Windows (best, native option) runs on 64-bit Windows 10 Pro, Enterprise and Education (1511 November update, Build 10586 or later). Docker Toolbox (legacy systems) runs on 64-bit Windows 7 or higher.

Setup of Docker Toolbox for legacy Windows systems (e.g. Windows 7 64-bit):

1. Download Docker Toolbox and follow installation instructions from [https://docs.docker.com/toolbox/toolbox\\_install\\_windows/](https://docs.docker.com/toolbox/toolbox_install_windows/).  
Install with default options, including “Create desktop shortcuts”.  
Note: If you already have Oracle VM VirtualBox installed, this can be unchecked.
2. Click the desktop shortcut “Docker Quickstart Terminal”.  
A command terminal should open in which several commands run, terminating with an interactive shell that looks like:



3. From this terminal, download the FLASH-dependencies docker image:  

```
> docker pull icksa/flash4-deps
```
4. To start the flash4-deps docker image with a local folder “C:\Users\Scott\Documents\Docker Test Folder” represented under “/mnt/scott”, execute:  

```
> docker run -v "//c/Users/Scott/Documents/Docker Test Folder:/mnt/scott" -it icksa/flash4-deps
```
5. To quite the docker image at any time, press “Ctrl + D”.

## 2.4 Running FLASH

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

```
mpirun -np  $N$  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 FLASH4 executable accepts an optional command-line argument for the runtime parameters file. If “-par\_file *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 `cs`h the two additional variables can be set using the commands

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

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

```
setenv IDL_PATH "${XFLASH3_DIR}:${IDL_DIR}:${IDL_DIR}/lib"
```

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

### FLASH Transition

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

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

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

When the control panel settings are to your satisfaction, click the ‘Plot’ button to generate the plot. For Postscript or PNG output, a file is created in the current directory. The result should look something like [Figure 2.3](#), although this figure was generated from a run with eight levels of refinement rather than the five used in the quick start example run. With fewer levels of refinement, the Cartesian grid causes the explosion to appear somewhat diamond-shaped. Please see [Chapter 39](#) 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.

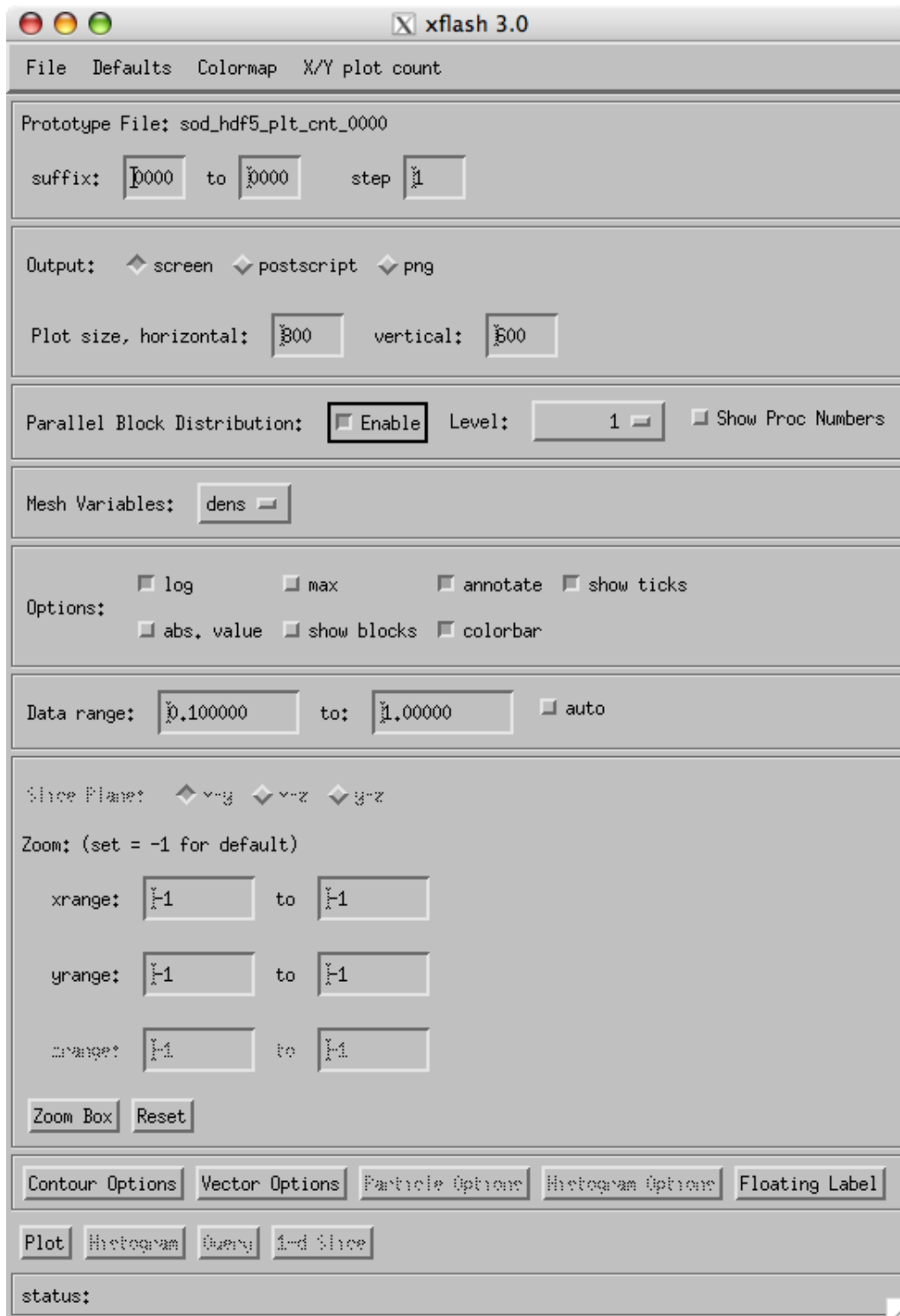


Figure 2.2: The main xflash3 widget.

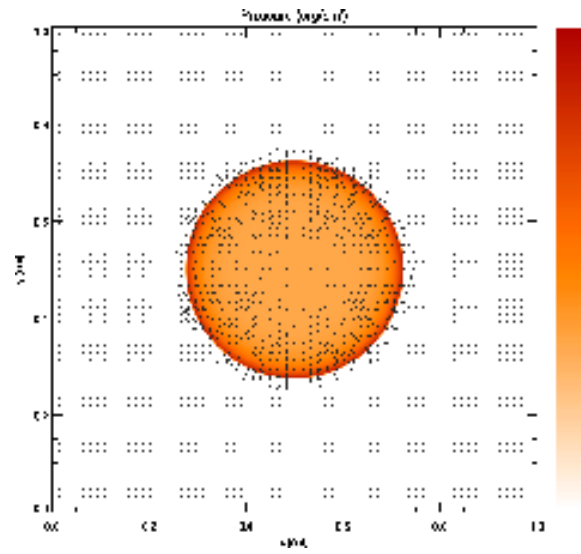


Figure 2.3: Example of `xflash` output for the Sedov problem with eight levels of refinement.





## Chapter 3

# Setting Up New Problems

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

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

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

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

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

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

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

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

### 3.2 Creating a Makefile

The file `Makefile` included in the Simulation directory does not have the standard Makefile format for `make/gmake`. Instead, the setup script generates a complete compilation Makefile from the machine/system specific one (see [Section 5.6](#)) and the unit Makefiles (see [Section 5.7.3](#)).

In general, standard module and routine dependencies are figured out by the `setup` script or are inherited from the directory structure. The `Makefile` for this example is very simple, it only adds the object file for `Simulation_data` to the Simulation unit `Makefile`. The additional object files such as `Simulation_init.o` are already added in the directory above `SimulationMain`.

### 3.3 Creating a Simulation\_data.F90

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

```
module Simulation_data
  implicit none

  !! Runtime Parameters
  real, save :: sim_rhoLeft, sim_rhoRight, sim_pLeft, sim_pRight
  real, save :: sim_uLeft, sim_uRight, sim_xAngle, sim_yAngle, sim_posn
  real, save :: sim_gamma, sim_smallP, sim_smallX

  !! Other unit variables
  real, save :: sim_xCos, sim_yCos, sim_zCos

end module Simulation_data
```

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

### 3.4 Creating a Simulation\_init.F90

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

#### FLASH Transition

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

The basic structure of the routine `Simulation_init` should consist of

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

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

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

### FLASH Transition

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

```
idents=dBaseKeyNumber('dens')
ucons(1,i) = solnData(idents,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.

```
subroutine Simulation_init()

  use Simulation_data
  use RuntimeParameters_interface, ONLY : RuntimeParameters_get
  implicit none
```

```

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

! get the runtime parameters relevant for this problem

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

! Do other initializations
! convert the shock angle parameters

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

sim_xCos = cos(sim_xAngle)

if (NDIM == 1) then
    sim_xCos = 1.
    sim_yCos = 0.
    sim_zCos = 0.

elseif (NDIM == 2) then
    sim_yCos = sqrt(1. - sim_xCos*sim_xCos)
    sim_zCos = 0.

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

end subroutine Simulation_init

```

### 3.5 Creating a Simulation\_initBlock.F90

The routine `Simulation_initBlock` is called by the `Grid` unit to apply initial conditions to the physical domain. If the AMR grid `PARAMESH` is being used, the formation of the physical domain starts at the lowest level of refinement. Initial conditions are applied to each block at this level by calling `Simulation_initBlock`. The `Grid` unit then checks the refinement criteria in the blocks it has created and refines the blocks if the criteria are met. It then calls `Simulation_initBlock` to initialize the newly created blocks. This process repeats until the grid reaches the required refinement level in the areas marked for refinement. The Uniform Grid has only one level, with same resolution everywhere. Therefore, only one block per processor is created and `Simulation_initBlock` is called to initialize this single block. It is important to note that a problem's

`Simulation_initBlock` routine is the same regardless of whether `PARAMESH` or Uniform Grid is being used. The `Grid` unit handles these differences, not the `Simulation` unit.

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

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

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

```
subroutine Simulation_initBlock(blockID)
  ! get the needed unit scope data
  use Simulation_data, ONLY: sim_posn, sim_xCos, sim_yCos, sim_zCos,&
                           sim_rhoLeft, sim_pLeft, sim_uLeft,   &
                           sim_rhoRight, sim_pRight, sim_uRight, &
                           sim_smallX, sim_gamma, sim_smallP
  use Grid_interfaces, ONLY : Grid_getBlkIndexLimits, Grid_getCellCoords, &
                           Grid_putPointData

  implicit none

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

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

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

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

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

```

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

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

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

```

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

### FLASH Transition

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

```

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

! get the size along each dimension for allocation and then allocate
sizeX = blkLimitsGC(HIGH,IAXIS)
sizeY = blkLimitsGC(HIGH,JAXIS)
sizeZ = blkLimitsGC(HIGH,KAXIS)
allocate(xLeft(sizeX))
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_getCellCoords(IAXIS, blockID, LEFT_EDGE,  gcell, xLeft,  sizeX)
call Grid_getCellCoords(IAXIS, blockID, CENTER,    gcell, xCenter, sizeX)
call Grid_getCellCoords(IAXIS, blockID, RIGHT_EDGE, gcell, xRight, sizeX)
call Grid_getCellCoords(JAXIS, blockID, CENTER,    gcell, yCoord, sizeY)
call Grid_getCellCoords(KAXIS, blockID, CENTER,    gcell, zCoord, sizeZ)

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

  do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
    yy = yCoord(j) ! center coordinates in the y-direction
    lPosn = lPosn0 - yy*sim_yCos/sim_xCos ! The position of the
                                           ! shock in the current yz-row.

    dataSize = 1 ! for Grid put data function, we are
                  ! initializing a single cell at a time and
                  ! sending it to Grid

    do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)

      xx = xCenter(i) ! center coordinate along x
      xxL = xLeft(i)  ! left coordinate along y
      xxR = xRight(i) ! right coordinate along z

```

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

```

if (xxR <= lPosn) then
  rhoZone = sim_rhoLeft
  presZone = sim_pLeft

  velxZone = sim_uLeft * sim_xCos
  velyZone = sim_uLeft * sim_yCos
  velzZone = sim_uLeft * sim_zCos

  ! initialize cells which straddle the shock. Treat them as though 1/2 of
  ! the cell lay to the left and 1/2 lay to the right.
elseif ((xxL < lPosn) .and. (xxR > lPosn)) then

  rhoZone = 0.5 * (sim_rhoLeft+sim_rhoRight)
  presZone = 0.5 * (sim_pLeft+sim_pRight)

  velxZone = 0.5 * (sim_uLeft+sim_uRight) * sim_xCos
  velyZone = 0.5 * (sim_uLeft+sim_uRight) * sim_yCos
  velzZone = 0.5 * (sim_uLeft+sim_uRight) * sim_zCos

```



```

! 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, ENER_VAR, EXTERIOR, axis, enerZone)
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 Creating a Simulation\_freeUserArrays.F90

From within `Simulation_init`, the user may create large allocatable arrays that are used for initialization within `Simulation_initBlock` or some other routine. An example would be a set of arrays that are used to interpolate fields onto the blocks as they are being created. If these arrays use a lot of allocated memory, the subroutine `Simulation_freeUserArrays` gives the user a chance to free this memory using `deallocate` statements. This subroutine is called after all initialization steps have been performed, and the default implementation is a stub which does nothing. A customized version for a particular setup may be made by copying the stub from the `Simulation` directory and editing it as need be.

### 3.7 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 of 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.

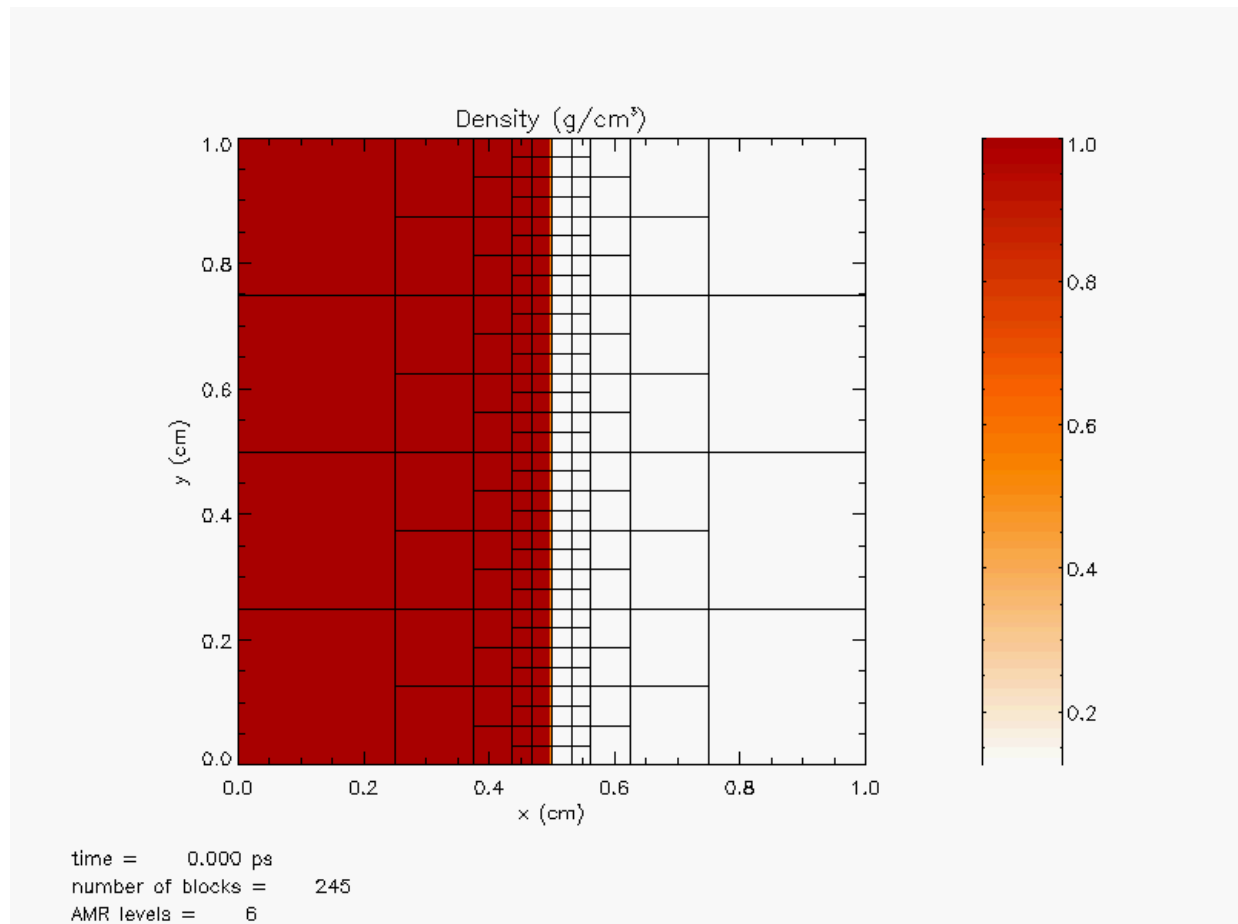


Figure 3.1: Image of the initial distribution of density in **example** setup.

## 3.8 Running your simulation

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

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

```
mpirun -np N flash4
```

or

```
mpiexec -n N flash4
```

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

```
./flash4
```

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

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

- Copy a FLASH parfile, normally **flash.par**, into the run directory.
- If FLASH was configured with **Paramesh4.0** in **LIBRARY** mode (this is *not* the default): copy the file **amr\_runtime\_parameters**, into the run directory. This file should have been generated in the object directory by running **./setup**.
- Some code units use additional data or table files, which are also copied into the object directory by **./setup**. (These files typically match one of the patterns **\*.dat** or **\*\_table\***.) copy or move those files into the run directory, too, if your simulation needs them.



## **Part II**

# **The FLASH Software System**





## Chapter 4

# Overview of FLASH architecture

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

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

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

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

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

The **simulation** unit is of particular significance because it defines how a FLASH application will be built and executed. When the setup script is invoked, it begins by examining the simulation’s `Config` file, which specifies the units required for the application, and the simulation-specific runtime parameters. Initial conditions for the problem are provided in the routines `Simulation_init` and `Simulation_initBlock`. As mentioned in [Chapter 3](#), the `Simulation` unit allows the user to overwrite any of FLASH’s default function implementations by writing a function with the same name in the application-specific directory. Additionally, runtime parameters declared in the simulation’s `Config` file override definitions of same-named parameters in other FLASH units. These helpful features enable users to customize their applications, and are described in more detail below in [Section 4.1](#) and online in [Architecture Tips](#). The simulation unit also provides some useful interfaces for modifying the behaviour of the application while it is running. For example there is an interface `Simulation_adjustEvolution` which is called at every time step. Most applications would use the null implementation, but its implementation can be placed in the `Simulation` directory of the application to customize behavior. The API functions of the `Simulation` unit are unique in that except `Simulation_initSpecies`, none of them have any default general implementations. At the API level there are the null implementations, actual implementations exist only for specific applications. The general implementations of `Simulation_initSpecies` exist for different classes of applications, such as those utilizing nuclear burning or ionization.

### FLASH Transition

Why the name change from “modules” in FLASH2 to “units” in FLASH3? The term “module” caused confusion among users and developers because it could refer both to a FORTRAN90 module and to the FLASH-specific code entity. In order to avoid this problem, FLASH3 started using the word “module” to refer exclusively to an F90 module, and the word “unit” for the basic FLASH code component. Also, FLASH no longer uses F90 modules to implement units. Fortran’s limitation of one file per module is too restrictive for some of FLASH4’s units, which are too complex to be described by a single file. Instead, FLASH4 uses interface blocks, which enable the code to take advantage of some of the advanced features of FORTRAN90, such as pointer arguments and optional arguments. Interface blocks are used throughout the code, even when such advanced features are not called for. For a given unit, the interface block will be supplied in the file “Unit\_interface.F90”. Please note that files containing calls to API-level functions must include the line `use Unit,` `ONLY: function-name1, function-name2, etc.` at the top of the file.

## 4.1 FLASH Inheritance

FORTTRAN90 is not an object-oriented language like Java or C++, and as such does not implement those languages’ characteristic properties of inheritance. But FLASH takes advantage of the Unix directory structure to implement an inheritance hierarchy of its own. Every child directory in a unit’s hierarchy inherits all the source code of its parent, thus eliminating duplication of common code. During setup, source files in child directories override same-named files in the parent or ancestor directories.

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

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

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

## 4.2 Unit Architecture

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

A good example to examine is the `Grid` unit API. Some of the accessor functions in this unit are `Grid_getCellCoords`, `Grid_getBlkData`, and `Grid_putBlkData`, while `Grid_fillGuardCells` and `Grid_updateRefinement` are examples of API routines which modify data in the `Grid` unit.

A unit can have more than one implementation of its API. The `Grid` Unit, for example, has both an **Adaptive Grid** and a **Uniform Grid** implementation. Although the implementations are different, they

both conform to the **Grid** API, and therefore appear the same to the outside units. This feature allows users to easily swap various unit implementations in and out of a simulation without affecting the way other units communicate. Code does not have to be rewritten if the user decides to implement the uniform grid instead of the adaptive grid.

### 4.2.1 Stub Implementations

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

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

  implicit none

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

  count=0
  listOfBlocks(1)=0

  return
end subroutine Grid_getListOfBlocks
```

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

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

#### FLASH Transition

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

### 4.2.2 Subunits

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

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

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

### 4.2.3 Unit Data Modules, `_init`, and `_finalize` routines

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

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

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

```
subroutine Driver_evolveFlash()

  use Driver_data, ONLY: dr_myPE, dr_numProcs, dr_nbegin, &
    dr_nend, dr_dt, dr_wallClockTimeLimit, &
```

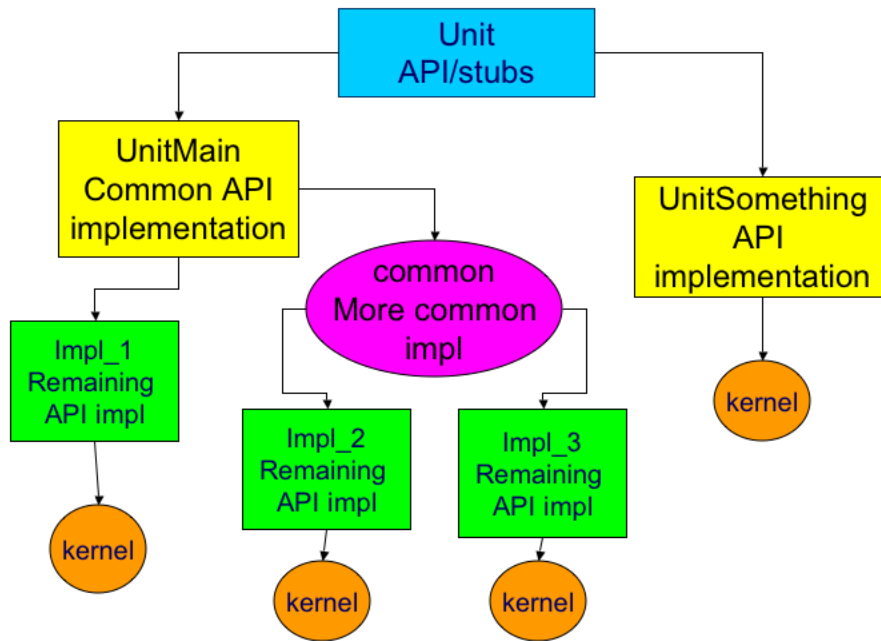


Figure 4.1: The unit hierarchy and inheritance.

```

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 15.1.2](#)). **PPM** is a directional solver and requires that data be passed to it in vector form. Routines like **hy\_sweep** and **hy\_block** are helper functions that collect data from the **Grid** unit, and put it in the format required by **PPM**. These routines also make sure that data stay in thermodynamic equilibrium through calls to the **Eos** unit. Neither **PARAMESH 2**, nor **PPM** has any knowledge of units outside their own.

### 4.3 Unit Test Framework

In keeping with good software practice, FLASH4 incorporates a unit test framework that allows for rigorous testing and easy isolation of errors. The components of the unit test show up in two different places in the FLASH source tree. One is a dedicated path in the **Simulation** unit, **Simulation/SimulationMain/-unitTest/UnitTestName**, where *UnitTestName* is the name of a specific unit test. The other place is a subdirectory called **unitTest**, somewhere in the hierarchy of the corresponding unit which implements a function **Unit\_unitTest** and any helper functions it may need. The primary reason for organizing unit tests in this somewhat confusing way is that unit tests are special cases of simulation setups that also need extensive access to internal data of the unit being tested. By splitting the unit test into two places, it is possible to meet both requirements without violating unit encapsulation. We illustrate the functioning of the unit test framework with the unit test of the **Eos** unit. For more details please see [Section 17.7](#). The **Eos** unit test needs its own version of the routine **Driver.evolveFlash** which makes a call to its **Eos\_unitTest** routine. The initial conditions specification and unit test specific **Driver.evolveFlash** are placed in **Simulation/SimulationMain/unitTest/Eos**, since the **Simulation** unit allows any substitute FLASH function to be placed in the specific simulation directory. The function **Eos\_unitTest** resides in **physics/Eos/unitTest**, and therefore has access to all internal **Eos** data structures and helper functions.

## Chapter 5

# The FLASH configuration script (setup)

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

The primary job of the `setup` script is to

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

More description of how `setup` and the FLASH4 architecture interact may be found in [Chapter 4](#). Here we describe its usage.

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

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

---

<sup>1</sup>if a machine has multiple hostnames, setup tries them all



After running `setup`, the user can create the FLASH executable by running `make` in the `object` directory. Note that the FLASH build system assumes that the command `make` invokes GNU Make and is unlikely to work properly with other implementations of the `make` command. On some systems it may be necessary to invoke GNU Make under the name `gmake`.

### FLASH Transition

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

### Save some typing

- All the setup options can be shortened to unambiguous prefixes, *e.g.* instead of `./setup -auto <problem-name>` one can just say `./setup -a <problem-name>` since there is only one `setup` option starting with `a`.
- The same abbreviation holds for the problem name as well. `./setup -a IsentropicVortex` can be abbreviated to `./setup -a Isen` assuming that `IsentropicVortex` is the only problem name which starts with `Isen`.
- Unit names are usually specified by their paths relative to the source directory. However, `setup` also allows unit names to be prefixed with an extra “source/”, allowing you to use the TAB-completion features of your shell like this

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

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

### Reduce compilation time

- To reuse compiled code when changing setup configurations, use the `-noclobber` setup option. For details see [Section 5.2](#).

## 5.1 Setup Arguments

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

1. *Setup Options* (example: `-auto`) begin with a dash and are built into the setup script itself. Many of the most commonly used arguments are setup options.
2. *Setup Variables* (example: `species=air,h2o`) are defined by individual units. When writing a `Config` file for any unit, you can define a setup variable. [Section 5.4](#) explains how setup variables can be created and used.



Table 5.1: List of Commonly Used Setup Arguments

Argument	Description
<code>-auto</code>	this option should almost always be set
<code>-unit=&lt;unit&gt;</code>	include a specified unit
<code>-objdir=&lt;dir&gt;</code>	specify a different object directory location
<code>-debug</code>	compile for debugging
<code>-opt</code>	enable compiler optimization
<code>-n[xyb]b=#</code>	specify block size in each direction
<code>-maxblocks=#</code>	specify maximum number of blocks per process
<code>-[123]d</code>	specify number of dimensions
<code>-maxblocks=#</code>	specify maximum number of blocks per process
<code>+cartesian</code>	use Cartesian geometry
<code>+cylindrical</code>	use cylindrical geometry
<code>+polar</code>	use polar geometry
<code>+spherical</code>	use spherical geometry
<code>+noio</code>	disable IO
<code>+ug</code>	use the uniform grid in a fixed block size mode
<code>+nofbs</code>	use the uniform grid in a non-fixed block size mode
<code>+pm2</code>	use the PARAMESH2 grid
<code>+pm40</code>	use the PARAMESH4.0 grid
<code>+pm4dev</code>	use the PARAMESH4DEV grid
<code>+uhd</code>	use the Unsplit Hydro solver
<code>+usm</code>	use the Unsplit Staggered Mesh MHD solver
<code>+splitHydro</code>	use a split Hydro solver

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

[Table 5.1](#) shows a list of some of the basic setup arguments that every FLASH user should know about. A comprehensive list of all setup arguments can be found in [Section 5.2](#) alongside more detailed descriptions of these options.

## 5.2 Comprehensive List of Setup Arguments

### `-verbose=<verbosity>`

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

### `-auto`

Normally, `setup` requires that the user supply a plain text file called `Units` (in the `object` directory<sup>2</sup>) that specifies the units to include. A sample `Units` file appears in [Figure 5.1](#). Each line is either a comment (preceded by a hash mark (`#`)) or the name of a an include statement of the form `INCLUDE unit`. Specific implementations of a unit may be selected by specifying the complete path to the implementation in question; If no specific implementation is requested, `setup` picks the default listed in the unit's `Config` file.

<sup>2</sup>Formerly, (in FLASH2) it was located in the FLASH root directory

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

#### `-cmake`

This option will enable `setup` to configure the object directory for a build using `cmake` to compile and configure dependencies of FLASH.

#### `-[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.

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

The default `Makefile` built by `setup` will use the optimized setting (`-opt`) for compilation and linking. Using `-debug` will force `setup` to use the flags relevant for debugging (*e.g.*, including `-g` in the compilation line). The user may use the option `-test` to experiment with different combinations of compiler and linker options. Exactly which compiler and linker options are associated with each of these flags is specified in `sites/<hostname>/Makefile*` where `<hostname>` is the hostname of the machine on which FLASH is running.

For example, to tell an Intel Fortran compiler to use real numbers of size 64 when the `-test` option is specified, the user might add the following line to his/her `Makefile.h`:

```
FFLAGS_TEST = -real_size 64
```

`-objdir=<dir>`

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

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

Use the specified `<unit>` in setting up the problem.

`-curvilinear`

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

`-defines=<def>[,<def>]...`

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

`[-fbs|-nofbs]`

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

`-geometry=<geometry>`

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

`-gridinterpolation=<scheme>`

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

- **monotonic**

This scheme attempts to ensure that monotonicity is preserved in interpolation, so that interpolation does not introduce small-scale non-monotonicity in the data.

The **monotonic** scheme is required for curvilinear coordinates and is automatically enabled if a non-`cartesian` geometry is chosen with the `-geometry` flag. For AMR `Grid` implementations, This flag will automatically add additional directories so that appropriate data interpolation methods are compiled it. The **monotonic** scheme is the default (by way of the `+default` shortcut), unlike in FLASH2.

- **native**

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

### Change in Interpolation

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

```

#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

### 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 in FLASH4 for compatibility with alternative grids. This is supported by both the Uniform Grid as well as PARAMESH, and is currently required for the **Chombo** grid.

#### **-makehide**

Ordinarily, the commands being executed during compilation of the FLASH executable are sent to standard out. It may be that you find this distracting, or that your terminal is not able to handle these long lines of display. Using the option **-makehide** causes **setup** to generate a **Makefile** so that GNU **make** 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 **make** 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 **make** 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** .

#### **-pyparfile=<filename>**

This causes **setup** to copy the specified runtime-parameters file in the simulation directory to the **object** directory with the new name **flashParm.py** .

**-append-parfiles**=[location1/]<filename1>[, [location2/]<filename2>]...

This option takes a comma-separated list of names of parameter files and combines them into one **flash.par** file in the **object** directory. File names without an absolute path are taken to be relative to the simulation directory, as for the **-parfile** option.

To use such a combined **flash.par** in case of runtime parameters occurring more than once, note that when FLASH reads a parameter file, the last instance of a runtime parameter supersedes previous ones.

If both **-append-parfiles** and **-parfile** are used, the files from the list are appended to the single parfile given by the latter in the order listed. If used with **-parfile**, **-append-parfiles** can append one or more parfiles to the one given by **-parfile**. If you only use **-append-parfiles** and not **-parfile** and give it fewer than two paths, an error will result. If more than one **-append-parfiles** option appears, the lists are concatenated in the order given.

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

This option instructs **setup** to adjust the particle methods for a particular particle type. It can only be used when a particle type has already been registered with a **PARTICLETYPE** line in a **Config** file (see [Section 6.6.1](#)). A possible scenario for using this option involves the user wanting to use a different passive particle initialization method without modifying the **PARTICLETYPE** line in the simulation **Config** file. In this case, an additional **-particlemethods=TYPE=passive,INIT=cellmass** adjusts the initialization method associated with passive particles in the **setup** generated **Particles\_specifyMethods()** subroutine. Since the specification of a method for mapping and initialization requires inclusions of appropriate implementations of **ParticlesMapping** and **ParticlesInitialization** subunits, and the specification of a method for time advancement requires inclusion of an appropriate implementation under **ParticlesMain**, it is the user's responsibility to adjust the included units appropriately. For example a user may want 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 21](#), especially [Section 21.3](#).

**-portable**

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

**-site**=<site>

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

**-unitsfile**=<filename>

This causes **setup** to copy the specified file to the **object** directory as **Units** before setting up the problem. This option can be used when **-auto** is not used, to specify an alternate **Units** file.

`-with-library=<libname>[,args], -library=<libname>[,args]`

This option instructs `setup` to link in the specified library when building the final executable. A *library* is a piece of code which is independent of FLASH. Internal libraries are those libraries whose code is included with FLASH. The `setup` script supports external as well as internal libraries. Information about external libraries is usually found in the site specific Makefile. The additional `args` if any are library-specific and may be used to select among multiple implementations. For more information see [Library-HOWTO](#).

`-tau=<makefile>`

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

`-without-library=<libname>`

Negates a previously specified `-with-library=<libname>[,args]`

`-without-unit=<unit>`

This removes all units specified in the command line so far, which are children of the specified unit (including the unit itself). It also negates any REQUESTS keyword found in a `Config` file for units which are children of the specified unit. However it does not negate a REQUIRES keyword found in a `Config` file.

`+default`

This shortcut specifies using basic default settings and is equivalent to the following:

`--with-library=mpi +io +grid-gridinterpolation=monotonic`

`+noio`

This shortcut specifies a simulation without IO and is equivalent to the following:

`--without-unit=physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser/LaserIO`  
`--without-unit=IO`

`+io`

This shortcut specifies a simulation with basic IO and is equivalent to the following:

`--with-unit=IO`

`+serialIO`

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

`+parallelIO`

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

`+hdf5`

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

`IO = hdf5`

`+pnetcdf`

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

`IO = pnetcdf`

**+hdf5TypeIO**

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

```
+io +parallelIO +hdf5 typeIO=True
```

**+pnetTypeIO**

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

```
+io +parallelIO +pnetcdf typeIO=True
```

**+nolog**

This shortcut specifies a simulation without log capability it is equivalent to the following:

```
-without-unit=monitors/Logfile
```

**+grid**

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

```
-unit=Grid
```

**+ug**

This shortcut specifies a simulation using a uniform grid, it is equivalent to the following:

```
+grid Grid=UG
```

**+pm2**

This shortcut specifies a simulation using Paramesh2 for the grid, it is equivalent to the following:

```
+grid Grid=PM2
```

**+pm40**

This shortcut specifies a simulation using Paramesh4.0 for the grid, it is equivalent to the following:

```
+grid Grid=PM40
```

**+pm3**

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

```
+pm40
```

**+chombo\_ug**

This shortcut specifies a simulation using a Chombo uniform grid, it is equivalent to the following:

```
-unit=Grid/GridMain/Chombo/UG -index-reorder Grid=Chombo -maxblocks=1 -nofbs  
-makefile=chombo chomboCompatibleHydro=True
```

**+chombo\_amr**

This shortcut specifies a simulation using a Chombo amr grid, it is equivalent to the following:

```
-unit=Grid/GridMain/Chombo/AMR -index-reorder Grid=Chombo -nofbs  
-makefile=chombo chomboCompatibleHydro=True
```

**+pm4dev\_clean**

This shortcut specifies a simulation using a version of Paramesh 4 that is closer to the version available on sourceforge. It is equivalent to:

```
+grid Grid=PM4DEV ParameshLibraryMode=True
```

**+pm4dev**

This shortcut specifies a simulation using a modified version of Paramesh 4 that includes a more scalable way of filling the `surr.blks` array. It is equivalent to:

```
+pm4dev_clean FlashAvoidOrrery=True
```



**+8wave**

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

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

**+usm**

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

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

**+pureHydro**

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

**+splitHydro**

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

```
--unit=physics/Hydro/HydroMain/split -without-unit=physics/Hydro/HydroMain/unsplit
SplitDriver=True
```

**+unsplitHydro**

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

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

**+uhd**

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

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

**+supportPPMUpwind**

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

**+cube64**

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

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

**+cube32**

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

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

**+cube16**

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

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

**+ptio**

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

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

**+rnf**

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

```
-3d -nxb=8 -nyb=16 -nzb=32 -nofbs +ug
```

**+nofbs**

This shortcut specifies a simulation using a uniform grid with a non-fixed block size. It is equivalent to:

```
-nofbs +ug parallelIO=True
```

**+curvilinear**

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

```
-curvilinear
```

**+cartesian**

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

```
-geometry=cartesian
```

**+spherical**

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

```
-geometry=spherical
```

**+polar**

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

```
-geometry=polar
```

**+cylindrical**

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

```
-geometry=cylindrical
```

**+curv-pm2**

This shortcut specifies a simulation using curvilinear coordinates along with Paramesh2, it is equivalent to:

```
+pm2 -unit=Grid/GridMain/paramesh/Paramesh2
-with-unit=Grid/GridMain/paramesh/Paramesh2/monotonic
```

**+spherical-pm2**

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

```
+pm2 +spherical
```

**+ptdens**

This shortcut specifies a simulation using passive particles initialized by density. It is equivalent to:

```
-without-unit=Particles/ParticlesInitialization/Lattice
-without-unit=Particles/ParticlesInitialization/WithDensity/CellMassBins
-unit=Particles/ParticlesMain
-unit=Particles/ParticlesInitialization/WithDensity
-particlemethods=TYPE=passive,INIT=With_Density
```

**+npg**

This shortcut specifies a simulation using NO'PERMANENT'GUARDCELLS mode in Paramesh4. It is equivalent to:

```
npg=True
```

**+mpole**

This shortcut specifies a simulation using multipole gravity, it is equivalent to:

```
-with-unit=physics/Gravity/GravityMain/Poisson/Multipole
```

**+longrange**

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

**+gravPfftNofbs**

This shortcut specifies a simulation using FFT based gravity solve on a uniform grid with no fixed block size. It is equivalent to:

`+ug +nofbs -with-unit=physics/Gravity/GravityMain/Poisson/Pfft`

**+gravMgrid**

This shortcut specifies a simulation using a multigrid based gravity solve. It is equivalent to:

`+pm40 -with-unit=physics/Gravity/GravityMain/Poisson/Multigrid`

**+gravMpole**

This shortcut specifies a simulation using multipole gravity, it is equivalent to:

`-with-unit=physics/Gravity/GravityMain/Poisson/Multipole`

**+noDefaultMpole**

This shortcut specifies a simulation *\*not\** using the multipole based gravity solve. It is equivalent to:

`-without-unit=Grid/GridSolvers/Multipole`

**+noMgrid**

This shortcut specifies a simulation *\*not\** using the multigrid based gravity solve. It is equivalent to:

`-without-unit=physics/Gravity/GravityMain/Poisson/Multigrid`

**+newMpole**

This shortcut specifies a simulation using the new multipole based gravity solve. It is equivalent to:

`+noMgrid +noDefaultMpole +gravMpole -with-unit=Grid/GridSolvers/Multipole_new`

**+mpi1**

This shortcut specifies a simulation using the MPI-1 standard. It is equivalent to:

`mpi1=True -defines=FLASH_MPI1`

**+mpi2**

This shortcut specifies a simulation using the MPI-2 standard. It is equivalent to:

`mpi2=True -defines=FLASH_MPI2`

**+mtmmmt**

This shortcut specifies use of the MultiTemp/MultiType and Tabulated EOSes (for HEDP simulations). It is equivalent to:

`-unit=physics/Eos/EosMain/multiTemp/Multitype -unit=physics/Eos/EosMain/Tabulated  
 Mtmmmt=1`

**+3t**

This shortcut sets a variable and a preprocessor symbol to request MultiTemp implementations of some units. It is equivalent to:

`ThreeT=1 -defines=FLASH_3T`

**+uhd3t**

This shortcut specifies a simulation using unsplit hydro with MultiTemp EOS. It is equivalent to:

`+3t -without-unit=physics/Hydro/HydroMain/split  
 -with-unit=physics/Hydro/HydroMain/unsplit/Hydro_Unsplit`

**+usm3t**

This shortcut specifies a simulation using unsplit MHD with MultiTemp EOS. It is equivalent to:

```
+3t -without-unit=physics/Hydro/HydroMain/split
--with-unit=physics/Hydro/HydroMain/unsplit/MHD_StaggeredMesh
--without-unit=physics/Hydro/HydroMain/split/MHD_8Wave
```

**+mgd**

This shortcut specifies a simulation using the MGD (magneto gas dynamic) radiative transfer module. It is equivalent to:

```
-unit=physics/materialProperties/Opacity -unit=physics/RadTrans/RadTransMain/MGD
```

**+laser**

This shortcut specifies use of source terms for energy deposition. It is equivalent to:

```
-unit=physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser
--without-unit=Particles
```

**+pic**

This shortcut specifies use of proper particle units to perform PIC (particle in cell) method. It is equivalent to:

```
+ug -unit=Grid/GridParticles/GridParticlesMove
--without-unit=Grid/GridParticles/GridParticlesMove/UG
--without-unit=Grid/GridParticles/GridParticlesMove/UG/Directional
```

**+cppapi**

This shortcut requests the C++ API bindings to the FLASH units that implement a CppAPI subunit. It is equivalent to:

```
cppAPI=1 -cppapi --with-unit=source/Cpp/CppMain
```

**+python**

This shortcut requests the bindings to the pybind11 python module `pyFlash4` as well as the bindings to the embedded python interpreter allowing for runtime parameters to be specified from a python script `flashParm.py`. It is equivalent to:

```
python=1 -pyflash --with-unit=source/Python/PythonMain +cppapi
```

**Grid**

This setup variable can be used to specify which gridding package to use in a simulation:

Name: `Grid`

Type: `String`

Values: `PM4DEV`, `PM40`, `UG`, `PM2`, `Chombo`

**IO**

This setup variable can be used to specify which IO package to use in a simulation:

Name: `IO`

Type: `String`

Values: `hdf5`, `pnetcdf`, `MPIHybrid`, `MPIDump`, `direct`

**parallelIO**

This setup variable can be used to specify which type of IO strategy will be used. A “parallel” strategy will be used if the value is true, a “serial” strategy otherwise.

Name: `parallelIO`

Type: `Boolean`

Values: `True`, `False`

**fixedBlockSize**

This setup variable indicates whether or not a fixed block size is to be used. This variable should not be assigned explicitly on the command line. It defaults to **True**, and the setup options **-nofbs** and **-fbs** modify the value of this variable.

Name: **fixedBlockSize**

Type: **Boolean**

Values: **True, False**

**nDim**

This setup variable gives the dimensionality of a simulation. This variable should not be set explicitly on the command line, it is automatically set by the setup options **-1d**, **-2d**, and **-3d**.

Name: **nDim**

Type: **integer**

Values: **1,2,3**

**GridIndexOrder**

This setup variable indicates whether the **-index-reorder** setup option is in effect. This variable should not be assigned explicitly on the command line.

Name: **GridIndexOrder**

Type: **Boolean**

Values: **True, False**

**nxb**

This setup variable gives the number of zones in a block in the X direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option **-nxb**.

Name: **nxb**

Type: **integer**

**nyb**

This setup variable gives the number of zones in a block in the Y direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option **-nyb**.

Name: **nyb**

Type: **integer**

**nzb**

This setup variable gives the number of zones in a block in the Z direction. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option **-nzb**.

Name: **nzb**

Type: **integer**

**maxBlocks**

This setup variable gives the maximum number of blocks per processor. This variable should not be assigned explicitly on the command line, it is automatically set by the setup option **-maxblocks**.

Name: **maxBlocks**

Type: **integer**

**ParameshLibraryMode**

If true, the setup script will generate file **amr\_runtime\_parameters** from template **amr\_runtime\_parameters.tpl** found in either the object directory (preferred) or the setup script (bin) directory. Selects whether Paramesh4 should be compiled in **LIBRARY** mode, i.e., with the preprocessor symbol **LIBRARY** defined.

Name: **ParameshLibraryMode**

Type: **Boolean**

Values: **True, False**

**PfftSolver**

PfftSolver selects a PFFT solver variant when the hybrid (*i.e.*, Multigrid with PFFT) Poisson solver is used.

Name: **PfftSolver**

Type: **String**

Values: **DirectSolver** (default), **HomBcTrigSolver**, others (unsupported) if recognized in `source/Grid/GridSolvers/Multigrid/PfftTopLevelSolve/Config`

**SplitDriver**

If True, a **Split Driver** implementation is requested.

Name: **SplitDriver**

Type: **Boolean**

**Mtmmmt**

Automatically set **True** by `+mtmmmt` shortcut. When true, this option activates the MTMMMT EOS.

Name: **Mtmmmt**

Type: **Boolean**

**mgd\_meshgroups**

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

Name: **mgd\_meshgroups**

Type: **Integer**

**species**

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

Name: **species**

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

**ed\_maxPulses**

Name: **ed\_maxPulses**

Type: **integer**

Remark: Maximum number of laser pulses (defaults to 5)

**ed\_maxBeams**

Name: **ed\_maxBeams**

Type: **integer**

Remark: Maximum number of laser beams (defaults to 6)

**threadHydroBlockList**

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

Name: **threadHydroBlockList**

Type: **Boolean**

Values: **True**, **False**

**threadMpoleBlockList**

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

Name: **threadMpoleBlockList**

Type: **Boolean**

Values: **True, False**

**threadRayTrace**

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

Name: **threadRayTrace**

Type: **Boolean**

Values: **True, False**

**threadHydroWithinBlock**

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

Name: **threadHydroWithinBlock**

Type: **Boolean**

Values: **True, False**

**threadEosWithinBlock**

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

Name: **threadEosWithinBlock**

Type: **Boolean**

Values: **True, False**

**threadMpoleWithinBlock**

This is used to turn on within block OPENMP threading of then multipole routine.

Name: **threadMpoleWithinBlock**

Type: **Boolean**

Values: **True, False**

### Dependencies among libraries

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

## 5.3 Using Shortcuts

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

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

you can just type

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

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

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

```

# 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

The “default” shortcut is special. `setup` always prepends `+default` to its command line thus making `./setup -a Sod` equivalent to `./setup +default -a Sod`. Thus changing the default IO to “hdf5/parallel”, is as simple as changing the definition of the “default” shortcut.

Some of the more commonly used shortcuts are described below:

## 5.4 Setup Variables and Preprocessing Config Files

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

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

sets the variable “Foo” to string “Bar” and “Baz” to boolean `True`<sup>3</sup>. `setup` can conditionally include and exclude parts of the `Config` file it reads based on the values of these variables. For example, the `IO/IOMain/hdf5/Config` file contains

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

Through judicious use of setup variables, the user can ensure that specific implementations are included or the simulation is properly configured. For example, the setup line `./setup -a Sod +ug` expands to `./setup -a Sod -unit=Grid/GridMain/ Grid=UG`. The relevant part of the `Grid/GridMain/Config` file is given below:

<sup>3</sup>All non-integral values not equal to `True/False/Yes/No/On/Off` are considered to be string values



Table 5.5: Shortcuts for often-used options

Shortcut	Description
+cartesian	use cartesian geometry
+cylindrical	use cylindrical geometry
+noio	omit IO
+nolog	omit logging
+pm4dev	use the PARAMESH4DEV grid
+polar	use polar geometry
+spherical	use spherical geometry
+ug	use the uniform grid in a fixed block size mode
+nofbs	use the uniform grid in a non-fixed block size mode
+usm	use the Unsplit Staggered Mesh MHD solver
+8wave	use the 8-wave MHD solver
+splitHydro	use a split Hydro solver
+vof	use the volume-of-fluid interface capturing in hydro

Table 5.6: Shortcuts for HEDP options

Shortcut	Description
+mtmmmt	Use the 3-T, multimaterial, multitype EOS
+uhd3t	Use the 3-T version of Unsplit Hydro
+usm3t	Use the 3-T version of Unsplit Staggered Mesh MHD
+mgd	Use Multigroup Radiation Diffusion and Opacities
+laser	Use the Laser Ray Trace package

```

DEFAULT serial

USESETUPVARS parallelIO

IF parallelIO
    DEFAULT parallel
ENDIF

```

```

# Requires use of the Grid SetupVariable
USESETUPVARS Grid

DEFAULT paramesh

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

```

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

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

## 5.5 Config Files

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

### 5.5.1 Configuration file syntax

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

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

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

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

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

```
##python:genLines

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

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

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

### 5.5.2 Configuration directives

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

- **# comment**  
A comment. Can appear as a separate line or at the end of a line.
- **DEFAULT *sub-unit***  
Every unit and sub-unit designates one implementation to be the “default”, as defined by the keyword **DEFAULT** in its **Config** file. If no specific implementation of the unit or its sub-units is selected by

the application, the designated default implementation gets included. For example, the `Config` file for the `EosMain` specifies `Gamma` as the default. If no specific implementation is explicitly included (*i.e.*, `INCLUDE physics/Eos/EosMain/Multigamma`), then this command instructs `setup` to include the `Gamma` implementation, as though `INCLUDE physics/Eos/EosMain/Gamma` had been placed in the `Units` file.

- **EXCLUSIVE** *implementation...*  
Specifies a list of implementations that cannot be included together. If no **EXCLUSIVE** instruction is given, it is perfectly legal to simultaneously include more than one implementation in the code. Using “**EXCLUSIVE \***” means that at most one implementation can be included.
- **CONFLICTS** *unit1[/sub-unit[/implementation...]] ...*  
Specifies that the current unit, sub-unit, or specific implementation is not compatible with the list of units, sub-units or other implementations that follows. `setup` issues an error if the user attempts a conflicting unit configuration.
- **REQUIRES** *unit[/sub-unit[/implementation...]] [ OR unit[/sub-unit...]]...*  
Specifies a unit requirement. Unit requirements can be general, without asking for a specific implementation, so that unit dependencies are not tied to particular algorithms. For example, the statement **REQUIRES** `physics/Eos` in a unit’s `Config` file indicates to `setup` that the `physics/Eos` unit is needed, but no particular equation of state is specified. As long as an `Eos` implementation is included, the dependency will be satisfied. More specific dependencies can be indicated by explicitly asking for an implementation. For example, if there are multiple species in a simulation, the `Multigamma` equation of state is the only valid option. To ask for it explicitly, use **REQUIRES** `physics/Eos/EosMain/Multigamma`. Giving a complete set of unit requirements is helpful, because `setup` uses them to generate the units file when invoked with the `-auto` option.
- **REQUESTS** *unit[/sub-unit[/implementation...]]*  
Requests that a unit be added to the Simulation. All requests are upgraded to a “**REQUIRES**” if they are not negated by a “`-without-unit`” option from the command line. If negated, the **REQUEST** is ignored. This can be used to turn off profilers and other “optional” units which are included by default.
- **SUGGEST** *unitname unitname ...*  
Unlike **REQUIRES**, this keyword suggests that the current unit be used along with one of the specified units. The setup script will print details of the suggestions which have been ignored. This is useful in catching inadvertently omitted units before the run starts, thus avoiding a waste of computing resources.
- **PARAMETER** *name type [CONSTANT] default [range-spec]*  
Specifies a runtime parameter. Parameter names are unique up to 20 characters and may not contain spaces. Admissible types include **REAL**, **INTEGER**, **STRING**, and **BOOLEAN**. Default values for **REAL** and **INTEGER** parameters must be valid numbers, or the compilation will fail. Default **STRING** values must be enclosed in double quotes (“). Default **BOOLEAN** values must be `.true.` or `.false.` to avoid compilation errors. Once defined, runtime parameters are available to the entire code. Optionally, any parameter may be specified with the **CONSTANT** attribute (*e.g.*, **PARAMETER** `foo REAL CONSTANT 2.2`). If a user attempts to set a constant parameter via the runtime parameter file, an error will occur.

The range specification is optional and can be used to specify valid ranges for the parameters. The range specification is allowed only for **REAL**, **INTEGER**, **STRING** variables and must be enclosed in ‘[]’.

For a **STRING** variable, the range specification is a comma-separated list of strings (enclosed in quotes). For a **INTEGER**, **REAL** variable, the range specification is a comma-separated list of (closed) intervals specified by `min ... max`, where `min` and `max` are the end points of the interval. If `min` or `max` is omitted, it is assumed to be  $-\infty$  and  $+\infty$  respectively. Finally `val` is a shortcut for `val ... val`. For example

```
PARAMETER pres REAL 1.0 [ 0.1 ... 9.9, 25.0 ... ]
PARAMETER coords STRING "polar" ["polar","cylindrical","2d","3d"]
```

indicates that `pres` is a REAL variable which is allowed to take values between 0.1 and 9.9 or above 25.0. Similarly `coords` is a string variable which can take one of the four specified values.

- *D parameter-name comment*

Any line in a `Config` file is considered a parameter comment line if it begins with the token `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,

```
D SOME_PARAMETER The purpose of this parameter is whatever
```

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

```
D SOME_PARAMETER The purpose of this parameter is whatever
D &                This is a second line of description
```

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

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

- *VARIABLE name [TYPE: vartype] [eosmap-spec]*

Registers variable with the framework with name *name* and a variable type defined by *vartype*. The `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 `Flash.h`.

The possible types for *vartype* are as follows:

- `PER_VOLUME`

This solution variable is represented in **conserved** form, *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 `PER_MASS` form instead).

- `PER_MASS`

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

- `GENERIC`

This is the default *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 `TYPE` attribute is only used to determine which variables should be converted to conservative form for certain `Grid` operations that may require interpolation (*i.e.*, prolongation, guardcell filling, and restriction) when one of the runtime parameters

`convertToConsvdForMeshCalls` or `convertToConsvdInMeshInterp` is set `true`. Only variables of 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

- variable types are irrelevant for uniform grids,

- variable 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`.

- **MULTIFLUID** *name* Registers pairs of variables with names *MFA\_name* and *MFB\_name* in the framework, intended to be used with the volume-of-fluid interface capturing scheme. The `setup` script collects multifluid 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 `Flash.h`.
- **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.
- **EDGE** *name* Registers an edge variable for use in the unit that declare it. This maps the the `paramesh nvaredge` variable through the preprocessor variable `FLASH_NEDGE_VAR` in `Flash.h`. The total number of edge variables is taken as the max over all units the total number of requested edge variables from each individual unit.

- **SCRATCHCENTERVAR** *name [eosmap-spec]*

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

- **SCRATCHFACEVAR** *name [eosmap-spec]*

This keyword is used in connection with the grid scope scratch space for face-centered data, it is identical in every other respect to **SCRATCHCENTERVAR**.

- **SCRATCHVAR** *name [eosmap-spec]*

This keyword is used for specifying instances of general purpose grid scope scratch space. The same space can support cell-centered as well as face-centered data. Like other scratch data structures, the variables in this data structure can also be asked with “name” and do not participate in guardcell filling.

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

- **MASS\_SCALAR** *name [RENORM: group-name] [eosmap-spec]*

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

For *eosmap-spec*, see above under **VARIABLE**. An *eosmap-spec* for a **MASS\_SCALAR** may be used in an **Eos.wrapped** invocation when the optional **gridDataStruct** argument is absent or has a value of **CENTER**.

### Avoid Confusion!

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

- **PARTICLETYPE** *particle-type INITMETHOD initialization-method MAPMETHOD map-method ADVMETHOD time-advance-method*

This keyword associates a **particle type** with mapping and initialization sub-units of **Particles** unit to operate on this particle type during the simulation. Here, *map-method* describes the method used to map the particle properties to and from the mesh (see [Section 21.2](#)), *initialization-method* describes the method used to distribute the particles at initialization, and *time-advance-method* describes the method used to advance the associated particle type in time (see [Section 21.1](#), and in general [Section 21.3](#)). This keyword has been introduced to facilitate inclusion of multiple particle types in the same simulation. It imposes certain requirements on the use of the **ParticlesMapping** and **ParticlesInitialization** subunits. **Particles** (of any type, whether called **passive** or anything else) do not have default methods for initialization, mapping, or time integration, so a **PARTICLETYPE** directive in a **Config** file (or an equivalent **-particlemethods=** setup option, see [Table 5.4](#)) is the only way to specify the appropriate implementations of the **Particles** subunits to be used. The declaration should be accompanied by appropriate “REQUESTS” or “REQUIRES” directives to specify the paths of the appropriate subunit

implementation directories to be included. For clarity, our technique has been to include this information in the simulation directory **Config** files only. All the currently available mapping and initialization methods have a corresponding identifier in the form of preprocessor definition in **Particles.h**. The user may select any *particle-type* name, but the *map-method*, *initialization-method* and *time-advance-method* must correspond to existing identifiers defined in **Particles.h**. This is necessary to navigate the data structure that stores the particle type and its associated mapping and initialization methods. Users desirous of adding new methods for mapping or initialization should also update the **Particles.h** file with additional identifiers and their preprocessor definitions. Note, it is possible to use the same methods for different particle types, but each particle type name must only appear once. Finally, the Simulations **Config** file is also expected to request appropriate implementations of mapping and initialization subunits using the keyword **REQUESTS**, since the corresponding Config files do not specify a default implementation to include. For example, to include **passive** particle types with **Quadratic** mapping, **Lattice** initialization, and **Euler** for advancing in time the following code segment should appear in the Config file of the Simulations directory.

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

- **PARTICLEPROP** *name type*

This keyword indicates that the particles data structure will allocate space for a sub-variable “NAME.-PART\_PROP.” For example if the Config file contains

```
PARTICLEPROP dens
```

then the code can directly access this property as

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

*type* may be **REAL** or **INT**, however **INT** is presently unused. See [Section 6.6](#) for more information and examples.

- **PARTICLEMAP** **TO** *partname* **FROM** *vartype varname*

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

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

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

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

Additional information on creating Config files for particles is obtained in [Section 21.3.2](#).

- **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* [*target*]

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](#).

*target* is used when building with `cmake` to specify a target that will be linked to the FLASH executable.

- **LINKIF** *filename* *unitname*

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

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

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

```
LINKIF Grid_markRefineDerefine.F90 Grid/GridMain/paramesh
```

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

- **PPDEFINE** *sym1 sym2 ...*

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

```
#ifdef FLASH_GRID_UG
    ug specific code
#endif

#ifdef FLASH_GRID_PARAMESH3OR4
    pm3+ specific code
#endif
```

By convention, many preprocessor symbols defined in Config files included in the FLASH code distribution start with the prefix “FLASH\_”.

- **USESETUPVARS** *var1, var2, ...*

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

- **CHILDORDER** *child1 child2 ...*

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

- **GUARDCELLS** *num*

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

- **SETUPERROR** *error message*

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

- **IF, ELSEIF, ELSE, ENDIF**

A conditional block is of the following form:

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

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

- **NONREP** *unktype name localmax globalparam ioformat*

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

- *unktype*: must be either **MASS\_SCALAR** or **VARIABLE**
- *name*: the name of this variable array. It is suggested that it be all capital letters, and must conform to what the C preprocessor will consider as a valid symbol for use in a **#define** statement.
- *localmax*: a positive integer specifying the maximum number of elements from the global variable array a mesh can hold. This is the actual number of UNK variables that are allocated on each processor, though not all of them will necessarily be used.
- *globalparam*: the name of a runtime parameter which dictates the size of this global array of variables.
- *ioformat*: a string representing how the elements of the array will be named when written to the output files. The question mark character **?** is used as a placeholder for the digits of the array index. As an example, the format string **x???** will generate the dataset names **x001**, **x002**, **x003**, etc. This string must be no more than four characters in length.

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

$$globalparam \leq meshCopyCount * localmax$$

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

Example:

Config file:

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

flash.par file:

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

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

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

## 5.6 Creating a Site-specific Makefile

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

- Create the directory `sites/<hostname>`, where `<hostname>` is the hostname of your machine.
- Start by copying `os/<your os>/Makefile.h` to `sites/<hostname>`
- Use `bin/suggestMakefile.sh` to help identify the locations of various libraries on your system. The script scans your system and displays the locations of some libraries. You must note the location of MPI library as well. If your compiler is actually an mpi-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](#).

### Compilation warning

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

### HDF5 Library Interface

If you are using HDF5 for I/O, and your HDF5 library version is greater than 1.6.x, the C source files of the I/O unit need to be compiled with the preprocessor symbol `H5_USE_16_API` defined. This is usually best achieved by having a line like the following in your `Makefile.h`:

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

## 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.

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

### 5.7.2 Code generated by the setup call

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

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

### 5.7.3 Makefiles generated by setup

Apart from the master `Makefile`, `setup` generates a makefile for each unit, which is “included” in the master `Makefile`. This is true even if the unit is not included in the application. These unit makefiles are named `Makefile.Unit` and are a concatenation of all the `Makefiles` found in unit hierarchy processed by `setup`.

For example, if an application uses `Grid/GridMain/paramesh/paramesh4/Paramesh4.0`, the file `Makefile.Grid` will be a concatenation of the `Makefiles` found in

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

As another example, if an application does not use `PhysicalConstants`, then `Makefile.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.

## 5.8 Setup a hybrid MPI+OpenMP FLASH application

There is the experimental inclusion of FLASH multithreading with OpenMP in the FLASH4 beta release. The units which have support for multithreading are split hydrodynamics 15.1.2, unsplit hydrodynamics 15.1.3, Gamma law and multigamma EOS 17.2, Helmholtz EOS 17.3, Multipole Poisson solver (improved version (support for 2D cylindrical and 3D cartesian)) 8.10.2.2 and energy deposition 18.4.

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

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

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

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

### Makefile.h variables

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

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

```
OPENMP = -fopenmp
```

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

```
OPENMP_FORTRAN = --openmp -Kpureomp
OPENMP_C = -fopenmp
OPENMP_LINK = --openmp -Kpureomp
```

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

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

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

Log file stamp	safe	unsafe (1)	unsafe (2)	unsafe (3)
Number of MPI tasks:	1	1	1	1
MPI version:	2	1	2	2
MPI subversion:	2	2	1	2
MPI thread support:	T	F	F	F
OpenMP threads/MPI task:	2	2	2	2
OpenMP version:	200805	200505	200505	200805
Is “_OPENMP” macro defined:	T	T	T	F

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

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

The FLASH applications in Table 5.7 are unsafe because

1. we are using an MPI-1 implementation.
2. we are using an MPI-2 implementation which is not built with thread support - the “MPI thread support in OpenMPI” Flash tip may help.
3. we are using a compiler that does not define the macro `_OPENMP` when it compiles source files with OpenMP support (see OpenMP standard). I have noticed that Absoft 64-bit Pro Fortran 11.1.3 for Linux x86\_64 does not define this macro. We use this macro in `Driver_initParallel.F90` to conditionally initialize MPI with `MPI_Init_thread`. If you find that `_OPENMP` is not defined you should define it in your `Makefile.h` in a manner similar to the following:

```
OPENMP_FORTRAN = -openmp -D_OPENMP=200805
```

### MPI thread support in OpenMPI

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

### MPI-IO issues when using a threaded FLASH application

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

You should not setup a FLASH application with both `threadBlockList` and `threadWithinBlock` equal to `True` - nested OpenMP parallelism is not supported. For further information about FLASH multithreaded applications please refer to Chapter 43.



## 5.9 Setup a FLASH+Chombo application

### 5.9.1 Overview

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

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

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

The shortcuts are used in FLASH setup lines as follows:

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

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

### 5.9.2 Build procedure

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

- C++ compiler
- Fortran compiler with C interoperability features
- MPI build of Chombo
- parallel HDF5

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

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

```

We expect to pick up the complete set of integer values
from 0 to 23 in contiguous memory locations
Fortran array element:  1  1  1  1.  Expected:    0, actual:    0.
Fortran array element:  1  2  1  1.  Expected:    1, actual:    1.
Fortran array element:  1  1  2  1.  Expected:    2, actual:    2.
Fortran array element:  1  2  2  1.  Expected:    3, actual:    3.
Fortran array element:  1  1  3  1.  Expected:    4, actual:    4.
Fortran array element:  1  2  3  1.  Expected:    5, actual:    5.
Fortran array element:  1  1  1  2.  Expected:    6, actual:    3.
Fortran array element:  1  2  1  2.  Expected:    7, actual:    4.
Fortran array element:  1  1  2  2.  Expected:    8, actual:    5.
Fortran array element:  1  2  2  2.  Expected:    9, actual:    6.
Fortran array element:  1  1  3  2.  Expected:   10, actual:    7.
Fortran array element:  1  2  3  2.  Expected:   11, actual:    8.
Fortran array element:  1  1  1  3.  Expected:   12, actual:    6.
Fortran array element:  1  2  1  3.  Expected:   13, actual:    7.
Fortran array element:  1  1  2  3.  Expected:   14, actual:    8.
Fortran array element:  1  2  2  3.  Expected:   15, actual:    9.
Fortran array element:  1  1  3  3.  Expected:   16, actual:   10.
Fortran array element:  1  2  3  3.  Expected:   17, actual:   11.
Fortran array element:  1  1  1  4.  Expected:   18, actual:    9.
Fortran array element:  1  2  1  4.  Expected:   19, actual:   10.
Fortran array element:  1  1  2  4.  Expected:   20, actual:   11.
Fortran array element:  1  2  2  4.  Expected:   21, actual:   12.
Fortran array element:  1  1  3  4.  Expected:   22, actual:   13.
Fortran array element:  1  2  3  4.  Expected:   23, actual:   14.
FAILURE!  Picked up unexpected C++ assigned values

```

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

```

CXX          = icpc
FC           = ifort
MPI          = TRUE
MPICXX       = /home/cdaley/software/mpich2/1.3.1/intel/bin/mpicxx
USE_64       = TRUE
USE_HDF      = TRUE
HDFINCFLAGS  = -I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include -DH5_USE_16_API
HDFLIBFLAGS  = -L/home/cdaley/software/hdf5/1.8.5-patch1/intel/lib -lhdf5
HDFMPIINCFLAGS = -I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include -DH5_USE_16_API
HDFMPILIBFLAGS = -L/home/cdaley/software/hdf5/1.8.5-patch1/intel/lib -lhdf5
USE_MT       = FALSE
syslibflags  = -lstdc++ -lz

```

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

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

In contrast to Paramesh, the Chombo source code is not included in the FLASH source tree. This means that you, the user, must manually download and then build Chombo before building FLASH applications that make use of Chombo. You should download version 3.0 or 3.1 of Chombo from

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

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

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

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

Only after you have successfully built and run the Chombo unit tests described in their user guide should you have any expectation that FLASH will work with Chombo library. For your reference we show the `Make.defs.local` macro definitions that we have used to build Chombo on a x86\_64 machine in a FLASH-compatible way in Figure 5.4 .

### Compiling Chombo on IBM BG/P

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

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

to

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

At this stage you can optionally build and run the unit test `source/Grid/GridMain/Chombo/wrapper/-unit_tests/2`. This is a standalone Fortran MPI application that uses Chombo library to create a distributed mesh over 4 MPI processes. It can be built by entering this directory and then executing the command `make all`. We expect that this program will be helpful for debugging mixed-language programming issues between FLASH and Chombo (especially on untested architectures / compiler combinations) independently of a full-blown FLASH application. Note that you will need to edit the Makefile appropriately for your installation of Chombo.

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

```
CFLAGS_CHOMBO: -I/home/cdaley/flash/chomboForFlash/current/lib/include
-DCH_SPACEDIM=2 -DCH_Linux -DCH_IFC -DCH_MPI -DMPICH_SKIP_MPICXX
-ULAM_WANT_MPI2CPP -DMPI_NO_CPPBIND -DCH_USE_SETVAL -DCH_USE_COMPLEX
-DCH_USE_64 -DCH_USE_DOUBLE -DCH_USE_HDF5
-I/home/cdaley/software/hdf5/1.8.5-patch1/intel/include
-DH5_USE_16_API -DCH_FORT_UNDESCORE
-I/home/cdaley/flash/chomboForFlash/current/lib/src/BaseTools
-DCH_LANG_CC

LIB_CHOMBO: -L/home/cdaley/flash/chomboForFlash/current/lib
-lamrtimedependent2d.Linux.64.mpicxx.ifort.DEBUG.MPI
-lamrtools2d.Linux.64.mpicxx.ifort.DEBUG.MPI
-lboxtools2d.Linux.64.mpicxx.ifort.DEBUG.MPI
-lbasetools2d.Linux.64.mpicxx.ifort.DEBUG.MPI -lstdc++
```

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

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

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

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

You should now be ready to setup and build FLASH applications with Chombo mesh package.

## 5.10 Setup a FLASH application with cmake

FLASH4.8 introduces the capability to configure a FLASH application using the `cmake` build tool, enabled by setting up with the option `-cmake`. In analogy to the site-specific `Makefile.h`, when using `cmake setup` will look for a `site.cmake` in `sites/<hostname>/site.cmake` that provides some site specific configuration information. Minimally a user needs to specify the various compilers and flags.

Libraries are searched using `cmake's find_package` to find external libraries and link against targets defined in the requested `LIBRARY` Config directives. For libraries that don't have a `FindLIBNAME` command provided by `cmake`, they can be added to the directory `tools/cmake`, which gets added to the

```

CHOMBO_PATH = /home/cdaley/flash/chomboForFlash/current

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

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

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

_MPI := \$(shell make vars DIM=\${_DIM} -C \${CHOMBO_PATH}/lib | \
    awk -F 'MPICXX=' '/^MPICXX/{print \$\$2}')

ifeq (\$(strip \${_MPI}),)
    \$(error "Chombo MPICXX variable is empty")
endif

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

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

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

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

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

```

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

```
# compiler specification
set(CMAKE_Fortran_COMPILER mpif90)
set(CMAKE_C_COMPILER      mpicc)
set(CMAKE_CXX_COMPILER    mpicxx)
# compiler flags
set(FFLAGS "-fdefault-real-8 -fdefault-double-8")
set(CFLAGS "")
set(CXXFLAGS "")
# hints that wrap some of the common cmake FindLIB commands
set(PYBIND11_DIR "path/to/pybind11")
set(HDF5_DIR "path/to/hdf5")
set(HYPRE_DIR "path/to/hypre")
```

Figure 5.6: cmake compiler specification.

```
# commonly a build directory is made inside the object directory
cd object
mkdir build
cd build
cmake ..
make
# FLASH may be built in any directory by pointing to the object directory
cmake /path/to/object
make
```

Figure 5.7: building a FLASH application with `cmake`

`CMAKE_MODULE_PATH` by the `setup` generated `CMakeLists.txt`. When possible some of the commonly used libraries in FLASH have wrappers in `tools/cmake` around the `cmake` provided commands that accept hints for the library's location that follow the `LIBNAME_DIR` format convention.

The FLASH application can be built by pointing `cmake` to the desired object directory

# Chapter 6

## The Flash.h file

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

### 6.1 UNK, FACE(XYZ) Dimensions

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

**NXB,NYB,NZB**

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

**MAXCELLS**

The maximum of (NXB,NYB,NZB)

**MAXBLOCKS**

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

**GRID\_(IJK)LO**

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

**GRID\_(IJK)HI**

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

**GRID\_(IJK)LO\_GC**

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

**GRID\_(IJK)HI\_GC**

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

**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<sup>1</sup>. However, in `FLASH4` the user does not need to be intimately aware of the `unk` array layout, as starting and ending indices of these groups of quantities are defined in `Flash.h`. The following pre-processor symbols define the indices of the various quantities related to a given cell. These symbols are primarily used to perform some computation with all property variables, species mass fractions, or all mass scalars.

**NPROP\_VARS**

The number of property variables in the simulation

**NSPECIES**

The total number of species in the simulation

**NMASS\_SCALARS**

The number of mass scalars in the simulation

**NUNK\_VARS**

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

**PROP\_VARS\_BEGIN,PROP\_VARS\_END**

The indices in the `unk` array used for property variable data

**SPECIES\_BEGIN,SPECIES\_END**

The indices in the `unk` array used for species data

**MASS\_SCALARS\_BEGIN,MASS\_SCALARS\_END**

The indices in the `unk` array used for mass scalars data

**UNK\_VARS\_BEGIN,UNK\_VARS\_END**

The low and high indices for the `unk` array

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

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

<sup>1</sup>See [\(15.9\)](#) for more information about mass scalars



a variable “temp” to hold temperature and a mass scalar “temp” indicating a temporary variable. Although the `Flash.h` file can distinguish between these two types of variables, many plotting routines such as `fidlr3.0` cannot.

## 6.3 Fluxes

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

`NPROP_FLUX`

The number of property variables in the simulation

`NSPECIES_FLUX`

The total number of species in the simulation

`NMASS_SCALARS_FLUX`

The number of mass scalars in the simulation

`NFLUXES`

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

`PROP_FLUX_BEGIN, PROP_FLUX_END`

The indices in the `fluxes` data structure used for property variable data

`SPECIES_FLUX_BEGIN, SPECIES_FLUX_END`

The indices in the `fluxes` data structure used for species data

`MASS_SCALARS_FLUX_BEGIN, MASS_SCALARS_FLUX_END`

The indices in the `fluxes` data structure used for mass scalars data

`FLUXES_BEGIN`

The first index for the `fluxes` data structure

## 6.4 Scratch Vars

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

are defined to hold the number and endpoints of the cell-centered scratch variables. For face-centered scratch variables the CENTER is in the above terms is replaced with FACEX/Y/Z while for SCRATCHVARS, the CENTER is replace with GRID.

## 6.5 Fluid Variables Example

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

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

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

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

SCRATCHVAR otmp
SCRATCHCENTERVAR ftmp
.....
```

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

```
#define DENS_VAR 1
#define EINT_VAR 2
#define ENER_VAR 3
#define PRES_VAR 4
#define TEMP_VAR 5
#define VELX_VAR 6
#define VELY_VAR 7
#define VELZ_VAR 8

#define E_FLUX 1
#define EINT_FLUX 2
#define P_FLUX 3
#define RHO_FLUX 4
#define U_FLUX 5
```

```
#define UT_FLUX 6
#define UTT_FLUX 7

#define OTMP_SCRATCH_GRID_VAR 1

#define FTMP_SCRATCH_CENTER_VAR 1
```

## 6.6 Particles

### 6.6.1 Particles Types

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

`Config` file :

```
PARTICLETYPE passive INITMETHOD lattice MAPMETHOD quadratic ADVMETHOD rungekutta

REQUIRES Particles/ParticlesMain
REQUESTS Particles/ParticlesMain/passive/RungeKutta
REQUESTS Particles/ParticlesMapping/Quadratic
REQUESTS Particles/ParticlesInitialization/Lattice
REQUESTS IO/IOMain/
REQUESTS IO/IOParticles
```

`Flash.h` :

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

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

### 6.6.2 Particles Properties

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

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

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

```

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

```

## 6.7 Non-Replicated Variable Arrays

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

### 6.7.1 Per-Array Macros

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

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

the following will be placed into `Flash.h`:

- **#define FOO\_NONREP**  
An integer constant greater than zero to be used as an id for this array. The user should rarely ever need to use this value. Instead, it is much more likely that the user would just query the preprocessor for the existence of this symbol (via `#ifdef FOO_NONREP`) to enable sections of code.
- **#define FOO\_NONREP\_LOC2UNK(*loc*)**  
Given the 1-based index into this mesh's local subset of the global array, returns the `UNK` index where that variable is stored.
- **#define FOO\_NONREP\_MAXLOCS**  
The integer constant *localmax* as supplied in the `Config` file `NONREP` line.
- **#define FOO\_NONREP\_RPCOUNT**  
The string literal value of *globalparam* from the `NONREP` line.

### 6.7.2 Array Partitioning Macros

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

- **#define NONREP\_NLOCS(*mesh*,*meshes*,*globs*)**  
Returns the number of elements in the mesh's local subset. This value will always be less than or equal to `FOO_NONREP_MAXLOCS`.
- **#define NONREP\_LOC2GLOB(*loc*,*mesh*,*meshes*)**  
Maps an index into a mesh's local subset to its index in the global array.
- **#define NONREP\_GLOB2LOC(*glob*,*mesh*,*meshes*)**  
Maps an index of the global array to the index into a mesh's local subset. If the mesh provided does not have that global element the result is undefined.
- **#define NONREP\_MESHOFGLOB(*glob*,*meshes*)**  
Returns the mesh that owns a given global array index.

Descriptions of arguments to the above macros:

- *loc*: 1-based index into the mesh-local subset of the global array's indices.

- **glob:** 1-based index into the global array.
- **globs:** Length of the global array. This should be the value read from the array's runtime parameter `FOO_NONREP_RPCOUNT`.
- **mesh:** 0-based index of the mesh in question. For a processor, this is its rank in the `MESH_ACROSS_COMM` communicator.
- **meshes:** The number of meshes. This should be the value of the runtime parameter `meshCopyCount`, or equivalently the number of processors in the `MESH_ACROSS_COMM` communicator.

### 6.7.3 Example

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

Config:

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

flash.par:

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

Fortran 90:

```
\#include "Flash.h"\\
! this shows how to iterate over the UNKs corresponding to this
! processor's subset of FOO
integer :: nfooglob, nfooloc ! number of foo vars, global and local
integer :: mesh, nmesh ! this mesh number, and total number of meshes
integer :: fooloc, fooglob ! local and global foo indices
integer :: unk\\
call Driver_getMype(MESH_ACROSS_COMM, mesh)
call Driver_getNumProcs(MESH_ACROSS_COMM, nmesh)
call RuntimeParameters_get(FOO_NONREP_RPCOUNT, nfooglob)
nfooloc = NONREP_NLOCS(mesh, nmesh, nfooglob)\\
! iterate over the local subset for this mesh
do fooloc=1, nfooloc
  ! get the location in UNK
  unk = FOO_NONREP_LOC2UNK(fooloc)
  ! get the global index
  fooglob = NONREP_LOC2GLOB(fooloc, mesh, nmesh)\\
  ! you have what you need, do your worst...
end do
```

## 6.8 Other Preprocessor Symbols

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

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

# **Part III**

## **Driver Unit**





## Chapter 7

# Driver Unit

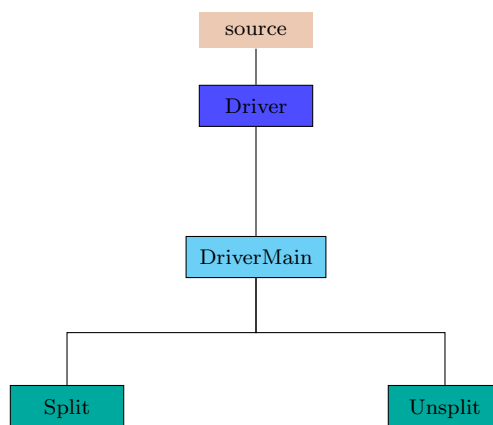


Figure 7.1: The **Driver** unit directory tree.

The **Driver** unit controls the initialization and evolution of FLASH simulations. In addition, at the highest level, the **Driver** unit organizes the interaction between units. Initialization can be from scratch or from a stored checkpoint file. For advancing the solution, the drivers can use either an operator-splitting technique adapted to directionally split physics operators like split Hydro (**Split**), or a more generic “**Unsplit**” implementation. The **Driver** unit also calls the IO unit at the end of every timestep to produce checkpoint files, plot files, or other output.

### 7.1 Driver Routines

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

```
program Flash
```

```
    implicit none
```

```
    call Driver_initParallel()
```

```
    call Driver_initFlash()
```

```
    call Driver_evolveFlash( )
```

```
    call Driver_finalizeFlash ( )
```

end program Flash

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 implementations for most of these files are located in the **Driver/DriverMain** unit implementation directory, with a few specialized ones in either **Driver/DriverMain/Split** or **Driver/DriverMain/Unsplit**.

### 7.1.1 Driver\_initFlash

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

### 7.1.2 Driver\_evolveFlash

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

#### 7.1.2.1 Strang Split Evolution

The code in the **Driver/DriverMain/Split** unit implementation directory has been the default time update method up to FLASH4.3, and can still be used for many setups that can be configured with FLASH. The routine **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 physics and other units in each time step goes something like this: **Hydro**, diffusive terms, source terms, **Particles**, **Gravity**; **Hydro**, diffusive terms, source terms, **Particles**, **Gravity**, **IO** (for output), **Grid** (for grid changes). The hydrodynamics update routines take a “sweep order” argument since they must be directionally split to work with this driver. Here, the first call usually 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** directory is the default since FLASH4.4. It is required specifically for the two unsplit solvers: unsplit staggered mesh MHD solver ([Section 15.3.3](#)) and the unsplit gas hydrodynamics solver ([Section 15.1.3](#)). This implementation in general calls each of the

physics routines only once per time step, and each call advances solution vectors by one timestep. At the end of one loop of timestep advancement, the condition for updating the adaptive mesh refinement pattern is tested and applied.

### 7.1.2.3 Super-Time-Stepping (STS)

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

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

$$\tau_i = \Delta t_{\text{CFL\_para}} \left[ (-1 + \nu_{\text{sts}}) \cos\left(\frac{\pi(2j-1)}{2N_{\text{sts}}} + 1 + \nu_{\text{sts}}\right) \right]^{-1}, \quad (7.1)$$

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

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

$$\tau_i = \Delta t_{\text{CFL}} \left[ (-1 + \nu_{\text{sts}}) \cos\left(\frac{\pi(2j-1)}{2N_{\text{sts}}} + 1 + \nu_{\text{sts}}\right) \right]^{-1}, \quad (7.2)$$

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

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

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

### 7.1.2.4 Runtime Parameters

The **Driver** unit supplies certain runtime parameters regardless of which type of driver is chosen. These are described in the online [Runtime Parameters Documentation page](#).

Table 7.1: Runtime parameters for STS

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

### FLASH Transition

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

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

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

#### 7.1.3 Driver\_finalizeFlash

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

#### 7.1.4 Driver accessor functions

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

### FLASH Transition

In FLASH4 most of the quantities that were in the FLASH2 database are stored in the `Grid` unit or are replaced with functionality in the `Flash.h` file. A few scalar 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`.

#### 7.1.5 Time Step Limiting

The `Driver` unit is responsible for determining the time step  $\Delta t$  that is used to advance the solution from time  $t = t^{n-1}$  to  $t^n$ . At startup, a tentative  $\Delta t$  is chosen based in runtime parameters, in particular `dtinit`. The routine `Driver_verifyInitDt` (usually invoked from `Driver_initFlash`) is used to check and, if necessary, modify the initial  $\Delta t$ . Subsequently, the routine `Driver_computeDt` (usually invoked from `Driver_evolveFlash` at the end of an iteration of the main evolution loop) is used to recompute  $\Delta t$  for the next evolution step.

The implementation of `Driver_computeDt` can lower or increase the time step. It takes various runtime parameters into consideration, including `dtmin`, `dtmax`, and `tstep-change-factor`. For the most part, however, `Driver_computeDt` calls on `UNIT_computeDt` routines of various code units to query those units for their time step requirements, and usually chooses the smallest time step that is acceptable to all units queried.

The code units that participate in this negotiation return a  $\Delta t$ , and usually some additional information about which location in the simulation domain caused the required step to be as low as returned. A unit's time step requirement can depend on the current state of the solution as well as on further runtime parameters. For example, the `dt` returned by `Hydro_computeDt` depends on the state of density, pressure, and velocities (and possibly additional variables) in the cells of the domain, as well as on the runtime parameter `cfl`.

##### 7.1.5.1 Generic time step limiting for positive definiteness

A generic method for limiting time steps, based on a requirement that certain (user-specified) variables should never become negative, has been added in FLASH4.5. To understand the basic idea, think of each variable that this limiter is applied to as a density of some quantity  $q$ . (Variables of `PER_MASS` type are actually converted to their `PER_VOLUME` counterparts in the implementation of the method.)

The algorithm is based on examining the distribution of  $q$  in neighboring cells, and making for each cell a near-worst-case estimate of the rate of depletion  $\dot{q}$  based on projected fluxes of  $q$  out of the cell over its faces. This can be applied to any variable, it is not required that the variable represents any quantity that is actually advected as described. See [Table 7.2](#) for how to use. This feature may be particularly useful when applied to "eion" in multidimensional 3T simulations in order to avoid some kinds of "negative ion temperature" failures, as an alternative to lowering the `Hydro` runtime parameter `cfl`.

Table 7.2: Runtime parameters for positive definiteness.

Parameter	Type	Default	Description
<code>dr_usePosdefComputeDt</code>	boolean	<code>.false.</code>	Set to <code>.true.</code> to turn positive-definite time step limiter on.
<code>dr_numPosdefVars</code>	integer	4	Number of variables for which positive definiteness should be enforced
<code>dr_posdefVar_N</code>	string	<code>"none"</code>	Name of Nth variable for positive-definite dt limiter
<code>dr_posdefDtFactor</code>	real	1.0	Scaling factor for dt limit from positive-definite time step limiter. Similar to CFL factor. If set to -1, use <code>CFL</code> factor from <code>Hydro</code> .

**Part IV**

**Infrastructure Units**





## Chapter 8

# Grid Unit

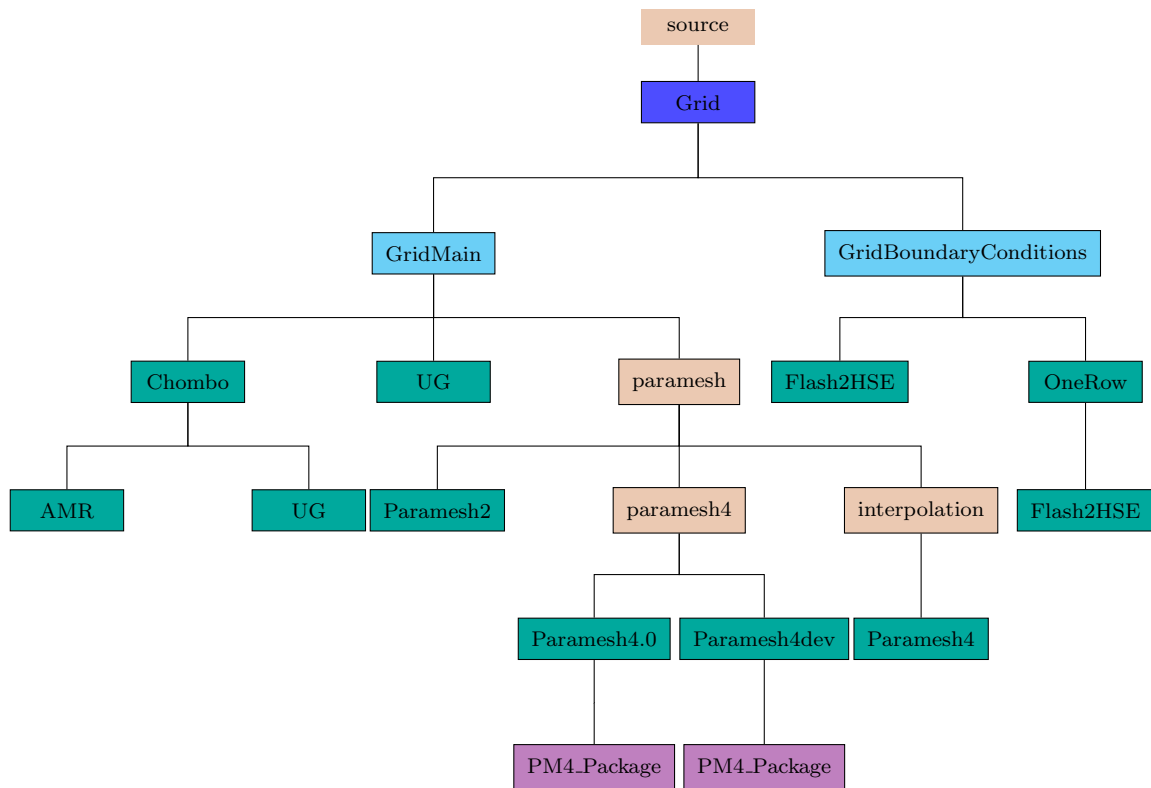


Figure 8.1: The **Grid** unit: structure of **GridMain** and **GridBoundaryCondition** subunits.

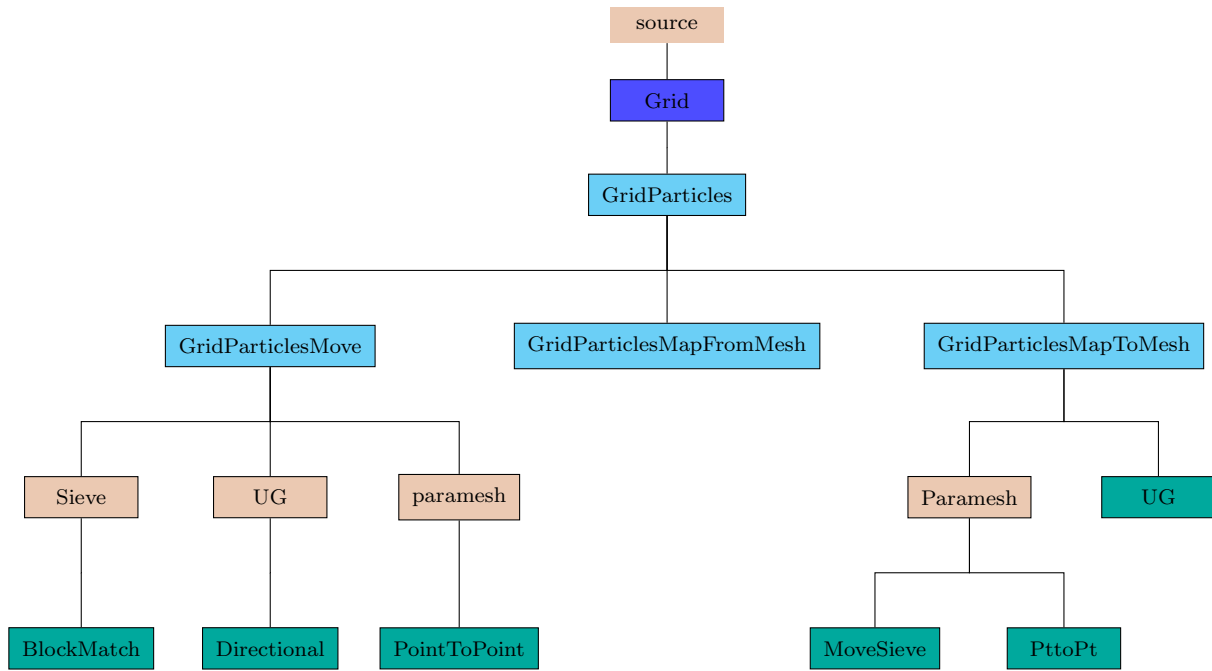
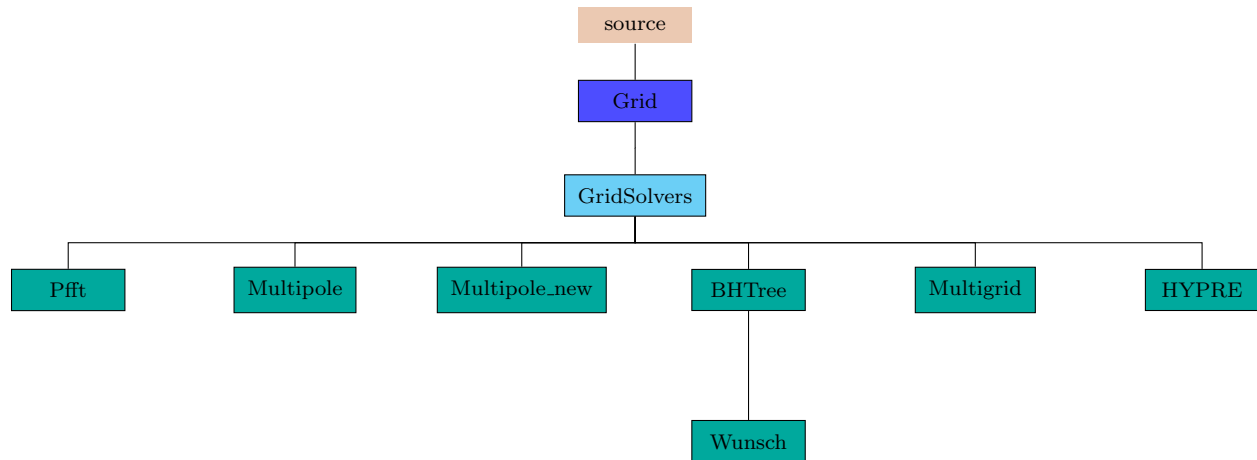


Figure 8.2: The `Grid` unit: structure of `GridParticles` subunit.

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 (`nxb` in the  $x$ -direction, `nyb` in the  $y$ -direction, and `nzb` in the  $z$ -direction). A perimeter of guardcells, of width `nguard` cells in each coordinate direction, surrounds each block of local data, providing it with data from the neighboring blocks or with boundary conditions, as shown in Figure 8.4. Since the majority of physics solvers used in FLASH are explicit, a block with its surrounding guard cells becomes a self-contained computational domain. Thus the physics units see and operate on only one block at a time, and this abstraction is reflected in their design.

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

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

FLASH can interchangeably use either a **uniform** or **adaptive grid** for most problems. Additionally, a new feature in FLASH4 is an option to replicate the mesh; that is processors are assumed to be partitioned into groups, each group gets a copy of the entire domain mesh. This feature is useful when it is possible to decompose the computation based upon certain compute intensive tasks that apply across the domain. One such example is radiation transfer with multigroup flux limited diffusion where each group needs an implicit solve. Here the state variable of the mesh are replicated on each group of processors, while the groups are unique. Thus at the cost of some memory redundancy, it becomes possible to compute a higher

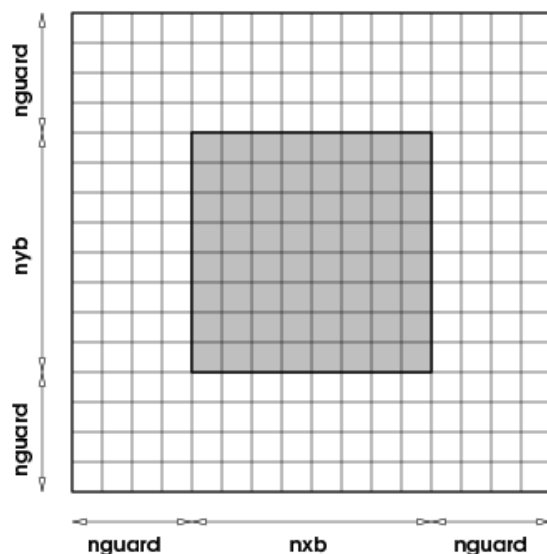


Figure 8.4: A single 2-D block showing the interior cells (shaded) and the perimeter of guard cells.

fidelity problem (see [Chapter 25](#) for an example). Because of this feature, the parallel environment of the simulation is now controlled by the Driver which differentiates between global communicators and mesh communicators. The Grid unit queries the Driver unit for mesh communicators. In all other respects this change is transparent to the Grid unit. Mesh replication can be invoked through the runtime parameter `meshCopyCount`

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

### FLASH 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 FLASH, the user must match up the correct IO alternative implementation with the correct Grid alternative implementation. Please see [Chapter 9](#) for more details. Note that the `setup` script has capabilities to let the user set up shortcuts, such as “+ugio”, which makes sure that the appropriate branch of IO is included when the uniform grid is being used. Please see [Section 5.3](#) for more information. Also see [grid tips](#) for shortcuts useful for the Grid unit.

## 8.2 GridMain Data Structures

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

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

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

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

In addition to the primary physical variables, the **Grid** unit has another set of data structures for storing auxiliary fluid variables. This set of data structures provides a mechanism for storing such variables whose spatial scope is the entire physical domain, but who do not need to maintain their guard cells updated at all times. The data structures in this set include: `SCRATCHCENTERVAR`, which has the same shape as the cell centered variables data structure; and `SCRATCHFACEXVAR`, `SCRATCHFACEYVAR` and `SCRATCHFACEZVAR`, which have the same shape as the corresponding face centered variables data structures. Early releases of

FLASH3 had `SCRATCHVAR`, dimensioned `array(NSCRATCH_GRID_VARS,nxb+1,nyb+1,nzb+1,blocks)`, as the only grid scope scratch data structure. For reasons of backward compatibility, and to maximize reusability of space, `SCRATCHVAR` continues to exist as a supported data structure, though its use is deprecated. The datastructures for face variables, though supported, are not used anywhere in the released code base. The unsplit MHD solver `StaggeredMesh` discussed in [Section 15.3.3](#) gives an example of the use of some of these data structures. It is important to note that there is no guardcell filling for the scratch variables, and the values in the scratch variables become invalid after a grid refinement step. While users can define scratch variables to be written to the plotfiles, they are not by default written to checkpoint files. The `Grid` unit also stores the metadata necessary for work distribution, load balancing, and other housekeeping activities. These activities are further discussed in [Section 8.5](#) and [Section 8.6](#), which describe individual implementations of the `Grid` unit.

## 8.3 Computational Domain

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

### FLASH Transition

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

The physical domain is mapped into a computational domain at problem initialization through routine `Grid_initDomain` in `PARAMESH` and `Chombo`, and `Grid_init` in `UG`. When using the uniform grid `UG`, the mapping is easy: one block is created for each processor in the run, which can be sized either at build time or runtime depending upon the mode of `UG` use.<sup>1</sup> Further description can be found in [Section 8.5](#). When using the AMR grid `PARAMESH`, the mapping is non-trivial. The adaptive mesh `gr_createDomain` function creates an initial mesh of `nblockx * nblocky * nblockz` top level blocks, where `nblockx`, `nblocky`, and `nblockz` are runtime parameters which default to 1.<sup>2</sup> 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.

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

<sup>2</sup>The `gr_createDomain` function also can remove certain blocks of this initial mesh, if this is requested by a non-default `Simulation.defineDomain` implementation.

### FLASH Transition

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

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

## 8.4 Boundary Conditions

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

- Recognition of boundary condition names as strings (in runtime parameters) and constants (in the source code); these are defined in `RuntimeParameters.mapStrToInt` and in `constants.h`, respectively.
- Handling of periodic boundary conditions; this is done within the underlying `GridMain` implementation. When using PARAMESH, the subroutine `gr_createDomain` is responsible for setting the neighbors of top-level blocks (to either other top-level blocks or to external boundary conditions) at initialization, after  $\text{Nblockx} \times \text{Nblocky} \times \text{Nblockz}$  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. Boundaries are selected by setting runtime parameters such as `xl_boundary_type` (for the ‘left’  $X$ -boundary) to one of the supported boundary types (Section 8.4.1) in `flash.par`. Even though the runtime parameters for specifying boundary condition types are strings, the Grid unit understands them as defined integer constants defined in the file `constants.h`, which contains all global constants for the code. The translation from the string specified in “flash.par” to the constant understood by the Grid unit is done by the routine `RuntimeParameters_mapStrToInt`.

<code>ab_boundary_type</code>	Description
<code>periodic</code>	Periodic (‘wrap-around’)
<code>reflect,reflecting</code>	Non-penetrating boundaries; plane symmetry, the normal vector components change sign
<code>outflow</code>	Zero-gradient boundary conditions; allows shocks to leave the domain
<code>diode</code>	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
<code>axisymmetric</code>	like <code>reflect</code> , but both normal and toroidal vector components change sign. Typically used with cylindrical geometry (R-Z) for the Z symmetry axis.
<code>eqtsymmetric</code>	like <code>reflect</code> for velocities but the magnetic field components, poloidal and toroidal, change sign. The sign of the normal magnetic field component remains the same. Typically used with cylindrical geometry (R-Z) for the R axis to emulate equatorial symmetry.
<code>hydrostatic-f2</code>	Hydrostatic boundary handling as in FLASH2. See remark in text.
<code>hydrostatic-f2+nvrefl</code> , <code>hydrostatic-f2+nvout</code> , <code>hydrostatic-f2+nvdiod</code>	Variants of <code>hydrostatic-f2</code> , where the normal velocity is handled specially in various ways, analogous to <code>reflect</code> , <code>outflow</code> , and <code>diode</code> boundary conditions, respectively. See remark in text.
<code>user-defined</code> or <code>user</code>	The user must implement the desired boundary behavior; see text.

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

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

```
REQUIRES Grid/GridBoundaryConditions/Flash2HSE
```

in the simulation directory’s `Config` file.

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

### 8.4.2 Boundary Conditions at Obstacles

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



<i>ab</i> _boundary_type	Constant	Remark
isolated	—	used by Gravity only for <b>grav_boundary_type</b>
—	DIRICHLET	used for multigrid solver
—	GRIDBC_MG_EXTRAPOLATE	for use by multigrid solver
—	PNEUMANN	(for use by multigrid solver)
hydrostatic	HYDROSTATIC	Hydrostatic, other implementation than FLASH2
hydrostatic+nvrefl	HYDROSTATIC_NVREFL	Hydrostatic variant, other impl. than FLASH2
hydrostatic+nvout	HYDROSTATIC_NVOUT	Hydrostatic variant, other impl. than FLASH2
hydrostatic+nvdiodi	HYDROSTATIC_NVDIODE	Hydrostatic variant, other impl. than FLASH2

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

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

### 8.4.3 Implementing Boundary Conditions

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

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

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

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

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

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

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

#### 8.4.3.1 Additional Concerns with PARAMESH 4

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

1. In other `Grid` implementations, filling of domain boundary guard cells is under control of the “user” (in this context, the user of the grid implementation, *i.e.*, FLASH): These cells can be filled for all blocks at a time that is predictable to the user code, as a standard part of handling `Grid_fillGuardCells`, only. With PARAMESH 4, the user-provided `amr_1blk_bcset` routine can be called from within the depths of PARAMESH on individual blocks (and cell regions, see below) during guard cell filling and at other times when the user has called a PARAMESH routine. It is not easy to predict when and in which sequence this will happen.
2. PARAMESH 4 does not want all boundary guard cells filled in one call, but requests individual regions in various calls.
3. PARAMESH 4 does not let the user routine `amr_1blk_bcset` operate on permanent storage (`unk` *etc.*) directly, but on (regions of) one-block buffers.
4. PARAMESH 4 occasionally invokes `amr_1blk_bcset` to operate on regions of a block that belongs to a remote processor (and for which data has been cached locally). Such block data is not associated with a valid local `blockID`, making it more complicated for user code to retrieve metadata that may be needed to implement the desired boundary handling.

Some consequences of this for FLASH3 users:

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

User code also is not allowed to make certain PARAMESH 4 calls while a call to `amr_1blk_bcset` is active, namely those that would modify the same one-block buffers that the current call is working on.

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

### Caveat Block Handles

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

## 8.5 Uniform Grid

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

### 8.5.1 `FIXEDBLOCKSIZE` Mode

`FIXEDBLOCKSIZE` mode, also called static mode, is the default for the uniform grid. In this mode, the block size is specified at compile time as `NXB×NYB×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  $NXB * iprocs \times NYB * jprocs \times NZB * kprocs$ . For example, if  $iprocs = jprocs = 4$  and  $kprocs = 1$ , the execution command should specify  $np = 16$  processors.

```
mpirun -np 16 flash3
```

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

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

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

### 8.5.2 NONFIXEDBLOCKSIZE mode

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

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

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

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

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

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  $nvar \times (2 * nguard + nxb) \times (2 * nguard + nyb) \times (2 * nguard + nzb)$  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  $64 \times 64 \times 64$ , being run on 8 processors.

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

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

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

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

## 8.6 Adaptive Mesh Refinement (AMR) Grid with Paramesh

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

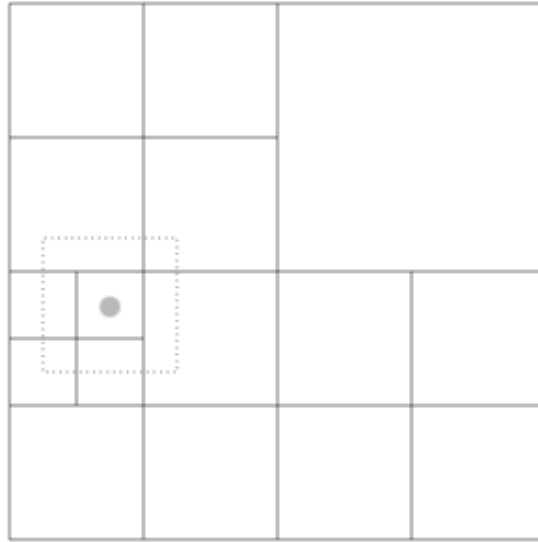


Figure 8.5: A simple computational domain showing varying levels of refinement in a total of 16 blocks. The dotted lines outline the guard cells for the block marked with a circle.

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

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

A simple two-dimensional domain is shown in **Figure 8.5**, illustrating the rules above. Each block contains  $\text{nxb} \times \text{nyb} \times \text{nz b}$  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  $\text{NUNK\_VARS} \times (2 * \text{nguard} + \text{nxb}) \times (2 * \text{nguard} + \text{nyb}) \times (2 * \text{nguard} + \text{nz b}) \times \text{MAXBLOCKS}$ . **PARAMESH** also needs memory to store its tree data structure for adaptive mesh management, over and above what is already mentioned with Uniform Grid. As the levels of refinement increase, the size of the tree also grows.

**PARAMESH** handles the filling of guard cells with information from other blocks or, at the boundaries of the physical domain, from an external boundary routine (see **Section 8.4**). If a block’s neighbor exists and has the same level of refinement, **PARAMESH** fills the corresponding guard cells using a direct copy

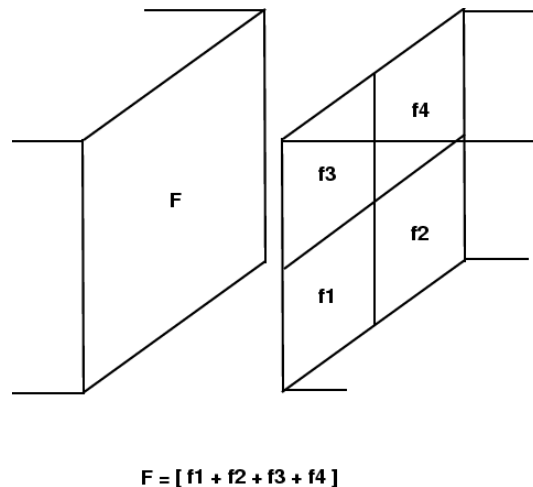


Figure 8.6: Flux conservation at a jump in refinement. The fluxes in the fine cells are added and replace the coarse cell flux ( $F$ ).

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

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

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

### 8.6.1 Additional Data Structures

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

FLASH has its own `oneBlock` data structure that stores block specific information. This data structure keeps the physical coordinates of each cell in the block. For each dimension, the coordinates are stored for the `LEFT_EDGE`, the `RIGHT_EDGE` and the center of the cell. The coordinates are determined from “`cornerID`”



which is also a part of this data structure.

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

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

## 8.6.2 Grid Interpolation (and Averaging)

The adaptive grid requires data **interpolation** or **averaging** when the refinement level (*i.e.*, mesh resolution) changes in space or in time.<sup>3</sup> If during guardcell filling a block's neighbor has a coarser level of refinement, the neighbor's cells are used to **interpolate** guard cell values to the cells of the finer block. Interpolation is also used when filling the blocks of children newly created in the course of automatic refinement. Data **averaging** is used to adapt data in the opposite direction, *i.e.*, from fine to coarse.

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

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

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

1. monotonic
2. native

The choice is made at `setup` time, see [Section 5.2](#).

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

### 8.6.2.1 Interpolation for mass-specific solution variables

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

`FLASH4` provides three ways to deal with this:

---

<sup>3</sup>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.

1. Do nothing—*i.e.*, assume that ignoring the difference between mass-specific and conserved form is a reasonable approximation. Depending on the smoothness of solutions in regions where refinement, derefinement, and jumps in refinement level occur, this assumption may be acceptable. This behavior can be forced by setting the `convertToConsvdInMeshInterp` runtime parameter to `.false`.
2. Convert mass-specific variables to conserved form *in all blocks throughout the physical domain* before invoking a `Grid` function that may result in some data interpolation or restriction (refinement, derefinement, guardcell filling); and convert back after these functions return. Conversion is done by cell-by-cell multiplication with the density (*i.e.*, the value of the “`dens`” variable, which should be declared as

```
VARIABLE dens TYPE: PER_VOLUME
```

in a `Config` file).

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

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

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

## 8.6.3 Refinement

### 8.6.3.1 Refinement Criteria

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

#### FLASH Transition

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

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

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

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



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

$$E_{i_1 i_2 i_3} = \left\{ \frac{\sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right)^2}{\sum_{pq} \left[ \left( \left| \frac{\partial u}{\partial x_p} \right|_{i_p+1/2} + \left| \frac{\partial u}{\partial x_p} \right|_{i_p-1/2} \right) \Delta x_p + \epsilon \frac{\partial^2 |u|}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right]^2} \right\}^{1/2}, \quad (8.2)$$

where the sums are carried out over coordinate directions, and where, unless otherwise noted, partial derivatives are evaluated at the center of the  $i_1 i_2 i_3$ -th cell.

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

$$E_i = \frac{|u_{i+2} - 2u_i + u_{i-2}|}{|u_{i+2} - u_i| + |u_i - u_{i-2}| + \epsilon[|u_{i+2}| + 2|u_i| + |u_{i-2}|]}, \quad (8.3)$$

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

$$E_{i_X i_Y i_Z} = \left\{ \frac{\sum_{pq} \left( \frac{\partial^2 u}{\partial x_p \partial x_q} \right)^2}{\sum_{pq} \left[ \frac{1}{2 \Delta x_q} \left( \left| \frac{\partial u}{\partial x_p} \right|_{i_q+1} + \left| \frac{\partial u}{\partial x_p} \right|_{i_q-1} \right) + \epsilon \frac{|u_{pq}|}{\Delta x_p \Delta x_q} \right]^2} \right\}^{1/2}, \quad (8.4)$$

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  $i_X i_Y i_Z$ -th cell, and  $|u_{pq}|$  stands for an *average* of the values of  $|u|$  over several neighboring cells in the  $p$  and  $q$  directions.

The constant  $\epsilon$  is by default given a value of  $10^{-2}$ , and can be overridden through the `refine.filter.#` runtime parameters. Blocks are marked for refinement when the value of  $E_{i_X i_Y i_Z}$  for any of the block's cells exceeds a threshold given by the runtime parameters `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  $E_{i_X i_Y i_Z}$  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



- `gr_lrefineMaxRedRadiusFact`
- `gr_lrefineMaxRedDoByTime`
- `gr_lrefineMaxRedTimeScale`
- `gr_lrefineMaxRedTRef`
- `gr_lrefineMaxRedLogBase`

## 8.7 Chombo

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

Most `flash.par` parameters relevant to the `Grid` unit will continue to work; the differences are described later in this section. There are some important restrictions to be aware of when using this `Grid` implementation.

- Only Cartesian geometries are handled.
- Only cell-centered solution variables are supported, i.e. those variables defined with keywords `VARIABLE` and `SCRATCHCENTERVAR` in Config files.

FLASH does not include the source code for `Chombo` library and so `Chombo` must be independently downloaded and then built using the instructions given in Section 5.9. Please also see the restrictions in Hydro Section 15.1.6 and I/O Section 9.12.

Our integration of FLASH with `Chombo` is slightly unusual because we do not use the higher-level `Chombo` classes for solving time-dependent problems on an AMR mesh. The reason is that these higher level classes also "drive" the simulation, that is, control the initialization and evolution of simulations. These are tasks that already happen in a FLASH application in the `Driver` unit. Our approach is therefore to use the lower-level `Chombo` classes so that `Chombo` library is only responsible for `Grid` related tasks.

The actual patch-based mesh is allocated and managed in `Chombo` library. Each individual patch is stored in contiguous memory locations with the slowest varying dimension being the solution variable. This has allowed us to create a `Grid_getBlkPtr` that builds a 4D Fortran pointer object from the memory address of the first element of a specified patch. The patch is treated as a standard FLASH block in the physics units of FLASH. The `Grid` unit obtains the memory address and other metadata such as patch size, coordinates, index limits, cell spacing and refinement level through non-decorated C++ function calls which query `Chombo` objects.

### 8.7.1 Using Chombo in a UG configuration

This is a `Grid` implementation that operates in the same way as the standard FLASH uniform grid mode in which there is one block per MPI process. The parameters accepted are identical to those described for a non fixed blocksize uniform grid in Section 8.5.2. The only difference is that the checkpoint file is written in `Chombo` file layout rather than FLASH file layout. Setup a simulation using:

```
./setup Sedov -auto +chombo_ug -parfile=test_chombo_ug_2d.par
```

### 8.7.2 Using Chombo in an AMR configuration

The real motivation for integrating `Chombo` into FLASH is for its adaptive mesh which is patch-based rather than an oct-tree like Paramesh. It has several desirable features including:

- refinement can happen cell-by-cell rather than block-by-block.

- blocks (or patches in **Chombo** terminology) are not limited to a fixed number of cells.
- refinement jumps of greater than 2 can happen between neighboring blocks.

A major difference in the generated meshes is that there is no longer the concept of “LEAF” blocks which cover the global domain. This was useful in **Paramesh** because a union of the space covered by the children of a block cover is exactly equal to the space covered by the parent block. It is therefore possible to evolve solution only on leaf blocks. Such complete overlap is not guaranteed in **Chombo**, in fact it rarely happens. Therefore, to be compatible with **Chombo**, physics units must evolve a solution on all blocks. For reference, the LEAF definition in **Flash.h** for FLASH-**Chombo** application now means all blocks.

A FLASH application with **Chombo** uses the refinement criteria similar to the one described in Section 8.6.3. The slight modification is that the  $i_X i_Y i_Z$ -th cell is tagged when  $E_{i_X i_Y i_Z}$  exceeds a threshold value. Recall that in the **Paramesh**, the entire block is tagged for refinement when the error estimator of *any* of cells within a block exceeds a given threshold. As a further extension to this cell tagging procedure we also tag all cells within a small radius of a given tagged cell. The tagging radius is set using the runtime parameter **tagradius**; a value of 1 means that 1 cell to the left and right of a tagged cell in each dimension is also tagged. Values below the default value of 2 may cause problems because of introduction of fine-coarse interfaces in regions of the domain where there are steep gradients. For example, a Sod problem does not converge in Riemann solver when the value is 0.

The runtime parameters are described below:

- **lrefine\_max** Used in an identical way to **Paramesh**. It is still a unit-based parameter and not zero-based. Defaults to 1.
- **igridsize, jgridsize, kgridsize** The global number of cells on the coarsest level of the adaptive mesh. Varying **igridsize** gives the same level of control as the unused **Paramesh** parameters **lrefine\_min** and **nblockx**. Defaults to 16.
- **maxBlockSize** The maximum number of cells along a single dimension of a block. Defaults to 16.
- **BRMeshRefineBlockFactor** The minimum number of cells along a single dimension of a block. See **BRMeshRefine** class in **Chombo** user-guide. Defaults to 8.
- **BRMeshRefineFillRatio** Overall grid efficiency to be generated. If this number is set low, the grids will tend to be larger and less filled with tags. If this number is set high, the grids will tend to be smaller and more filled with tags. See **BRMeshRefine** class in **Chombo** user-guide. Defaults to 0.75.
- **BRMeshRefineBufferSize** Proper nesting buffer size. This will be the minimum number of level  $l$  cells between any level  $l+1$  cell and a level  $l-1$  cell. See **BRMeshRefine** class in **Chombo** user-guide. Defaults to 1.
- **verbosity** The verbosity of debugging output written to local log files - higher values mean progressively more debugging output. As a rough guide 0 means no debugging output, 2 means debugging output from FLASH objects, and values above 2 add debugging output from **Chombo** objects. Defaults to 0.
- **refRatio** The refinement jump between fine-coarse levels, typically 2, 4 or 8. Defaults to 2.
- **restrictBeforeGhostExchange** Invokes a full mesh restriction before guard cells are exchanged. This is to make the guard cell exchange behave the same way as **Paramesh** guard cell exchange. We do not know yet if it is necessary. Defaults to True.
- **tagRadius** The radius of cells around a tagged cell that should also be tagged. Defaults to 2.

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

Setup a simulation using:

```
./setup Sedov -auto +chombo_amr -parfile=test_chombo_amr_2d.par
```

The **GridParticles** and **GridSolvers** sub-unit do not interoperate with **Chombo** yet.

## 8.8 GridMain Usage

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

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

### FLASH Transition

The `Grid_getBlkData` and `Grid_putBlkData` functions replace the `dBaseGetData` and `dBasePutData` functions in FLASH2. The bulk of the `dBase` functionality from FLASH2 is now handled by the `Grid` unit. For example, the global mesh data structures “unk” and “tree” now belong to `Grid`, and any information about them is queried from it. However, `dBase` and `Grid` are not identical. `dBase` was a central storage for all data, whereas in FLASH4 some of the data, such as simulation parameters like `dt` and simulation time are owned by the `Driver` unit instead of the `Grid` unit.

### FLASH Transition

In FLASH2, variables `nxb`, `nyb` and `nzb` traditionally described blocksize. From FLASH3 on, the symbols `NXB`, `NYB`, and `NZB` are intended not to be used directly except in the `Grid` unit itself. The variables like `iHi_gc` and `iHi` etc. that marked the endpoints of the blocks were replaced in FLASH3 with in their new incarnation (`GRID_IHI_GC` etc. ) Again, these new variables are used only in sizing arrays in declaration headers. Even when they are used for array sizing, they are enclosed in preprocessor blocks that can be eliminated at compile time. This separation was done to compartmentalize the FLASH3 code. The grid layout is not required to be known in any other unit (with the exception of `IO`). FLASH3 provides an API function `Grid_getBlkIndexLimits` that fetches the block endpoints from the `Grid` unit for each individual block. The fetched values are then used as do loop end points in all  $\langle i, j, k \rangle$  loops in all the client units. When working in `NONFIXEDBLOCKSIZE` mode, the same fetched values are also used to size and allocate arrays. We have retained `GRID_IHI_GC` etc. for array sizing as compile time option for performance reasons. Statically allocated arrays allow better compiler optimization.

In addition to the functions to access the data, the `Grid` unit also provides a collection of routines that drive some housekeeping functions of the grid without explicitly fetching any data. A good example of such routines is `Grid_fillGuardCells`. Here no data transaction takes place between `Grid` and the calling unit. The calling unit simply instructs the `Grid` unit that it is ready for the guard cells to be updated, and doesn't concern itself with the details. The `Grid_fillGuardCells` routine makes sure that all the blocks get the right data in their guardcells from their neighbors, whether they are at the same, lower or higher resolution, and if instructed by the calling routine, also ensures that EOS is applied to them.

In large-scale, highly parallel FLASH simulations with AMR, the processing of `Grid_fillGuardCells` calls may take up a significant part of available resource like CPU time, communication bandwidth, and buffer space. It can therefore be important to optimize these calls in particular. From FLASH3.1, `-Grid_fillGuardCells` provides ways to

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

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

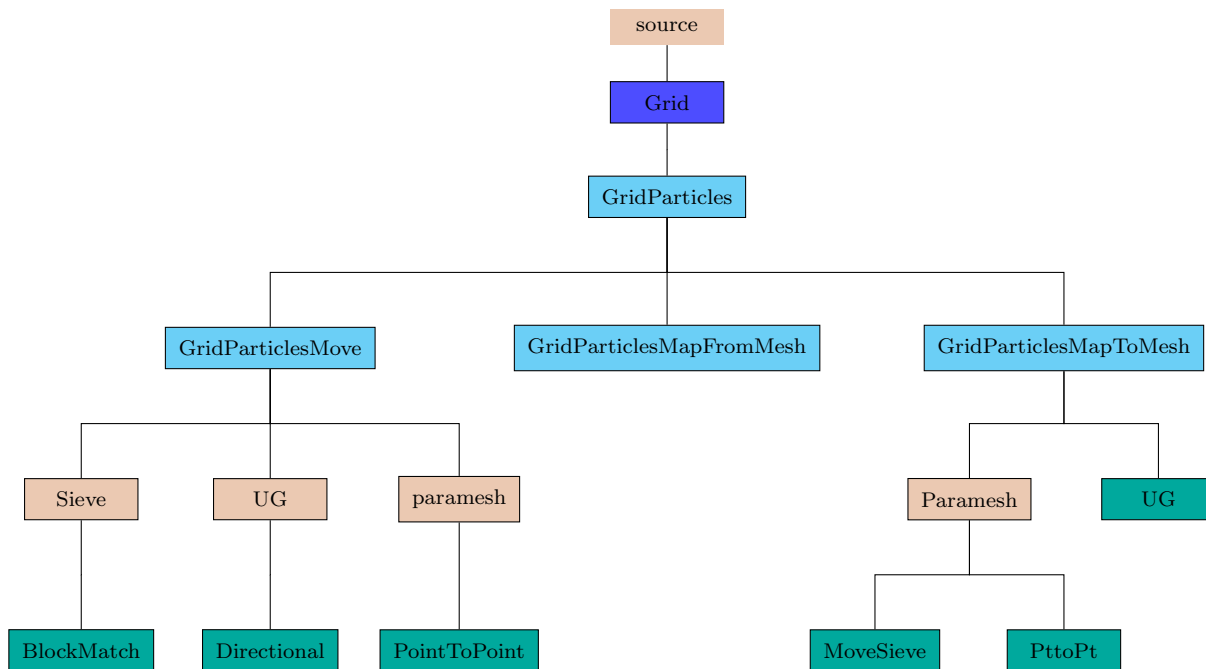
Another routine that may change the global state of the grid is `Grid_updateRefinement`. This function is called when the client unit wishes to update the grid's resolution. again, the calling unit does not need to know any of the details of the refinement process.

### FLASH Transition

As mentioned in [Chapter 4](#), FLASH allows every unit to identify scalar variables for check-pointing. In the `Grid` unit, the function that takes care of consolidating user specified checkpoint variable is `Grid_sendOutputData`. Users can select their own variables to check-point by including an implementation of this function specific to their requirements in their Simulation setup directory.

## 8.9 GridParticles

FLASH is primarily an Eulerian code, however, there is support for tracing the flow using Lagrangian particles. In FLASH4 we have generalized the interfaces in the Lagrangian framework of the `Grid` unit in such a way that it can also be used for miscellaneous non-Eulerian data such as tracing the path of a ray through the domain, or tracking the motion of solid bodies immersed in the fluid. FLASH also uses active particles with mass in cosmological simulations, and charged particles in a hybrid PIC solver. Each particle has an associated data structure, which contains fields such as its physical position and velocity, and relevant physical attributes such as mass or field values in active particles. Depending upon the time advance method, there may be other fields to store intermediate values. Also, depending upon the requirements of the simulation, other physical variables such as temperature *etc.* may be added to the data structure. The `GridParticles` subunit of the `Grid` unit has two sub-subunits of its own. The `GridParticlesMove` sub-subunit moves the data structures associated with individual particles when the particles move between blocks; the actual movement of the particles through time advancement is the responsibility of the `Particles`

Figure 8.8: The `Grid` unit: structure of `GridParticles` subunit.

unit. Particles move from one block to another when their time advance places them outside their current block. In AMR, the particles can also change their block through the process of refinement and derefinement. The `GridParticlesMap` sub-subunit provides mapping between particles data and the mesh variables. The mesh variables are either cell-centered or face-centered, whereas a particle's position could be anywhere in the cell. The `GridParticlesMap` sub-subunit calculates the particle's properties at its position from the corresponding mesh variable values in the appropriate cell. When using active particles, this sub-subunit also maps the mass of the particles onto the specified mesh variable in appropriate cells. The next sections describe the algorithms for moving and mapping particles data.

### 8.9.1 GridParticlesMove

FLASH4 has implementations of three different parallel algorithms for moving the particles data when they are displaced from their current block. FLASH3 had an additional algorithm, `Perfect Tree Level` which made use of the oct-tree structure. However, because in all performance experiments it performed significantly worse than the other two algorithms, it is not supported currently in FLASH4. The simplest algorithm, `Directional algorithm` is applicable only to the uniform grid when it is configured with one block per processor. This algorithm uses directional movement of data, and is easy because the directional neighbors are trivially known. The movement of particles data is much more challenging with AMR even when the grid is not refining. Since the blocks are at various levels of refinement at any given moment, a block may have more than one neighbor along one or more of its faces. The distribution of blocks based on space-filling curve is an added complication since the neighboring blocks along a face may reside at a non-neighboring processor. The remaining two algorithms included in FLASH4 implement `GridParticlesMove` subunit for the adaptive mesh; `Point to Point` and `Sieve`, of which only the `Sieve` algorithm is able to move the data when the mesh refines. Thus even when a user opts for the `PointToPoint` implementation for moving particles with time evolution, some part of the `Sieve` implementation must necessarily be included to successfully move the data upon refinement.



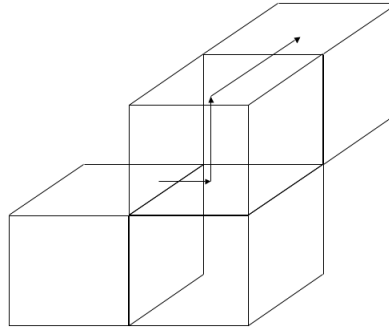


Figure 8.9: Moving one particle to a neighbor on the corner.

#### 8.9.1.1 Directional Move

The Directional Move algorithm for moving particles in a Uniform Grid minimizes the number of communication steps instead of minimizing the volume of data moved. Its implementation has the following steps:

1. Scan particle positions. Place all particles with their  $x$  coordinate value greater than the block bounding box in the Rightmove bin, and place those with  $x$  coordinate less than block bounding box in Leftmove bin.
2. Exchange contents of Rightbin with the right block neighbor's Leftbin contents, and those of the Leftbin with left neighbor's Rightbin contents.
3. Merge newly arrived particles from step 2 with those which did not move outside their original block.
4. Repeat steps 1-3 for the  $y$  direction.
5. Repeat step 1-2 for the  $z$  direction.

At the end of these steps, all particles will have reached their destination blocks, including those that move to a neighbor on the corner. [Figure 8.9](#) illustrates the steps in getting a particle to its correct destination.

#### 8.9.1.2 Point To Point Algorithm

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



### 8.9.1.3 Sieve Algorithm

The **Sieve** algorithm does not concern itself with the configuration of the underlying mesh at any time. It also does not distinguish between data movements due to time evolution or regridding, and is therefore the only usable implementation when the particles are displaced as a consequence of mesh refinement. The sieve implementation works with two bins, one collects particles that have to be moved off-processor, and the other receives particles sent to it by other processors. The following steps broadly describe the algorithm:

1. For each particle, find if its current position is on the current block
2. If not, find if its current position is on another block on the same processor. If it is move the particle to that block, otherwise put it in the send bin.
3. Send contents of the send bin to the designated neighbor, and receive contents of another neighbor's send bin into my receive bin.
4. Repeat step 2 on the contents of the receive bin, and step 3 until all particles are at their destination.
5. For every instance of step 3, the designated send and receive neighbors are different from any of the previous steps.

In this implementation, the trick is to use an algorithm to determine neighbors in such a way that all the particles reach their destination in minimal number of hops. Using  $MyPE + n * (-1)^{n+1}$  as the destination processor and  $MyPE + n * (-1)^n$  as the source processor in modulo *numProcs* arithmetic meets the requirements. Here *MyPE* is the local processor number and *numProcs* is the number of processors.

## 8.9.2 GridParticlesMapToMesh

FLASH4 provides support for particles that can experience forces and contribute to the problem dynamics. These are termed *active* particles, and are described in detail in [Chapter 21](#). As these particles may move independently of fluid flow, it is necessary to update the grid by mapping an attribute of the particles to the cells of the grid. We use these routines, for example, during the PM method to assign the particles' mass to the particle density solution variable **pden**. The hybrid PIC method uses its own variation for mapping appropriate physical quantities to the mesh.

In general the mapping is performed using the grid routines in the **GridParticlesMapToMesh** directory and the particle routines in the **ParticlesMapping** directory. Here, the particle subroutines map the particles' attribute into a temporary array which is independent of the current state of the grid, and the grid subroutines accumulate the mapping from the array into valid cells of the computational domain. This means the grid subroutines accumulate the data according to the grid block layout, the block refinement details, and the simulation boundary conditions. As these details are closely tied with the underlying grid, there are separate implementations of the grid mapping routines for UG and **PARAMESH** simulations.

The implementations are required to communicate information in a relatively non-standard way. Generally, domain decomposition parallel programs do not write to the guard cells of a block, and only use the guard cells to represent a sufficient section of the domain for the current calculation. To repeat the calculation for the next time step, the values in the guard cells are refreshed by taking updated values from the internal section of the relevant block. In FLASH4, **PARAMESH** refreshes the values in the guard cells automatically, and when instructed by a **Grid\_fillGuardCells** call.

In contrast, the guard cell values are mutable in the particle mapping problem. Here, it is possible that a portion of the particle's attribute is accumulated in a guard cell which represents an internal cell of another block. This means the value in the updated guard cell must be communicated to the appropriate block. Unfortunately, the mechanism to communicate information in this direction is not provided by **PARAMESH** or UG grid. As such, the relevant communication is performed within the grid mapping subroutines directly.

In both **PARAMESH** and UG implementations, the particles' attribute is "smeared" across a temporary array by the generic particle mapping subroutine. Here, the temporary array represents a single leaf block from the local processor. In simulations using the **PARAMESH** grid, the temporary array represents each LEAF block from the local processor in turn. We assign a particle's attribute to the temporary array when that

particle exists in the same space as the current leaf block. For details about the attribute assignment schemes available to the particle mapping sub-unit, please refer to [Section 21.2](#).

After particle assignment, the **Grid** sub-unit applies simulation boundary conditions to those temporary arrays that represent blocks next to external boundaries. This may change the quantity and location of particle assignment in the elements of the temporary array. The final step in the process involves accumulating values from the temporary array into the correct cells of the computational domain. As mentioned previously, there are different strategies for **UG** and **PARAMESH** grids, which are described in [Section 8.9.2.1](#) and [Section 8.9.2.2](#), respectively.

### FLASH Transition

The particle mapping routines can be run in a custom debug mode which can help spot data errors (and even detect possible bugs). In this mode we inspect data for inconsistencies. To use, append the following line to the setup script:

```
-defines=DEBUG_GRIDMAPPARTICLES
```

#### 8.9.2.1 Uniform Grid

The Uniform Grid algorithm for accumulating particles' attribute on the grid is similar to the particle redistribution algorithm described in [Section 8.9.1.1](#). We once again apply the concept of directional movement to minimize the number of communication steps:

1. Take the accumulated temporary array and iterate over all elements that correspond to the x-axis guard cells of the low block face. If a guard cell has been updated, determine its position in the neighboring block of the low block face. Copy the guard cell value and a value which encodes the destination cell into the send buffer.
2. Send the buffer to the low-side processor, and receive a buffer from the high-side processor. For processors next to a domain boundary assume periodic conditions because all processors must participate. If the simulation does not have periodic boundary conditions, there is still periodic communication at the boundaries, but the send buffers do not contain data.
3. Iterate over the elements in the receive buffer and accumulate the values into the local temporary array at the designated cells. It is possible to accumulate values in cells that represent internal cells and guard cells. A value accumulated in a guard cell will be repacked into the send buffer during the next directional (y or z) sweep.
4. Repeat steps 1-3 for the high block face.
5. Repeat steps 1-4 for the y-axis, and then the z-axis.

When the guard cell's value is packed into the send buffer, a single value is also packed which is a 1-dimensional representation of the destination cell's N-dimensional position. The value is obtained by using an array equation similar to that used by a compiler when mapping an array into contiguous memory. The receiving processor applies a reverse array equation to translate the value into N-dimensional space. The use of this communication protocol is designed to minimize the message size.

At the end of communication, each local temporary buffer contains accumulated values from guard cells of another block. The temporary buffer is then copied into the solution array.

#### 8.9.2.2 Paramesh Grid

There are two implementations of the AMR algorithms for accumulating particles' attribute on the grid. They are inspired by a particle redistribution algorithms **Sieve** and **Point to Point** described in [Section 8.9.1.3](#) and [Section 8.9.1.2](#) respectively.

The **MoveSieve** implementation of the mapping algorithm uses the same back and forth communication pattern as **Sieve** to minimize the number of message exchanges. That is, processor  $MyPE$  sends to  $MyPE + n * (-1)^{n+1}$  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 space filling curve should be high in data locality, *i.e.*, nearest neighbor blocks should be placed on the same processor or nearest neighbor processors.

Similarly, the **Point to Point** implementation of the mapping algorithm exploits the cached neighborhood knowledge, and uses a judicious combination of global communications with asynchronous receives and synchronous sends, as described in [Section 8.9.1.2](#). Other than their communication patterns, the two implementations are very similar as described below.

1. Accumulate the temporary array values into the central section of the corresponding leaf block.
2. Divide the leaf block guard cells into guard cell regions. Determine whether the neighbor(s) to a particular guard cell region exist on the same processor.
3. If a neighbor exists on the same processor, the temporary array values are accumulated into the central cells of that leaf block. If the neighbor exists off processor, all temporary array values corresponding to a single guard cell region are copied into a send buffer. Metadata is also packed into the send buffer which describes the destination of the updated values.
4. Repeat steps 1-3 for each leaf block.
5. Carry out data exchange with off-processor destinations as described in the [Section 8.9.1.3](#) or [Section 8.9.1.2](#)

The guard cell region decomposition described in Step 2 is illustrated in [Figure 8.10](#). Here, the clear regions correspond to guard cells and the gray region corresponds to internal cells. Each guard cell region contains cells which correspond to the internal cells of a single neighboring block at the same refinement.

3,1	3,2	3,3
2,1		2,3
1,1	1,2	1,3

Figure 8.10: A single 2-D block showing how guard cells are divided into regions.

We use this decomposition as it makes it possible to query public **PARAMESH** data structures which contain the block and process identifier of the neighboring block at the same refinement. However, at times this is not enough information for finding the block neighbor(s) in a refined grid. We therefore categorize neighboring blocks as: Existing on the same processor, existing on another processor and the block and process ID are

known, and existing on another processor and the block and process ID are unknown. If the block and process identifier are unknown we use the FLASH4 corner ID. This is a viable alternative as the corner ID of a neighboring block can always be determined.

The search process also identifies the refinement level of the neighboring block(s). This is important as the guard cell values cannot be directly accumulated into the internal cells of another block if the blocks are at a different refinement levels. Instead the values must be operated on in processes known as restriction and prolongation (see [Section 8.6.2](#)). We perform these operations directly in the `GridParticlesMapToMesh` routines, and use quadratic interpolation during prolongation.

Guard cell data is accumulated in blocks existing on the same processor, or packed into a send buffer ready for communication. When packed into the send buffer, we keep values belonging to the same guard cell region together. This enables us to describe a whole region of guard cell data by a small amount of metadata. The metadata consists of: Destination block ID, destination processor ID, block refinement level difference, destination block corner ID (IAXIS, JAXIS, KAXIS) and start and end coordinates of destination cells (IAXIS, JAXIS, KAXIS). This is a valid technique because there are no gaps in the guard cell region, and is sufficient information for a receiving processor to unpack the guard cell data correctly.

We size the send / receive buffers according to the amount of data that needs to be communicated between processors. This is dependent upon how the `PARAMESH` library distributes blocks to processors. Therefore, in order to size the communication buffers economically, we calculate the number of guard cells that will accumulate on blocks belonging to another processor. This involves iterating over every single guard cell region, and keeping a running total of the number of off-processor guard cells. This total is added to the metadata total to give the size of the send buffer required on each processor. We use the maximum of the send buffer size across all processors as the local size for the send / receive buffer. Choosing the maximum possible size prevents possible buffer overflow when an intermediate processor passes data onto another processor.

After the point to point communication in step 6, the receiving processor scans the destination processor identifier contained in each metadata block. If the data belongs to this processor it is unpacked and accumulated into the central cells of the relevant leaf block. As mentioned earlier, it is possible that some guard cell sections do not have the block and processor identifier. When this happens, the receiving processor attempts to find the same corner ID in one of its blocks by performing a linear search over each of its leaf blocks. Should there be a match, the guard cells are copied into the matched block. If there is no match, the guard cells are copied from the receive buffer into the send buffer, along with any guard cell region explicitly designated for another processor. The packing and unpacking will continue until all send buffers are empty, as indicated by the result of the collective communication.

It may seem that the algorithm is unnecessarily complicated, however, it is the only viable algorithm when the block and process identifiers of the nearest block neighbors are unknown. This is the situation in FLASH3.0, in which some data describing block and process identifiers are yet to be extracted from `PARAMESH`. As an aside, this is different to the strategy used in FLASH2, in which the entire `PARAMESH` tree structure was copied onto each processor. Keeping a local copy of the entire `PARAMESH` tree structure on each processor is an unscalable approach because increase in the levels of resolution increases the meta-data memory overhead, which restricts the size of active particle simulations. Therefore, Point to Point method is a better option for larger simulations, and significantly, simulations that run on massively parallel processing (MPP) hardware architectures.

In FLASH3.1 we added a routine which searches non-public `PARAMESH` data to obtain **all** neighboring block and process identifiers. This discovery greatly improves the particle mapping performance because we no longer need to perform local searches on each processor for blocks matching a particular corner ID.

As another consequence of this discovery, we are able to experiment with alternative mapping algorithms that require all block and process IDs. From FLASH3.1 on we also provide a non-blocking point to point implementation in which off-processor cells are sent directly to the appropriate processor. Here, processors receive messages at incremented positions along a data buffer. These messages can be received in any order, and their position in the data buffer can change from run to run. This is very significant because the mass accumulation on a particular cell can occur in any order, and therefore can result in machine precision discrepancies. Please be aware that this can actually lead to slight variations in end-results between two runs of the exact same simulation.

## 8.10 GridSolvers

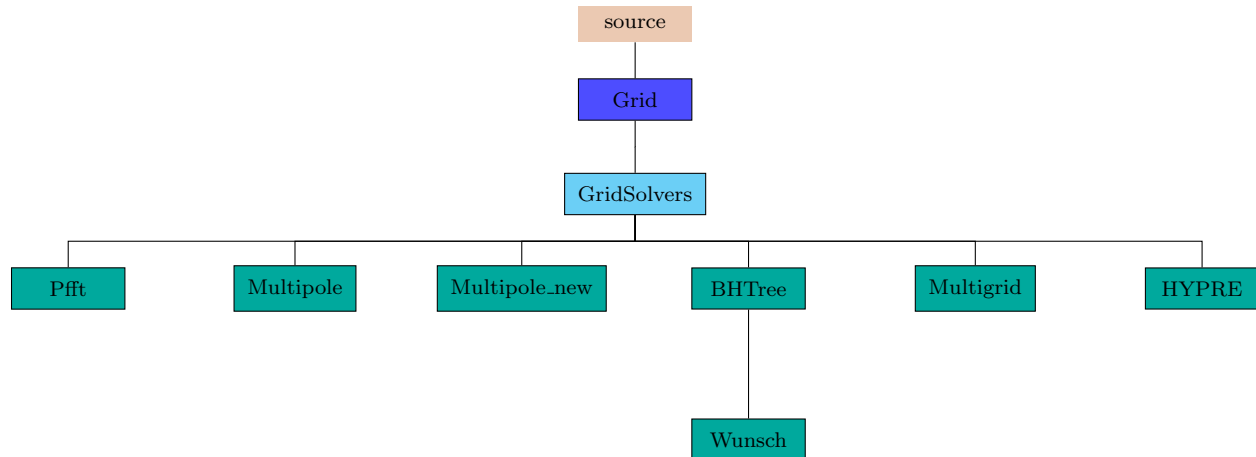


Figure 8.11: The **Grid** unit: structure of **GridSolvers** subunit.

The **GridSolvers** unit groups together subunits that are used to solve particular types of differential equations. Currently, there are two types of solvers: a parallel Fast Fourier Transform package ([Section 8.10.1](#)) and various solvers for the Poisson equation ([Section 8.10.2](#)).

### 8.10.1 Pfft

**Pfft** is a parallel frame work for computing a Fast Fourier Transform (FFT) on uniform grids. It can also be combined with the Multigrid solver described below in [Section 8.10.2.6](#) to let the composite solver scale to thousands of processors.

**Pfft** has a layered architecture where the lower layer contains functions implementing basic tasks and primary data structures. The upper layer combines pieces from the lowest layer to interface with FLASH and create the parallel transforms. The computational part of **Pfft** is handled by sequential 1-dimensional FFT's, which can be from a native, vendor supplied scientific library, or from public domain packages. The current distribution of FLASH uses **fftpack** from NCAR for the 1-D FFTs, since that package also contains transforms that are useful with non-periodic boundary conditions.

The lowest layer has three distinct components. The first component redistributes data. It includes routines for distributed transposes for two and three dimensional data. The second component provides a uniform interface for FFT calls to hide the details of individual libraries. The third component is the data structures. There are global data structures to keep track of the type of transform, number of data dimensions, and physical and transform space information about each dimension. The dimensional information includes the start and end point of data (useful when the dimension is spread over more than one processor), the MPI communicator, the coordinates of the node in the processor grid etc. The structures also include pointers to the trigonometric tables and work space needed by the sequential FFT calls.

The upper layer of PFFT combines the lower layer routines to create end-to-end transforms in a variety of ways. The available one dimensional transforms include real-to-complex, complex-to-complex, sine and cosine transforms. They can be combined to create two or three dimensional 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  $\mathbf{x}$ .

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:

$$N_1 \times N_2 \times N_3 / P, \quad (8.5)$$

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

$$\begin{aligned} N_1 \times N_2 / P_1 \times N_3 / P_2 \\ P = P_1 \times P_2. \end{aligned} \quad (8.6)$$

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

### 8.10.1.1 Using Pfft

**Pfft** can only be used with a pencil grid, with the constraint that the number of processors along the **IAXIS** must be 1. This is because all one dimensional transforms are computed locally within a processor. However, FLASH contains a set of data movement subroutines that generate a usable pencil grid from any UG grid or any level of a PM grid. These routines are briefly explained in Section 8.10.1.2.

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

### 8.10.1.2 Pfft data movement subroutines

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

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

Both mesh reconfiguration implementations generate a map describing the data movement before moving any grid data. The map is retained between calls to the Pfft routines and is only regenerated when the grid changes. This avoids repeating the same global communications, but means communication buffers are left allocated between calls to Pfft.

In the `Collective` implementation, the map coordinates are used to specify where the FLASH data is copied into a send communication buffer. Two `MPI_Alltoall` calls then move 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:

```
Memory(bytes) = sizeof(REAL) * total grid points at solve level * 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`s, and then traversing the FLASH list and sending block fragment using `MPI_Isends`. After performing `MPI_Waits`, the received data in the nodes of the Pfft list is copied into internal Pfft arrays.

Note, the linked list is constructed using an include file stored at `flashUtilities/datastructures/-linkedlist`. The file is named `ut_listMethods.includeF90` and is meant to be included in any Fortran90 module to create lists with nodes of a user-defined type. Please see the README file, and the unit test example at `flashUtilities/datastructures/linkedlist/UnitTest`.

### 8.10.1.3 Unit Test

The unit test for Pfft solver solves the following equation:

$$\nabla^2(\mathbf{F}) = -13.0 * \cos 2x * \sin 3y \quad (8.7)$$



The simplest analytical solution of this equation assuming no constants is

$$F = \cos 2x * \sin 3y \quad (8.8)$$

We discretize the domain by assuming  $xmin, ymin, zmin = 0$ , and  $xmax, ymax, zmax = 2\pi$ . The equation satisfies periodic boundary conditions in this formulation and FFT based poisson solve techniques can be applied. In the unit test we initialize one variable of the solution data with the function  $F$ , and another one with the right hand side of (8.7). We compute the forward real-to-complex transform of the solution data variable that is initialized with the right hand side of (8.7). This variable is then divided by  $(k_i^2 + k_j^2 + k_k^2)$  where  $k_i, k_j$  and  $k_k$  are the wavenumbers at any point  $i, j, k$  in the domain. An inverse complex-to-real transform after the division should give the function  $F$  as result. Hence the unit test is considered successful if both the variables have matching values within the specified tolerance.

## 8.10.2 Poisson equation

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

$$\nabla^2 \phi(\mathbf{x}) = \alpha \rho(\mathbf{x}) . \quad (8.9)$$

Here  $\alpha$  is a constant that depends upon the application. For example, when the gravitational Poisson equation is being solved,  $\rho(\mathbf{x})$  is the mass density,  $\phi(\mathbf{x})$  is the gravitational potential, and  $\alpha = 4\pi G$ , where  $G$  is Newton's gravitational constant.

### 8.10.2.1 Multipole Poisson solver (original version)

This section describes the multipole Poisson solver that has been included in all the past releases of FLASH. It is included in the current release also, however, certain limitations found in this solver lead to a new implementation described in [Section 8.10.2.2](#). This version is retained in FLASH4, because the new version is missing the ability to treat a non-zero minimal radius for spherical geometries and the ability to specify a point mass contribution to the potential. This will be implemented for the next coming release.

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions (Müller (1995)). 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  $\mathbf{x}_{cm}$

$$\mathbf{x}_{cm} = \frac{\int d^3\mathbf{x} \mathbf{x} \rho(\mathbf{x})}{\int d^3\mathbf{x} \rho(\mathbf{x})} . \quad (8.10)$$

We will take  $\mathbf{x}_{cm}$  as our origin. In integral form, Poisson's ((8.9)) is

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \int d^3\mathbf{x}' \frac{\rho(\mathbf{x}')}{|\mathbf{x} - \mathbf{x}'|} . \quad (8.11)$$

The Green's function for this equation satisfies the relationship

$$\frac{1}{|\mathbf{x} - \mathbf{x}'|} = 4\pi \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell+1} \frac{r_{<}^{\ell}}{r_{>}^{\ell+1}} Y_{\ell m}^*(\theta', \varphi') Y_{\ell m}(\theta, \varphi) , \quad (8.12)$$

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

$$\begin{aligned} r_{<} &\equiv \min\{|\mathbf{x}|, |\mathbf{x}'|\} \\ r_{>} &\equiv \max\{|\mathbf{x}|, |\mathbf{x}'|\} . \end{aligned} \quad (8.13)$$



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

$$Y_{\ell m}(\theta, \varphi) \equiv (-1)^m \sqrt{\frac{2\ell+1}{4\pi} \frac{(\ell-m)!}{(\ell+m)!}} P_{\ell m}(\cos \theta) e^{im\varphi}. \quad (8.14)$$

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

$$\begin{aligned} \phi(\mathbf{x}) = & -\alpha \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell+1} \left\{ Y_{\ell m}(\theta, \varphi) \times \right. \\ & \left. \left[ r^{\ell} \int_{r < r'} d^3 \mathbf{x}' \frac{\rho(\mathbf{x}') Y_{\ell m}^*(\theta', \varphi')}{r'^{\ell+1}} + \frac{1}{r^{\ell+1}} \int_{r > r'} d^3 \mathbf{x}' \rho(\mathbf{x}') Y_{\ell m}^*(\theta', \varphi') r'^{\ell} \right] \right\}. \end{aligned} \quad (8.15)$$

In practice, we carry out the first summation up to some limiting multipole  $\ell_{\max}$ . 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  $\ell_{\max}$ , the error created by neglecting high-multipole terms is minimized. Note that the product of spherical harmonics in (8.15) is real-valued

$$\begin{aligned} \sum_{m=-\ell}^{\ell} Y_{\ell m}^*(\theta', \varphi') Y_{\ell m}(\theta, \varphi) = & \frac{2\ell+1}{4\pi} \left[ P_{\ell 0}(\cos \theta) P_{\ell 0}(\cos \theta') + \right. \\ & \left. 2 \sum_{m=1}^{\ell} \frac{(\ell-m)!}{(\ell+m)!} P_{\ell m}(\cos \theta) P_{\ell m}(\cos \theta') \cos(m(\varphi - \varphi')) \right]. \end{aligned} \quad (8.16)$$

Using a trigonometric identity to split up the last cosine in this expression and substituting for the inner sums in (8.15), we obtain

$$\begin{aligned} \phi(\mathbf{x}) = & -\frac{\alpha}{4\pi} \sum_{\ell=0}^{\infty} P_{\ell 0}(\cos \theta) \left[ r^{\ell} \mu_{\ell 0}^{\text{eo}}(r) + \frac{1}{r^{\ell+1}} \mu_{\ell 0}^{\text{ei}}(r) \right] - \\ & \frac{\alpha}{2\pi} \sum_{\ell=1}^{\infty} \sum_{m=1}^{\ell} P_{\ell m}(\cos \theta) \left[ (r^{\ell} \cos m\varphi) \mu_{\ell m}^{\text{eo}}(r) + (r^{\ell} \sin m\varphi) \mu_{\ell m}^{\text{oo}}(r) + \right. \\ & \left. \frac{\cos m\varphi}{r^{\ell+1}} \mu_{\ell m}^{\text{ei}}(r) + \frac{\sin m\varphi}{r^{\ell+1}} \mu_{\ell m}^{\text{oi}}(r) \right]. \end{aligned} \quad (8.17)$$

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

$$\mu_{\ell m}^{\text{ei}}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r > r'} d^3 \mathbf{x}' r'^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \cos m\varphi' \quad (8.18)$$

$$\mu_{\ell m}^{\text{oi}}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r > r'} d^3 \mathbf{x}' r'^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos \theta') \sin m\varphi' \quad (8.19)$$

$$\mu_{\ell m}^{\text{eo}}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r < r'} d^3 \mathbf{x}' \frac{\rho(\mathbf{x}')}{r'^{\ell+1}} P_{\ell m}(\cos \theta') \cos m\varphi' \quad (8.20)$$

$$\mu_{\ell m}^{\text{oo}}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r < r'} d^3 \mathbf{x}' \frac{\rho(\mathbf{x}')}{r'^{\ell+1}} P_{\ell m}(\cos \theta') \sin m\varphi'. \quad (8.21)$$

The procedure is thus to compute the moment integrals ((8.18) – (8.21)) for a given source field  $\rho(\mathbf{x})$ , and then to use these moments in (8.17) to compute the potential.

In practice, the above procedure must take account of the fact that the source and the potential are assumed to be cell-averaged quantities discretized on a block-structured mesh with varying cell size. Also, because of the radial dependence of the multipole moments of the source function, these moments must be tabulated as functions of distance from  $\mathbf{x}_{\text{cm}}$ , with an implied discretization. The solver allocates storage for moment samples spaced a distance  $\Delta$  apart in radius

$$\mu_{\ell m, q}^{\text{ei}} \equiv \mu_{\ell m}^{\text{ei}}(q\Delta) \quad \mu_{\ell m, q}^{\text{eo}} \equiv \mu_{\ell m}^{\text{eo}}((q-1)\Delta) \quad (8.22)$$

$$\mu_{\ell m, q}^{\text{oi}} \equiv \mu_{\ell m}^{\text{oi}}(q\Delta) \quad \mu_{\ell m, q}^{\text{oo}} \equiv \mu_{\ell m}^{\text{oo}}((q-1)\Delta). \quad (8.23)$$

The sample index  $q$  varies from 0 to  $N_q$  ( $\mu_{\ell m,0}^{\text{eo}}$  and  $\mu_{\ell m,0}^{\text{oo}}$  are not used). The sample spacing  $\Delta$  is chosen to be one-half the geometric mean of the  $x$ ,  $y$ , and  $z$  cell spacings at the highest level of refinement, and  $N_q$  is chosen to be large enough to span the diagonal of the computational volume with samples.

Determining the contribution of individual cells to the tabulated moments requires some care. To reduce the error caused by the grid geometry, in each cell  $ijk$  an optional subgrid can be established (see `mpole_subSample`) consisting of  $N'$  points at the locations  $\mathbf{x}'_{i'j'k'}$ , where

$$x'_{i'} = x_i + (i' - 0.5(N' - 1)) \frac{\Delta x_i}{N'}, \quad i' = 0 \dots N' - 1 \quad (8.24)$$

$$y'_{j'} = y_j + (j' - 0.5(N' - 1)) \frac{\Delta y_j}{N'}, \quad j' = 0 \dots N' - 1 \quad (8.25)$$

$$z'_{k'} = z_k + (k' - 0.5(N' - 1)) \frac{\Delta z_k}{N'}, \quad k' = 0 \dots N' - 1, \quad (8.26)$$

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

$$\mu_{\ell m, q \geq q'}^{\text{ei}} \leftarrow \mu_{\ell m, q \geq q'}^{\text{ei}} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} r'^{\ell}_{i'j'k'} \rho(\mathbf{x}'_{i'j'k'}) P_{\ell m}(\cos \theta'_{i'j'k'}) \cos m \varphi'_{i'j'k'} \quad (8.27)$$

$$\mu_{\ell m, q \geq q'}^{\text{oi}} \leftarrow \mu_{\ell m, q \geq q'}^{\text{oi}} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} r'^{\ell}_{i'j'k'} \rho(\mathbf{x}'_{i'j'k'}) P_{\ell m}(\cos \theta'_{i'j'k'}) \sin m \varphi'_{i'j'k'} \quad (8.28)$$

$$\mu_{\ell m, q \leq q'}^{\text{eo}} \leftarrow \mu_{\ell m, q \leq q'}^{\text{eo}} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{r'^{\ell+1}_{i'j'k'}} P_{\ell m}(\cos \theta'_{i'j'k'}) \cos m \varphi'_{i'j'k'} \quad (8.29)$$

$$\mu_{\ell m, q \leq q'}^{\text{oo}} \leftarrow \mu_{\ell m, q \leq q'}^{\text{oo}} + \frac{(\ell - m)!}{(\ell + m)!} \frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{r'^{\ell+1}_{i'j'k'}} P_{\ell m}(\cos \theta'_{i'j'k'}) \sin m \varphi'_{i'j'k'}, \quad (8.30)$$

where

$$q' = \left\lfloor \frac{|\mathbf{x}'_{i'j'k'}|}{\Delta} \right\rfloor + 1 \quad (8.31)$$

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 ( $\mu_{\ell m, N_q}^{\text{ei}}$ ) is exactly equal to the total implied by  $\rho_{ijk}$ .

Another way to reduce grid geometry errors when using the multipole solver is to modify the AMR refinement criterion to refine all blocks containing the center of mass (in addition to other criteria that may be used, such as the second-derivative criterion supplied with `PARAMESH`). This ensures that the center-of-mass point is maximally refined at all times, further restricting the volume which contributes errors to the moments because  $r' \sim \Delta x$ .

The default value of  $N'$  is 1; note that large values of this parameter very quickly increase the amount of time required to evaluate the multipole moments (as  $N'^3$ ). In order to speed up the moment summations, the sines and cosines in ((8.27)) – (8.30) are evaluated using trigonometric recurrence relations, and the factorials are pre-computed and stored at the beginning of the run.

When computing the cell-averaged potential, we again employ a subgrid, but here the subgrid points fall on cell boundaries to improve the continuity of the result. Using  $N' + 1$  subgrid points per dimension, we have

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

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

$$z'_{k'} = z_k + (k' - 0.5N') \frac{\Delta z_k}{N'}, \quad k' = 0 \dots N'. \quad (8.34)$$

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

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

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

### 8.10.2.2 Multipole Poisson solver (improved version)

The multipole Poisson solver is based on a multipolar expansion of the source (mass for gravity, for example) distribution around a conveniently chosen center of expansion. The angular number  $L$  entering this expansion is a measure of how detailed the description of the source distribution will be on an angular basis. Higher  $L$  values mean higher angular resolution with respect to the center of expansion. The multipole Poisson solver is thus appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. For problems which require high spatial resolution throughout the entire domain (like, for example, galaxy collision simulations), the multipole Poisson solver is less suited, unless one is willing to go to extremely (computationally unfeasible) high  $L$  values. For stellar evolution, however, the multipole Poisson solver is the method of choice.

The new implementation of the multipole Poisson solver is located in the directory

`source/Grid/GridSolvers/Multipole_new`.

This implementation improves upon the original implementation in many ways: i) efficient memory layout ii) elimination of numerical over- and underflow errors for large angular momenta when using astrophysical (dimensions  $\approx 10^9$ ) domains iii) elimination of subroutine call overhead (1 call per cell), iv) minimization of error due to non-statistical distributions of moments near the multipolar origin. The following paragraphs explain the new approach to the multipole solver and an explanation of the above improvements. Details about the theory of the new implementation of the Poisson solver can be found in Couch et al. (2013).

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. It currently works in 1D spherical, 2D spherical, 2D cylindrical, 3D Cartesian and 3D cylindrical. Symmetries can be specified for the 2D spherical and 2D cylindrical cases (a horizontal symmetry plane along the radial axis) and the 3D Cartesian case (assumed axisymmetric property). Because of the radial symmetry in the 1D spherical case, only the monopole term ( $\ell = 0$ ) contributes, while for the 3D Cartesian axisymmetric, the 2D cylindrical and 2D spherical cases only the  $m = 0$  moments need to be used (the other  $m \neq 0$  moments effectively cancel out).

The multipole algorithm consists of the following steps. First, the center of the multipolar expansion  $\mathbf{x}_{\text{cen}}$  is determined via density-squared weighted integration over position:

$$\mathbf{x}_{\text{cen}} = \frac{\int \mathbf{x} \rho^2(\mathbf{x}) d\mathbf{x}}{\int \rho^2(\mathbf{x}) d\mathbf{x}} . \quad (8.36)$$

We will take  $\mathbf{x}_{\text{cen}}$  as our origin. In integral form, Poisson's equation (8.9) becomes

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

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

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

where  $x_{<}$  ( $x_{>}$ ) indicate the smaller (larger) of the magnitudes and  $\gamma$  denotes the angle between  $\mathbf{x}$  and  $\mathbf{x}'$ . Note, that this definition includes those cases where both magnitudes are equal. The expansion is always convergent if  $\cos \gamma < 1$ . Expansion of the Legendre polynomials in terms of spherical harmonics gives

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

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

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

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

we can rewrite Eq.(8.37) in the form

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \int \sum_{\ell m} R_{\ell m}(x_{<}) I_{\ell m}^*(x_{>}) \rho(\mathbf{x}') d\mathbf{x}', \quad (8.42)$$

where the summation sign is a shorthand notation for the double sum over all the allowed  $\ell$  and  $m$  values. In FLASH both the source and the potential are assumed to be cell-averaged quantities discretized on a block-structured mesh with varying cell size. The integration must hence be replaced by a summation over all leaf cells

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

where  $m$  denotes the cell's mass. Note, that the symbol  $q$  stands for cell index as well as its defining distance position from the expansion center in the computational domain. This discrete Poisson equation is incorrect. It contains the divergent  $q' = q$  term on the rhs. The  $q' = q$  contribution to the potential corresponds to the cell self potential  $\phi_{\text{Self}}(q)$  and is divergent in our case because all the cell's mass is assumed to be concentrated at the cell's center. The value of this divergent term can easily be calculated from Eq.(8.38) by setting  $\cos \gamma = 1$ :

$$\phi_{\text{Self}}(q) = m(q) \frac{L+1}{x_q}, \quad (8.44)$$

where  $m$  is the cell's mass,  $L$  the highest angular number considered in the expansion and  $x_q$  the radial distance of the cell center from the expansion center. To avoid this divergence problem, we evaluate the potentials on each face of the cell and form the average of all cell face potentials to get the cell center potential. Eq.(8.43) will thus be replaced by

$$\phi(q) = \frac{1}{n_F} \sum_F \phi(\mathbf{x}_F) \quad (8.45)$$

and

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}([q', x_F]_{<}) I_{\ell m}^*([q', x_F]_{>}) m(q'), \quad (8.46)$$

where  $\mathbf{x}_F$  is the cell face radial distance from the expansion center and  $[q', x_F]_{<}$  denotes the larger of the magnitudes between the cell center radial distance  $q'$  and the cell face radial distance  $x_F$ . Splitting the summation over cells in two parts

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left\{ \sum_{q' \leq x_F} \sum_{\ell m} [R_{\ell m}(q') m(q')] I_{\ell m}^*(\mathbf{x}_F) + \sum_{q' > x_F} \sum_{\ell m} R_{\ell m}(\mathbf{x}_F) [I_{\ell m}^*(q') m(q')] \right\}, \quad (8.47)$$

and defining the two moments

$$M_{\ell m}^R(\mathbf{x}_F) = \sum_{q' \leq x_F} R_{\ell m}(q') m(q') \quad (8.48)$$

$$M_{\ell m}^I(\mathbf{x}_F) = \sum_{q' > x_F} I_{\ell m}(q') m(q'), \quad (8.49)$$

we obtain

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left[ \sum_{\ell m} M_{\ell m}^R(\mathbf{x}_F) I_{\ell m}^*(\mathbf{x}_F) + \sum_{\ell m} M_{\ell m}^{I*}(\mathbf{x}_F) R_{\ell m}(\mathbf{x}_F) \right] \quad (8.50)$$

and using vector notation

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} [\mathbf{M}^R(\mathbf{x}_F) \cdot \mathbf{I}^*(\mathbf{x}_F) + \mathbf{M}^{I*}(\mathbf{x}_F) \cdot \mathbf{R}(\mathbf{x}_F)] . \quad (8.51)$$

We now change from complex to real formulation. We state this for the regular solid harmonic functions, the same reasoning being applied to the irregular solid harmonic functions and all their derived moments. The regular solid harmonic functions can be split into a real and imaginary part

$$R_{\ell m} = R_{\ell m}^c + i R_{\ell m}^s. \quad (8.52)$$

The labels 'c' and 's' are shorthand notations for 'cosine' and 'sine', reflecting the nature of the azimuthal function of the corresponding real spherical harmonics. When inserting (8.52) into (8.51) all cosine and sine mixed terms of the scalar products cancel out. Also, due to the symmetry relations

$$R_{\ell, -m}^c = (-1)^m R_{\ell m}^c \quad (8.53)$$

$$R_{\ell, -m}^s = -(-1)^m R_{\ell m}^s \quad (8.54)$$

we can restrict ourselves to the following polar angle number ranges

$$c : \ell \geq 0, \ell \geq m \geq 0 \quad (8.55)$$

$$s : \ell \geq 1, \ell \geq m \geq 1. \quad (8.56)$$

The real formulation of (8.51) becomes then

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left\{ \begin{bmatrix} \mathbf{M}^{Rc}(\mathbf{x}_F) \\ \mathbf{M}^{Rs}(\mathbf{x}_F) \end{bmatrix} \cdot \Delta \begin{bmatrix} \mathbf{I}^c(\mathbf{x}_F) \\ \mathbf{I}^s(\mathbf{x}_F) \end{bmatrix} + \begin{bmatrix} \mathbf{M}^{Ic}(\mathbf{x}_F) \\ \mathbf{M}^{Is}(\mathbf{x}_F) \end{bmatrix} \cdot \Delta \begin{bmatrix} \mathbf{R}^c(\mathbf{x}_F) \\ \mathbf{R}^s(\mathbf{x}_F) \end{bmatrix} \right\}, \quad (8.57)$$

which, when compared to (8.51), shows, that all vectors now contain a cosine and a sine section. The  $\Delta$  matrix is a diagonal matrix whose elements are equal to 2 for  $m \neq 0$  and 1 otherwise, i.e.:

$$\Delta = \text{diag}(2 - \delta_{m0}). \quad (8.58)$$

The recursion relations for calculating the solid harmonic vectors are

$$R_{00}^c = 1 \quad (8.59)$$

$$R_{\ell\ell}^c = -\frac{xR_{\ell-1,\ell-1}^c - yR_{\ell-1,\ell-1}^s}{2\ell} \quad (8.60)$$

$$R_{\ell\ell}^s = -\frac{yR_{\ell-1,\ell-1}^c + xR_{\ell-1,\ell-1}^s}{2\ell} \quad (8.61)$$

$$R_{\ell m}^{c/s} = \frac{(2\ell-1)zR_{\ell-1,m}^{c/s} - r^2R_{\ell-2,m}^{c/s}}{(\ell+m)(\ell-m)}, \quad 0 \leq m < \ell \quad (8.62)$$

and

$$I_{00}^c = \frac{1}{r} \quad (8.63)$$

$$I_{\ell\ell}^c = -(2\ell-1) \frac{xI_{\ell-1,\ell-1}^c - yI_{\ell-1,\ell-1}^s}{r^2} \quad (8.64)$$

$$I_{\ell\ell}^s = -(2\ell-1) \frac{yI_{\ell-1,\ell-1}^c + xI_{\ell-1,\ell-1}^s}{r^2} \quad (8.65)$$

$$I_{\ell m}^{c/s} = \frac{(2\ell-1)zI_{\ell-1,m}^{c/s} - [(\ell-1)^2 - m^2]I_{\ell-2,m}^{c/s}}{r^2}, \quad 0 \leq m < \ell \quad (8.66)$$

in which  $x, y, z$  are the cartesian location coordinates of the cell face and  $r^2 = x^2 + y^2 + z^2$ . For geometries depending on polar angles one must first calculate the corresponding cartesian coordinates for each cell before applying the recursions. In FLASH, the order of the two cosine and sine components for each solid harmonic vector is such that  $\ell$  precedes  $m$ . This allows buildup of the vectors with maximum number of unit strides. The same applies of course for the assembly of the moments. For 2D cylindrical and 2D spherical geometries only the  $m = 0$  parts of both recursions are needed, involving only the cartesian  $z$  coordinate and  $r^2$ . Symmetry along the radial axes of these 2D geometries inflicts only the sign change  $z \rightarrow -z$ , resulting in the symmetry relations  $R_{\ell 0}^c \rightarrow R_{\ell 0}^c$  for even  $\ell$  and  $R_{\ell 0}^c \rightarrow -R_{\ell 0}^c$  for odd  $\ell$ , the same holding for the irregular solid harmonic vector components. Thus symmetry in 2D can effectively be treated by halving the domain size and multiplying each even  $\ell$  moments by a factor of 2 while setting the odd  $\ell$  moments equal to 0. For 3D cartesian geometries introduction of symmetry is far more complicated since all  $m$  components need to be taken into account. It is not sufficient to simply reduce the domain to the appropriate size and multiply the moments by some factor, but rather one would have to specify the exact symmetry operations intended (generators of the symmetry group  $O_h$  or one of its subgroups) in terms of their effects on the  $x, y, z$  cartesian basis. The resulting complications in calculating symmetry adapted moments outweighs the computational gain that can be obtained from it. Options for 3D symmetry are thus no longer available in the improved FLASH multipole solver. The 'octant' symmetry option from the old multipole solver, using only the monopole  $\ell = 0$  term, was too restrictive in its applicability (exact only for up to angular momenta  $\ell = 3$  due to cancellation of the solid harmonic vector components).

From the above recursion relations (8.59-8.66), the solid harmonic vector components are functions of  $x^i y^j z^k$  monomials, where  $i + j + k = \ell$  for the  $\mathbf{R}$  and (formally)  $i + j + k = -(\ell + 1)$  for the  $\mathbf{I}$ . For large astrophysical coordinates and large  $\ell$  values this leads to potential computational over- and underflow. To get independent of the size of both the coordinates and  $\ell$  we introduce a damping factor  $Dx, Dy, Dz$  for the coordinates for each solid harmonic type before entering the recursions.  $D$  will be chosen such that for the highest specified  $\ell = L$  we will have approximately a value close to 1 for both solid harmonic components:

$$R_{Lm}^{c/s} \approx 1 \quad (8.67)$$

$$I_{Lm}^{c/s} \approx 1. \quad (8.68)$$

This ensures proper handling of size at the solid harmonic function evaluation level and one does not have to rely on size cancellations at a later stage when evaluating the potential via Eq.(8.57). We next state the evaluation of the damping factor  $D$ . Due to the complicated nature of the recursions, the first step is to find solid harmonic components which have a simple structure. To do this, consider a cell face with  $x, y = 0$  and  $z \neq 0$ . Then  $r^2 = z^2$ ,  $|z| = r$  and only the  $m = 0$  components are different from zero. An explicit form can be stated for the absolute values of these components in terms of  $r$ :

$$|R_{\ell 0}| = \frac{r^\ell}{\ell!} \quad (8.69)$$

$$|I_{\ell 0}| = \frac{\ell!}{r^{\ell+1}}. \quad (8.70)$$

Since  $r = \sqrt{x^2 + y^2 + z^2}$ , damping of the coordinates with  $D$  results in a damped radial cell face distance  $D r$ . Inserting this result into (8.69) and (8.70) and imposing conditions (8.67) and (8.68) results in

$$D_R = \frac{1}{r} \sqrt[\ell]{L!} \approx \frac{1}{r} \frac{L}{e} \sqrt[2L]{2\pi L} \quad (8.71)$$

$$D_I = \frac{1}{r} \sqrt[\ell+1]{L!} \approx \frac{1}{r} \frac{L}{e} \sqrt[2L+2]{\frac{2\pi e^2}{L}}, \quad (8.72)$$

where the approximate forms are obtained by using Stirling's factorial approximation formula for large  $L$ . In FLASH only the approximate forms are computed for  $D_R$  and  $D_I$  to avoid having to deal with factorials of large numbers.

From the moment defining equations (8.48) and (8.49) we see, that the moments are sums over subsets of cell center solid harmonic vectors multiplied by the corresponding cell mass. From Eq.(8.57) it follows that for highest accuracy, the moments should be calculated and stored for each possible cell face. For high

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

$$\mathbf{M}^R(Q) = \sum_{q \leq Q} \mathbf{R}(q)m(q) \quad (8.73)$$

$$\mathbf{M}^I(Q) = \sum_{q \geq Q} \mathbf{I}(q)m(q), \quad (8.74)$$

where  $q \leq Q$  means including all cells belonging to  $Q$  and all radial bins with lower radial boundaries than  $Q$ . The two basic operations of the multipole solver are thus: i) assembly of the radial bin moments and ii) formation of the scalar products via Eq.(8.57) to obtain the potentials. The memory declaration of the moment array should reflect the way the individual moment components are addressed and the most efficient layout puts the angular momentum indices in rows and the radial bin indices in columns.

How do we extract moments  $\mathbf{M}^R(\mathbf{x})$  and  $\mathbf{M}^I(\mathbf{x})$  at any particular position  $\mathbf{x}$  inside the domain (and, in particular, at the cell face positions  $\mathbf{x}_F$ ), which are ultimately needed for the potential evaluation at that location? Assume that the location  $\mathbf{x}$  corresponds to a particular radial bin  $\mathbf{x} \rightarrow Q$ . Consider the three consecutive radial bins  $Q-1$ ,  $Q$  and  $Q+1$ , together with their calculated moments:

$$\begin{array}{c|c|c} \mathbf{M}^R(Q-1) & \mathbf{M}^R(Q) & \mathbf{M}^R(Q+1) \\ \hline \mathbf{M}^I(Q-1) & \mathbf{M}^I(Q) & \mathbf{M}^I(Q+1) \end{array} \quad (8.75)$$

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

$$\mathbf{M}^R(\mathbf{x}) = \mathbf{M}^R(Q-1) + R_{frac} [\mathbf{M}^R(Q) - \mathbf{M}^R(Q-1)] \quad (8.76)$$

$$\mathbf{M}^I(\mathbf{x}) = \mathbf{M}^I(Q+1) + I_{frac} [\mathbf{M}^I(Q) - \mathbf{M}^I(Q+1)], \quad (8.77)$$

The extraction of the moments via (8.76) and (8.77) is of course an approximation that relies on the statistically dense distribution of the individual cell center moments inside each radial bin. For bins which are reasonably far away from the expansion center this statistical approximation is valid but for those close to the expansion center the statistical distribution does not hold and calculating the moments via the above scheme introduces a large statistical error. The way out of this problem is to move from a statistical radial bin description around the expansion center to a more discrete one, by constructing very narrow isolated radial bins. The code is thus forced to analyze the detailed structure of the geometrical domain grid surrounding the expansion center and to establish the inner radial zone of discrete distributed radial bins. The statistical radial bins are then referred to as belonging to the outer radial zone(s).

While the structure of the inner radial zone is fixed due to the underlying geometrical grid, the size of each radial bin in the outer radial zones has to be specified by the user. There is at the moment no automatic derivation of the optimum (accuracy vs memory cost) bin size for the outer zones. There are two types of radial bin sizes defined for the FLASH multipole solver: i) exponentially and/or ii) logarithmically growing:

$$\text{exponential bin size upper radial limit} = s \cdot \Delta r \cdot Q^t \quad (8.78)$$

$$\text{logarithmic bin size upper radial limit} = s \cdot \Delta r \cdot \frac{e^{tQ} - 1}{e^t - 1}. \quad (8.79)$$

In these definitions,  $\Delta r$  is a small distance 'atomic' (basic unit) radial measure, defined as half the geometric mean of appropriate cell dimensions at highest refinement level,  $s$  is a scalar factor to optionally increase or decrease the atomic unit radial measure and  $Q = 1, 2, \dots$  is a local bin index counter for each outer zone.



The atomic radial distance  $\Delta r$  is calculated for each individual domain geometry as follows:

Domain Geometry	$\Delta r$
3D cartesian	$\frac{1}{2} \sqrt[3]{\Delta x \Delta y \Delta z}$
3D cylindrical	$\frac{1}{2} \sqrt{\Delta R \Delta z}$
2D cylindrical	$\frac{1}{2} \sqrt{\Delta R \Delta z}$
2D spherical	$\frac{1}{2} \Delta R$
1D spherical	$\frac{1}{2} \Delta R$

(8.80)

where  $\Delta x, \Delta y, \Delta z$  are the usual cartesian cell dimensions and  $\Delta R$  is the radial cell dimension. Note, that since  $\Delta r$  measures a basic radial unit along the radial distance from the expansion center (which, for approximate spherical problems, is located close to the domain's geometrical origin), only those cell dimensions for calculating each  $\Delta r$  are taken, which are directly related to radial distances from the geometrical domain origin. For 3D cylindrical domain geometries for example, only the radial cylindrical and z-coordinate cell dimensions determine the 3D radial distance from the 3D cylindrical domain origin. The angular coordinate is not needed. Likewise for spherical domains only the radial cell coordinate is of importance. Definitions (8.78) and (8.79) define the upper limit of the radial bins. Hence in order to obtain the true bin size for the  $Q$ -th bin one has to subtract its upper radial limit from the corresponding one of the  $(Q - 1)$ -th bin:

$$Q\text{-th exponential bin size} = s \cdot \Delta r \cdot [Q^t - (Q - 1)^t] \quad (8.81)$$

$$Q\text{-th logarithmic bin size} = s \cdot \Delta r \cdot e^{t(Q-1)}. \quad (8.82)$$

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

Multithreading of the code is currently enabled in two parts: 1) during moment evaluation and 2) during potential evaluation. The threading in the moment evaluation section is achieved by running multiple threads over separate, non-conflicting radial bin sections. Moment evaluation is thus organized as a single loop over all relevant radial bins on each processor. Threading over the potential evaluation is done over blocks, as these will address different non-conflicting areas of the solution vector.

The improved multipole solver was extensively tested and several runs have been performed using large domains ( $> 10^{10}$ ) and extremely high angular numbers up to  $L = 100$  for a variety of domain geometries. Currently, the following geometries can be handled: 3D cartesian, 3D cylindrical, 2D cylindrical, 2D spherical and 1D spherical. The structure of the code is such that addition of new geometries, should they ever be needed by some applications, can be done rapidly.

### 8.10.2.3 Multipole Poisson solver unit test (MacLaurin spheroid)

The first unit test for the multipole Poisson solver is based on the MacLaurin spheroid analytical gravitational solution given in section 35.3.4. The unit test sets up a spheroid with uniform unit density and determines both the analytical and numerical gravitational fields. The absolute relative error based on the analytical solution is formed for each cell and the maximum of all the absolute errors is compared to a predefined error tolerance value for a particular uniform refinement level. The multipole unit test runs in 2D cylindrical, 2D spherical, 3D cartesian and 3D cylindrical geometries and threaded multipole unit tests are also available.

### 8.10.2.4 Tree Poisson Solver

The tree solver is based on the Barnes & Hut (1986, Nature, 324, 446) tree code for calculation of gravity forces in N-body simulations. However, it is more general and it includes some more modern features described for instance in Salmon & Warren (1994, J. Comp. Phys, 136, 155), Springel (2005, MNRAS, 364, 1105) and other works. It builds a global octal tree over the whole computational domain, communicates its part to all processors and uses it for calculations of various physical problems provided as separate units in source/physics directory. The global tree is an extension of the AMR mesh octal tree down to individual cells. The communication of the tree is implemented so that only parts of the tree that are needed for the calculation of the potential on a given processor are sent to it. The calculation of physical problems is done by walking the tree for each grid cell (hereafter *point-of-calculation*) and evaluating whether the tree node should be used for calculation or whether its children should be open.



The tree solver is connected to physical units by several wrapper subroutines that are called at specific places of the tree build and tree walk, and that call corresponding subroutines of physical units. In this way, physical units can include arbitrary quantity into the tree, and then, use it to calculate some other physical quantity by integrating contributions of all tree nodes during the tree walk. In this version, the only working unit implementation is `physics/Gravity/GravityMain/Poisson/BHTree`, which calculates the gravitational potential.

The tree solver algorithm consists of four parts. The first one, communication of block properties, is called only if the AMR grid changes. The other three, building of the tree, communication of the tree and calculation of the potential, are called in each time-step.

*Communication of block properties.* In recent version, each processor needs to know some basic information about all blocks in the simulation. It includes arrays: `nodetype`, `lrefine` and `child`. These arrays are distributed from each processor to all the other processors. They can occupy a substantial amount of memory on large number of processors (memory required for statically allocated arrays of the tree solver can be calculated by a script `tree_mem_use.py`).

*Building the tree.* The global tree is constructed from bottom and the process consists of three steps. In the first one, the so-called *block-tree* is constructed in each leaf block on a given processor. The *block-trees* are 1-dimensional dynamically allocated arrays (see Figure 8.12) and pointers to them are stored in array `gr_bhTreeArray`. In the second step, top nodes of block-trees (corresponding to whole blocks) are distributed to all processors and stored in array `gr_bhTreeParentTree`. In the last step, higher nodes of the parent tree are calculated by each processor and stored in the `gr_bhTreeParentTree` array. At the end, each processor holds information about the global tree down to the level of leaf blocks. During the whole process of tree building, 5 subroutines providing the interface to physical units are called: `gr_bhFillBotNode`, `gr_bhAccBotNode`, `gr_bhAccNode`, `gr_bhNormalizeNode` and `gr_bhPostprocNode` (see their auto-documentation and source code for details). Each of them calls a corresponding subroutine of all physical units with a name where the first two letters 'gr' are replaced with the name of the unit (e.g. `Gravity_bhFillBotNode`).

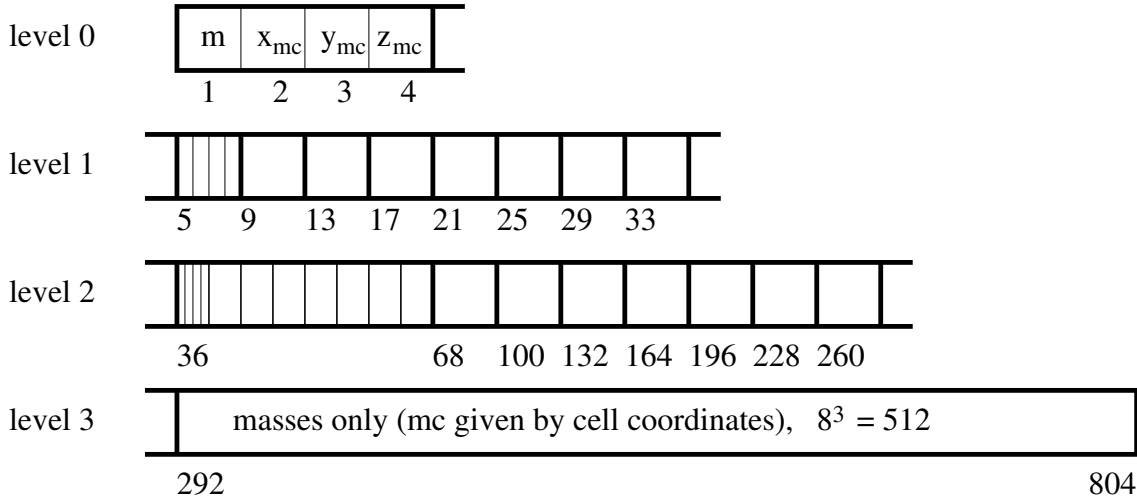


Figure 8.12: Example of a block-tree in case of `nxb=nyb=nzb=8` and in case physical units do not store any further information to tree nodes (masses and mass centre positions are included by the tree solver itself).

*Communication of the tree.* Most of the tree data is contained on the bottom levels in individual *block-trees*. In order to save memory and communication time, only parts of *block-trees* that are needed on a given processor are sent to it. The procedure consists of three steps. In the first one, a level down to which each *block-tree* has to be sent to each processor is determined. For a given *block-tree*, it is done by evaluating the criterion for the node acceptance (traditionally called multipole acceptance criterion, shortly MAC) for all blocks on a remote processor, searching for the maximum level down to which the evaluated node will be needed on a given remote processor. In the second step, information about the *block-tree* levels which are going to be communicated is sent to all processors. This information is needed for allocation of arrays in

which *block-trees* are stored on remote processors. In the third step, the *block-tree* arrays are allocated, all *block-trees* for a given processor are packed into a single message and the messages are communicated.

The MAC is implemented in subroutine `gr_bhMAC` which includes only a simple geometrical MAC used also by Barnes & Hut. The node is accepted for calculation if

$$\frac{S_{\text{node}}}{D} < \text{gr\_bhTreeLimAngle} , \quad (8.83)$$

where  $S_{\text{node}}$  is the node size (defined as the largest edge of the corresponding cuboid) and  $D$  is the distance between the node and the *point-of-calculation*. Additionally, `gr_bhMAC` checks that the *point-of-calculation* is not located within the node itself enlarged by factor `gr_bhTreeSafeBox`. On the top of that, `gr_bhMAC` calls MACs of physical units and the node is accepted only if all criteria are fulfilled.

*Tree walk.* The tree is traversed from the top to the bottom, evaluating MAC of each node and in case it is not fulfilled, continuing the tree walk with its children. If the node's MAC is fulfilled, the node is accepted for the calculation and subroutine `gr_bhBotNodeContrib` or `gr_bhNodeContrib` is called, depending on whether it is a bottom-most node (i.e. a single grid cell) or higher node, respectively. These subroutines only call the corresponding subroutines of physical units (e.g., `Gravity_bhNodeContrib`). This is the most CPU-intensive part of the tree solver, it usually takes more than 90% of the total tree solver time. It is completely parallel and it does not include any communication (apart from sending some statistics to the main processor at the end).

The tree solver includes several implementations of the tree walk. The default algorithm is the Barnes-Hut like tree walk in which the whole tree is traversed from the top down to nodes fulfilling MAC for each cell separately. This algorithm is used in case the runtime parameter `gr_bhUnifiedTreeWalk` is true (default). If it this parameter is set to false, another algorithm is used in which instead of walking the whole tree for each cell individually, MAC is at first evaluated for whole mesh block (interacting with some node). If the node is accepted and if the node is a parent node (i.e. corresponding to whole mesh block), the node is accepted for all cells of the block and the contribution of the node is added to them. However, the node contribution is calculated separately for each cell, because the distance between the node and individual cells differs. The third tree walk algorithm is an implementation of the so called **SumSquare** MAC described by Salmon & Warren (1994). The tree is traversed using the priority queue, taking contribution of the most important nodes first. This algorithm provides much better error control, however, the implementation in this code version is highly experimental.

The tree solver supports isolated and periodic boundary conditions that can be set independently in each direction. In the latter case, when a node is considered for MAC evaluation and eventually calculation by calling `Gravity_bhNodeContrib`, periodic copies of the node are checked, and the minimum distance among the node periodic copies is taken in account. This allows for instance to calculate gravitational potential with periodic boundary conditions using the Ewald method (see description of the Gravity unit).

#### 8.10.2.5 Tree Poisson solver unit test

The unit test for the tree gravity solver calculates the gravitational potential of the Bonnor-Ebert sphere (Bonnor, W. B., 1956, MNRAS, 116, 351) and compares it to the analytical potential. The density distribution and the analytical potential are calculated by the python script `bes-generator.py`. The simulation setup only reads the file with radial profiles of these quantities and sets it on the grid. It also normalizes the analytical potential (adds a constant to it) so that the minimum values of the analytical and numerical potential are the same. The error of the gravitational potential calculated by the tree code is stored in the field array PERR (written into the PlotFile). The maximum absolute and relative errors are written into the log file.

#### 8.10.2.6 Multigrid Poisson solver

This section of the User's Guide is taken from a paper by Paul Ricker, "A Direct Multigrid Poisson Solver for Oct-Tree Adaptive Meshes" (2008). 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, 2008) requires a few modifications. The original formulation required that there be shared points between the coarse and fine patches. Contrast this with finite-volume, nested-cell, cell-averaged grids as used in FLASH (Figure 8.13). This is overcome by the exchange of guardcells from coarse to fine using monotonic interpolation (Section 8.6.2) and external boundary extrapolation for the calculation of the residual.

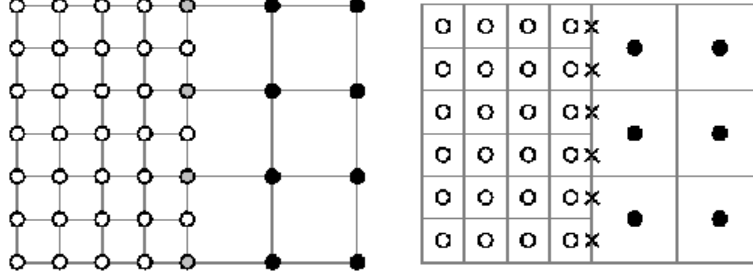


Figure 8.13: Contrast between jumps of refinement in meshes used in the original paper (left) and the oct-tree adapted method (right).

Another difference between the method of (Ricker 2008) 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:

$$R(\mathbf{x}) \equiv 4\pi G\rho(\mathbf{x}) - \nabla^2 \tilde{\phi}(\mathbf{x}) . \quad (8.84)$$

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

$$\mathcal{D}_\ell \tilde{\phi}_{ijk}^{b\ell} \equiv \frac{1}{\Delta x_\ell^2} \left( \tilde{\phi}_{i+1,jk}^{b\ell} - 2\tilde{\phi}_{ijk}^{b\ell} + \tilde{\phi}_{i-1,jk}^{b\ell} \right) + \frac{1}{\Delta y_\ell^2} \left( \tilde{\phi}_{i,j+1,k}^{b\ell} - 2\tilde{\phi}_{ijk}^{b\ell} + \tilde{\phi}_{i,j-1,k}^{b\ell} \right) \quad (8.85)$$

$$+ \frac{1}{\Delta z_\ell^2} \left( \tilde{\phi}_{ij,k+1}^{b\ell} - 2\tilde{\phi}_{ijk}^{b\ell} + \tilde{\phi}_{ij,k-1}^{b\ell} \right) . \quad (8.86)$$

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

$$(\mathcal{R}_\ell \tilde{\phi})_{ijk}^{\mathcal{P}(c),\ell} \equiv \frac{1}{2^d} \sum_{i'j'k'} \tilde{\phi}_{i'j'k'}^{c,\ell+1} , \quad (8.87)$$

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  $\mathcal{I}_\ell$ ,

need only be defined on the edges of blocks, because it is only used to set boundary values for the direct single-block Poisson solver:

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

When needed, boundary zone values are set as for the difference operator. We use conservative quartic interpolation to set edge values, then solve with homogeneous Dirichlet boundary conditions after using second-order boundary-value elimination. The coefficients  $\alpha$  determine the interpolation scheme. For the  $-x$  face in 3D,

$$\begin{aligned} \alpha_{1/2,j'k'pqr} &= \beta_p \gamma_{j'q} \gamma_{k'r} \\ (\beta_p) &= \left( -\frac{1}{12}, \frac{7}{12}, \frac{7}{12}, -\frac{1}{12}, 0 \right) \\ (\gamma_{j'q}) &= \begin{cases} \left( -\frac{3}{128}, \frac{11}{64}, 1, -\frac{11}{64}, \frac{3}{128} \right) & j' \text{ odd} \\ \left( \frac{3}{128}, -\frac{11}{64}, 1, \frac{11}{64}, -\frac{3}{128} \right) & j' \text{ even} \end{cases} \end{aligned} \quad (8.89)$$

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

### 8.10.2.7 The direct solver

In the case of problems with Dirichlet boundary conditions, a  $d$ -dimensional fast sine transform is used. The transform-space Green's Function for this is:

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

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

$$\rho_{1jk} \rightarrow \rho_{1jk} - \frac{2}{\Delta x_\ell^2} \phi(x_{1/2}, y_j, z_k). \quad (8.91)$$

For periodic problems only the coarsest block must be handled differently; block adjacency for finer levels is handled naturally. The periodic solver uses a real-to-complex FFT with the Green's function:

$$G_{ijk}^\ell = \begin{cases} -16\pi G \left[ \frac{1}{\Delta x_\ell^2} \sin^2 \left( \frac{(i-1)\pi}{n_x} \right) + \frac{1}{\Delta y_\ell^2} \sin^2 \left( \frac{(j-1)\pi}{n_y} \right) + \frac{1}{\Delta z_\ell^2} \sin^2 \left( \frac{(k-1)\pi}{n_z} \right) \right]^{-1} & i, j, \text{ or } k \neq 1 \\ 0 & i = j = k = 1 \end{cases} \quad (8.92)$$

This solve requires that the source be zero-averaged; otherwise the solution is non-unique. Therefore the source average is subtracted from all blocks. In order to decimate error across same-refinement-level boundaries, Gauss-Seidel relaxations to the outer two layers of zones in each block are done after applying the direct solver to all blocks on a level. With all these components outlined, the overall solve may be described by the following algorithm:

1. Restrict the source function  $4\pi G\rho$  to all levels. Subtract the global average for the **periodic** case.
2. *Interpolation step:* For  $\ell$  from 1 to  $\ell_{\max}$ ,

- (a) Reset the grid so that  $\ell$  is the maximum refinement level
  - (b) Solve  $\mathcal{D}_\ell \tilde{\phi}_{ijk}^{b\ell} = 4\pi G \rho_{ijk}^{b\ell}$  for all blocks  $b$  on level  $\ell$ .
  - (c) Compute the residual  $R_{ijk}^{b\ell} = 4\pi G \rho_{ijk}^{b\ell} - \mathcal{D}_\ell \tilde{\phi}_{ijk}^{b\ell}$
  - (d) For each block  $b$  on level  $\ell$  that has children, prolong face values for  $\tilde{\phi}_{ijk}^{b\ell}$  onto each child block.
3. *Residual propagation step:* Restrict the residual  $R_{ijk}^{b\ell}$  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  $\ell_{\max}$ ,
- (a) Reset the grid so that  $\ell$  is the maximum refinement level
  - (b) Solve  $\mathcal{D}_\ell C_{ijk}^{b\ell} = R_{ijk}^{b\ell}$  for all blocks  $b$  on level  $\ell$ .
  - (c) Overwrite  $R_{ijk}^{b\ell}$  with the new residual  $R_{ijk}^{b\ell} - \mathcal{D}_\ell C_{ijk}^{b\ell}$  for all blocks  $b$  on level  $\ell$ .
  - (d) Correct the solution on all leaf-node blocks  $b$  on level  $\ell$ :  $\tilde{\phi}_{ijk}^{b\ell} \rightarrow \tilde{\phi}_{ijk}^{b\ell} + C_{ijk}^{b\ell}$ .
  - (e) For each block  $b$  on level  $\ell$  that has children, interpolate face boundary values of  $C_{ijk}^{b\ell}$  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.

### 8.10.2.8 A Hybrid Poisson Solver: Interfacing PFFT with Multigrid

We can improve the performance of the Multigrid solver in Section 8.10.2.6 by replacing single block FFTs with a parallel FFT at a specified coarse level, where, the coarse level is any level which is fully refined, i.e. containing blocks that completely cover the computational domain. Currently, we automatically select the maximum refinement level that is fully refined.

There is load imbalance in the Multigrid solver because each processor performs single block FFTs on the blocks it owns. At the coarse levels there are relatively few blocks compared to available processors which means many processors are effectively idle during the coarse level solves. The introduction of PFFT, and creation of a hybrid solver, eliminates some of the coarse level solves.

The performance characteristics of the hybrid solver are described in “Optimization of multigrid based elliptic solver for large scale simulations in the FLASH code” (2012) which is available online at <http://onlinelibrary.wiley.com/doi/10.1002/cpe.2821/pdf>. Performance results are obtained using the PFFT\_PoissonFD unit test.

### 8.10.3 Using the Poisson solvers

The `GridSolvers` subunit solves the Poisson equation ((8.9)). Two different elliptic solvers are supplied with FLASH: a multipole solver, suitable for approximately spherical source distributions, and a multigrid solver, which can be used with general source distributions. The multipole solver accepts only isolated boundary conditions, whereas the multigrid solver supports Dirichlet, given-value, Neumann, periodic, and isolated boundary conditions. Boundary conditions for the Poisson solver are specified using an argument to the `Grid_solvePoisson` routine which can be set from different runtime parameters depending on the physical context in which the Poisson equation is being solved. The `Grid_solvePoisson` routine is the primary entry point to the Poisson solver module and has the following interface

```
call Grid_solvePoisson (iSoln, iSrc, bcTypes(6), bcValues(2,6), poisfact) ,
```

Table 8.3: Runtime parameters used with `poisson/multipole`.

Variable	Type	Default	Description
<code>mpole_lmax</code>	integer	10	Maximum multipole moment
<code>quadrant</code>	logical	<code>.false.</code>	Use symmetry to solve a single quadrant in 2D axisymmetric cylindrical $(r, z)$ coordinates, instead of a half domain.

where *iSoln* and *iSrc* are the integer-valued indices of the solution and source (density) variables, respectively. *bcTypes*(6) is an integer array specifying the type of boundary conditions to employ on each of the (up to) 6 sides of the domain. Index 1 corresponds to the -x side of the domain, 2 to +x, 3 to -y, 4 to +y, 5 to -z, and 6 to +z. The following values are accepted in the array

<i>bcTypes</i>	Type of boundary condition
0	Isolated boundaries
1	Periodic boundaries
2	Dirichlet boundaries
3	Neumann boundaries
4	Given-value boundaries

Not all boundary types are supported by all solvers. In this release, *bcValues*(2,6) is not used and can be filled arbitrarily. Given-value boundaries are treated as Dirichlet boundaries with the boundary values subtracted from the outermost interior cells of the source; for this case the solution variable should contain the boundary values in its first layer of boundary cells on input to `Grid_solvePoisson`. It should be noted that if `PARAMESH` is used, the values must be set for all levels. Finally, *poisfact* is real-valued and indicates the value of  $\alpha$  multiplying the source function in ((8.9)).

When solutions found using the Poisson solvers are to be differenced (*e.g.*, in computing the gravitational acceleration), it is strongly recommended that for AMR meshes, quadratic (or better) spatial interpolation at fine-coarse boundaries is chosen. (For `PARAMESH`, this is automatically the case by default, and is handled correctly for Cartesian as well as the supported curvilinear geometries. But note that the default interpolation implementation may be changed at configuration time with the `'-gridinterpolation=...'` setup option; and with the default implementation, the interpolation order may be lowered with the `interpol_order` runtime parameter.) If the order of the gridinterpolation of the mesh is not of at least the same order as the differencing scheme used in places like `Gravity_accelOneRow`, unphysical forces will be produced at refinement boundaries. Also, using constant or linear grid interpolation may cause the multigrid solver to fail to converge.

### 8.10.3.1 Multipole (original version)

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

### 8.10.3.2 Multipole (improved version)

To include the new multipole solver in a simulation, the best option is to use the shortcut `+newMpole` at setup command line, effectively replacing the following setup options :

```
-with-unit=Grid/GridSolvers/Multipole_new
-with-unit=physics/Gravity/GravityMain/Poisson/Multipole
-without-unit=Grid/GridSolvers/Multipole
```

The improved multipole solver currently accepts only two setup parameters, either one switching on multi-threading:

- **threadBlockList**: enables multithreaded compilation and execution.
- **threadWithinBlock**: enables multithreaded compilation and execution.

The names of these two setup parameters are misleading, since there is only one universal threading strategy used. The use of these two setup parameters is a temporary solution and will be replaced in near future by only one setup parameter.

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

- **mpole\_Lmax**: The maximum angular moment  $L$  to be used for the multipole Poisson solver. Depending on the domain geometry, the memory and time scaling factors due to this variable alone are: i) 3D cartesian, 3D cylindrical  $\rightarrow (L+1)(L+1)$ , ii) 3D cartesian axisymmetric, 2D cylindrical, 2D spherical  $\rightarrow (L+1)$ , iii) 1D spherical  $\rightarrow 1$ . Assuming no memory limitations, the multipole solver is numerically stable for very large  $L$  values. Runs up to  $L = 100$  for 3D cartesian domains have been performed. For 2D geometries,  $L = 1000$  was the maximum tested.
- **mpole\_2DSymmetryPlane**: In 2D coordinates, this runtime parameter enables the user to specify a plane of symmetry along the radial part of the domain coordinates. In effect, this allows a reduction of the computational domain size by one half. The code internally computes the multipole moments as if the other symmetric part is present, i.e. no memory or execution time savings can be achieved by this runtime parameter.
- **mpole\_3DAxisymmetry**: Forces rotational invariance around the main ( $z$ ) axis in 3D cartesian domains. The assumed rotational invariance in the  $(x, y)$  plane effectively cancels all  $m \neq 0$  multipole moments and one can restrict the calculation to the  $m = 0$  multipole moments only. The time and memory savings compared to a asymmetric 3D cartesian run is thus about a factor of  $(L+1)$ . For 3D cylindrical domains, rotational invariance in the  $(x, y)$  plane is equivalent of setting up the corresponding 2D cylindrical domain, hence this runtime parameter is not honored for 3D cylindrical domains, and the user is informed about the 3D to 2D cylindrical domain reduction possibility.
- **mpole\_DumpMoments**: This parameter is meant mainly for debugging purposes. It prints the entire moment array for each radial bin for each time step. This option should be used with care and for small problems only. The output is printed to a text file named '`<basenm>_dumpMoments.txt`', where `<basenm>` is the base name given for the output files.
- **mpole\_PrintRadialInfo**: This parameter enables showing all detailed radial bin information at each time step. This option is especially useful for optimizing the radial bin sizes. The output is written to the text file '`<basenm>_printRadialInfo.txt`'.
- **mpole\_IgnoreInnerZone**: Controls switching on/off the radial inner zone. If it is set `.true.`, the inner zone will not be recognized and all inner zone radii will be treated statistically. This parameter is meant only for performing some error analysis. For production runs it should always be at its default value of false. Otherwise errors will be introduced in calculating the moments near the expansion center.
- **mpole\_InnerZoneSize**: The size defining the discrete inner zone. The size is given in terms of the inner zone smallest (atomic) radius, which is determined at each time step by analyzing the domain grid structure around the multipolar origin (expansion center). Only very rarely will this value ever have to be changed. The default setting is very conservative and only under unusual circumstances (ex: highly nonuniform grid around the expansion center) this might be necessary. This value needs to be an integer, as it is used by the code to define dimensions of certain arrays. Note, that by giving this runtime parameter a large integer value ( $\geq 1000$  for domain refinement levels up to 5) one can enforce the code to use only non-statistical radial bins.
- **mpole\_InnerZoneResolution**: Defines the inner zone radial bin size for the inner zone in terms of the inner zone smallest (atomic) radius. Two inner zone radii will be considered different, if they are more



than this resolution value apart. A very tiny number (for example  $10^{-8}$ ) will result in a complete separation of all inner zone radii into separate radial bins. The default value of 0.1 should never be surpassed, and any attempt to do so will stop the program with the appropriate information to the user. Likewise with a meaningless resolution value of 0.

- **mpole\_MaxRadialZones**: The maximum number of outer radial zones to be used. In contrast to the inner radial zone, the outer radial zones are much more important for the user. Their layout defines the performance of the multipole solver both in cpu time spent and accuracy of the potential obtained at each cell. The default value of 1 outer radial zone at maximum refinement level leads to high accuracy, but at the same time can consume quite a bit of memory, especially for full 3D runs. In these cases the user can specify several outer radial zones each having their own radial bin size determination rule.
- **mpole\_ZoneRadiusFraction\_n**: The fraction of the maximum domain radius defining the n-th outer zone maximum radial value. The total number of fractions given must match the maximum number of outer radial zones specified and the fractions must be in increasing order and less than unity as we move from the 1st outer zone upwards. The last outer zone must always have a fraction of exactly 1. If not, the code will enforce it.
- **mpole\_ZoneType\_n**: String value containing the outer radial zone type for the n-th outer zone. If set to 'exponential', the radial equation  $r(Q) = s \cdot \Delta r \cdot Q^t$ , defining the upper bound radius of the Q-th radial bin in the n-th outer zone, is used. If set to 'logarithmic', the radial equation  $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$  is used. In these equations  $Q$  is a local radial bin counting index for each outer zone and  $s, t$  are parameters defining size and growth of the outer zone radial bins (see below).
- **mpole\_ZoneScalar\_n**: The scalar value  $s$  in the n-th outer radial zone equation  $r(Q) = s \cdot \Delta r \cdot Q^t$  or  $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$ . The scalar is needed to be able to increase (or decrease) the size of the first radial bin with respect to the default smallest outer zone radius  $\Delta r$ .
- **mpole\_ZoneExponent\_n**: The exponent value  $t$  in the n-th outer radial zone equations  $r(Q) = s \cdot \Delta r \cdot Q^t$  or  $r(Q) = s \cdot \Delta r \cdot (e^{Qt} - 1)/(e^t - 1)$ . The exponent controls the growth (shrinkage) in size of each radial bin with increasing bin index. For the first equation, growing will occur for  $t > 1$ , shrinking for  $t < 1$  and same size for  $t = 1$ . For the logarithmic equation, growing will occur for  $t > 0$ , shrinking for  $t < 0$ , but the same size option  $t = 0$  will not work because the denominator becomes undefined. The same size option must hence be treated using the exponential outer zone type choice.
- **Runtime parameter types, defaults and options:**

Parameter	Type	Default	Options
<b>mpole_Lmax</b>	integer	0	> 0
<b>mpole_2DSymmetryPlane</b>	logical	false	true
<b>mpole_3DAxisymmetry</b>	logical	false	true
<b>mpole_DumpMoments</b>	logical	false	true
<b>mpole_PrintRadialInfo</b>	logical	false	true
<b>mpole_IgnoreInnerZone</b>	logical	false	true
<b>mpole_InnerZoneSize</b>	integer	16	> 0
<b>mpole_InnerZoneResolution</b>	real	0.1	less than 0.1 and > 0.0
<b>mpole_MaxRadialZones</b>	integer	1	> 1
<b>mpole_ZoneRadiusFraction_n</b>	real	1.0	less than 1.0 and > 0.0
<b>mpole_ZoneType_n</b>	string	"exponential"	"logarithmic"
<b>mpole_ZoneScalar_n</b>	real	1.0	> 0.0
<b>mpole_ZoneExponent_n</b>	real	1.0	> 0.0 (exponential)
	real	-	any $\neq 0$ (logarithmic)

### 8.10.3.3 Tree Poisson solver

The tree gravity solver can be included by **setup** or a **Config** file by requesting



physics/Gravity/GravityMain/Poisson/BHTree

The current implementation works only in 3D Cartesian coordinates, and blocks have to be logically cubic (*i.e.*, `nxb=nyb=nzb`). Physical dimensions of blocks can be arbitrary, however, some multipole acceptance criteria can provide inaccurate error estimates with non-cubic blocks. The computational domain can have arbitrary dimensions, and there can be more blocks with `lrefine=1` (*i.e.*, `nblockx`, `nblocky` and `nblockz` can have different values).

Runtime parameters `gr_bhPhysMACTW` and `gr_bhPhysMACComm` control whether MACs of physical units are used in tree walk and communication, respectively. If one of them (or both) is set `.false.`, only purely geometric MAC is used for a corresponding part of the tree solver. It is not allowed to set `gr_bhPhysMACTW = .false.` and `gr_bhPhysMACComm = .true.`.

Runtime parameter `gr_bhTreeLimAngle` allows to set the limit opening angle for the purely geometrical MAC. Another condition controlling the acceptance of the node for the calculations is that the *point-of-calculation* must lie out of the box obtained by increasing the considered node by factor `gr_bhTreeSafeBox`.

Parameter `gr_bhUseUnifiedTW` controls whether the Barnes-Hut like tree walk algorithm is used (`.true.`) or whether an alternative algorithm is used which checks the MAC only once for whole block for interactions with parent blocks (`.false.`; see 8.10.2.4 for more details). The latter one is 10 – 20% faster, however, it may lead to higher errors at block boundaries, in particular if the gravity modules calculates the potential which is subsequently differentiated to obtain gravitational acceleration. The tree walk algorithm base on the priority queue is used if `grv_bhMAC` is set to "SumSquare".

Variable	Type	Default	Description
<code>gr_bhPhysMACTW</code>	logical	<code>.false.</code>	whether physical MAC should be used in tree walk
<code>gr_bhPhysMACComm</code>	logical	<code>.false.</code>	whether physical MAC should be used in communication
<code>gr_bhTreeLimAngle</code>	real	0.5	limiting opening angle
<code>gr_bhTreeSafeBox</code>	real	1.2	relative size of restricted volume around node where the point-of-calculation is not allowed to be located
<code>gr_bhUseUnifiedTW</code>	logical	<code>.true.</code>	whether Barnes-Hut like tree walk algorithm should be used
<code>gr_bhTWMaxQueueSize</code>	integer	<code>.true.</code>	maximum length of the priority queue

#### 8.10.3.4 Multigrid

The `Grid/GridSolvers/Multigrid` module is appropriate for general source distributions. It solves Poisson's equation for 1, 2, and 3 dimensional problems with Cartesian geometries. It only supports the `PARAMESH` Grid with one block at the coarsest level. For any other mesh configuration it is advisable to use the hybrid solver, which switches to a uniform grid exact solve when the specified level of coarsening has been achieved. In most use cases for FLASH, the multigrid solver will be used to solve for Gravity (see: [Chapter 20](#)). 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.

Variable	Type	Default	Description
<code>mg_MaxResidualNorm</code>	real	$1 \times 10^{-6}$	Maximum ratio of the norm of the residual to that of the right-hand side
<code>mg_maxCorrections</code>	integer	100	Maximum number of iterations to take
<code>mg_printNorm</code>	real	<code>.true.</code>	Print the norm ratio per-iteration
<code>mpole_lmax</code>	integer	4	The number of multipole moments used in the isolated case

### 8.10.3.5 Hybrid (Multigrid with PFFT)

The hybrid solver can be used in place of the Multigrid solver for problems with

- all-periodic
- 2 periodic and 1 Neumann
- 1 periodic and 2 Neumann

boundary conditions, if the default PFFT solver variant (called `DirectSolver`) is used. To use the hybrid solver in this way, add `Grid/GridSolvers/Multigrid/PfftTopLevelSolve` to your setup line or request the solver in your Simulation Config file (see e.g. `unitTest/PFFT_PoissonFD`). The following setup lines create a unit test that uses first the hybrid solver and then the standard Multigrid solver

```
./setup unitTest/PFFT_PoissonFD -auto -3d -parfile=flash_pm_3d.par -maxblocks=800 +noio

./setup unitTest/PFFT_PoissonFD -auto -3d -parfile=flash_pm_3d.par -maxblocks=800 +noio
--without-unit=Grid/GridSolvers/Multigrid/PfftTopLevelSolve
--with-unit=Grid/GridSolvers/Multigrid
```

It is also possible to select a different PFFT solver variant. In that case, different combinations of boundary conditions for the Poisson problem may be supported. The `HomBcTrigSolver` variant supports the largest set of combinations of boundary conditions. Use the `PfftSolver` setup variable to choose a variant. Thus, appending `PfftSolver=HomBcTrigSolver` to the `setup` chooses the `HomBcTrigSolver` variant. When using the hybrid solver with the PFFT variants `HomBcTrigSolver` or `SimplePeriodicSolver`, the runtime parameter `gr_pfftDiffOpDiscretize` should be set to 1.

The Multigrid runtime parameters given in the previous section also apply.

### 8.10.4 HYPRE

As a part of implicit time advancement we end up with a system of equations that needs to be solved at every time step. In FLASH4 the HYPRE linear algebra package is used to solve these systems of equations. Therefore it is necessary to install Hype if this capability of FLASH is to be used.

`Grid_advanceDiffusion` is the API function which solves the system of equations. This API is provided by both the split and unsplit solvers. The unsplit solver uses HYPRE to solve the system of equations and split solver does a direct inverse using Thomas algorithm. Note that the split solver relies heavily on PFFT infrastructure for data exchange and a significant portion of work in split `Grid_advanceDiffusion` involves PFFT routines. In the unsplit solver the data exchange is implicitly done within HYPRE and is hidden.

The steps in unsplit `Grid_advanceDiffusion` are as follows:

- Setup HYPRE grid object
- Exchange Factor B
- Set initial guess
- Compute HYPRE Matrix M such that  $B = MX$
- Compute RHS Vector B
- Compute matrix A
- Solve system  $AX = B$
- Update solution (in FLASH4)

Mapping UG grid to HYPRE matrix is trivial, however mapping PARAMESH grid to a HYPRE matrix can be quite complicated. The process is simplified using the grid interfaces provided by HYPRE.

- Struct Grid interface
- SStruct Grid interface
- IJ System interface

The choice of an interface is tightly coupled to the underlying grid on which the problem is being solved. We have chosen the SSTRUCT interface as it is the most compatible with the block structured AMR mesh in FLASH4. Two terms commonly used in HYPRE terminology are part and box. We define these terms in equivalent FLASH4 terminology. A HYPRE box object maps directly to a leaf block in FLASH4. The block is then defined by its extents. In FLASH4 this information can be computed easily using a combination of `Grid_getBlkCornerID` and `Grid_getBlkIndexLimits`. All leaf blocks at a given refinement level form a HYPRE part. So number of parts in a typical FLASH4 grid would be give by,

```
nparts = leaf.block(lrefine.max) - leaf.block(lrefine.min) + 1
```

So, if a grid is fully refined or UG,  $nparts = 1$ . However, there could still be more than one box object.

Setting up the HYPRE grid object is one of the most important step of the solution process. We use the SSTRUCT interface provided in HYPRE to setup the grid object. Since the HYPRE Grid object is mapped directly with FLASH4 grid, whenever the FLASH4 grid changes the HYPRE grid object needs to be updated. Consequently with AMR the HYPRE grid setup might happen multiple times.

Setting up a HYPRE grid object is a two step process,

- Creating stenciled relationships.
- Creating Graph relationships.

Stenciled relationships typically exist between leaf blocks at same refinement level (intra part) and graph relationships exist between leaf blocks at different refinement levels (inter part). The fine-coarse boundary is handled in such a way that fluxes are conserved at the interface (see [Chapter 19](#) for details). UG does not require any graph relationships.

Whether a block needs a graph relationship depends on the refinement level of its neighbor. While this information is not directly available in PARAMESH, it is possible to determine whether the block neighbor is coarser or finer. Combining this information with the constraint of at best a factor of two jump in refinement at block boundaries, it is possible to compute the part number of a neighbor block, which in turn determines whether we need a graph. Creating a graph involves creating a link between all the cells on the

block boundary.

Once the grid object is created, the matrix and vector objects are built on the grid object. The diffusion solve needs uninterpolated data from neighbor blocks even when there is a fine-coarse boundary, therefore it cannot rely upon the guardcell fill process. A two step process is used to handle this situation,

- Since HYPRE has access to  $X$ (at  $n$ , *i.e.*, initial guess), the RHS vector  $B$  can be computed as  $MX$  where  $M$  is a modified Matrix.
- Similarly the value of Factor  $B$  can be shared across the fine-coarse boundary by using `Grid_conserveFluxes`, the fluxes need to be set in a intuitive way to achieve the desired effect.

With the computation of Vector  $B$  (RHS), the system can be solved using HYPRE and UNK can be updated.

#### 8.10.4.1 HYPRE Solvers

In FLASH4 we use the HYPRE PARCSR storage format as this exposes the maximum number of iterative solvers.

Table 8.5: Solvers, Preconditioners combinations used with `Grid/GridSolvers/HYPRE`.

Solver	Preconditioner
PCG	AMG, ILU(0)
BICGSTAB	AMG, ILU(0)
GMRES	AMG, ILU(0)
AMG	-
SPLIT	-

**Parallel runs:** One issue that has been observed is that there is a difference in the results produced by HYPRE using one or more processors. This would most likely be due to use of CG (krylov subspace methods), which involves an `MPI_SUM` over the dot product of the residue. We have found this error to be dependent on the type of problem. One way to get across this problem is to use direct solvers in HYPRE like `SPLIT`. However we have noticed that direct solvers tend to be slow. The released code has an option to use the `SPLIT` solver, but this solver has not been extensively tested and was used only for internal debugging purposes and the usage of the HYPRE `SPLIT` solver in FLASH4 is purely experimental.

**Customizing solvers:** HYPRE exposes a lot more parameters to tweak the solvers and preconditioners mentioned above. We have only used those which are applicable to general diffusion problems. Although in general these settings might be good enough it is by no means complete and might not be applicable to all class of problems. Use of additional HYPRE parameters might require direct manipulation of FLASH4 code.

**Symmetric Positive Definite (SPD) Matrix:** PCG has been noticed to have convergence issues which might be related to (not necessarily),

- A non-SPD matrix generated due to round of errors (only).
- Use of BoomerAMG as PC (refer to HYPRE manual).

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

Table 8.6: Runtime parameters used with Grid/GridSolvers/HYPRE.

Variable	Type	Default	Description
<code>gr_hyprePCType</code>	string	"hypre_amg"	Algebraic Multigrid as Preconditioner
<code>gr_hypreMaxIter</code>	integer	10000	Maximum number of iterations
<code>gr_hypreRelTol</code>	real	$1 \times 10^{-8}$	Maximum ratio of the norm of the residual to that of the initial residue
<code>gr_hypreSolverType</code>	string	"hypre_pcg"	Type of linear solver, Preconditioned Conjugate gradient
<code>gr_hyprePrintSolveInfo</code>	boolean	FALSE	enables/disables some basic solver information (for e.g number of iterations)
<code>gr_hypreInfoLevel</code>	integer	1	Verbosity level of solver diagnostics (refer HYPRE manual).
<code>gr_hypreFloor</code>	real	$1 \times 10^{-12}$	Used to floor the end solution.
<code>gr_hypreUseFloor</code>	boolean	TRUE	whether to apply <code>gr_hypreFloor</code> to floor results from HYPRE.

#### 8.10.4.2 Unified Hypre Solver

Often the systems of equations that we want to solve with HYPRE are some variation of parabolic PDES that can be expressed generically as

$$A_i \frac{\partial U_i}{\partial t} = C_{ij} U_j^* + \partial_k (B_{ijkl} \partial_l U_j^* + E_{ijk} U_j^*) + D_i + F_{ijk} \partial_k U_j^* \quad (8.93)$$

Typically these derivatives need to be discretized in the same manner independent of the system of equations with the only differences being in the factors and transport coefficients used in the tensors  $A_i$ ,  $B_{ijkl}$ ,  $C_{ij}$ ,  $D_i$ ,  $E_{ijk}$  &  $F_{ijk}$ . FLASH4.8 introduces a unified HYPRE solver that will construct all the necessary HYPRE objects, solve the system equations given the tensors in Eq. 8.93, and update the `Grid` variables. This is supported for uniform and paramesh grids.

Previous versions of the HYPRE solvers, as described above, shared a single instance of all the HYPRE objects. The unified solver described in this section maintains separate objects for each solver that are maintained in `source/Grid/GridSolvers/HYPRE/Unified/gr_uhypreData`. New solvers are registered at `setup` in `source/Grid/GridSolvers/HYPRE/Unified/Config`. This will also generate runtime parameters unique to each solver as described in Table 8.6 replacing `gr_hypre`  $\rightarrow$  `gr_hypreSolver_`.

Table 8.7: Setup generated runtime parameters for each unified HYPRE solver created. For a given solver, `Solver`, the actual runtime parameters will be prepended with `gr_uhypreSolver_`.

Parameter	Type	Default	Description
<code>SolverType</code>	STRING	"HYPRE'PCG"	Runtime parameters used with Grid/GridSolvers/HYPRE.
<code>PCType</code>	STRING	"HYPRE'AMG"	Algebraic Multigrid as Preconditioner
<code>RelTol</code>	REAL	1.0e-8	Maximum ratio of the norm of the residual to that of the initial residue

AbsTol	REAL	0.0	
SolverAutoAbsTolFact	REAL	0.0	
MinIter	INTEGER	0	Minimum number of iterations for HYPRE solve phase
MaxIter	INTEGER	500	Maximum number of iterations for HYPRE solve phase
PrintSolveInfo	BOOLEAN	FALSE	When <code>.true.</code> prints solve info from HYPRE
InfoLevel	INTEGER	1	level of output when using <code>PrintSolveInfo</code>
FloorType	INTEGER	0	
Floor	REAL	1.0e-12	Used to floor the end solution.
Use2Norm	BOOLEAN	FALSE	
CfTol	REAL	0.0	
RecomputeResidual	BOOLEAN	FALSE	
RecomputeResidualP	INTEGER	-1	
RelChange	BOOLEAN	FALSE	
SlopeLimType	STRING	"HYPRESL'MC"	

Systems are solved with calls to `Grid_advanceImplicit` by passing an instance of `parabolicProblemParms` from the `Grid_parabolicProblem` module, whose parameters are detailed in Table 8.8. This type contains all the information about the system of equations described by Eq. 8.93. The various tensors are described by multidimensional arrays, such as `FactorE` whose elements will also be multiplied by a variable in `unk` through `iFactorE`, which should match the index in `iFactorBMap`.

Table 8.8: `parabolicProblemParms` type from `Grid_parabolicProblem`

Variable	Type	Description
<code>iFactorBMap</code>	<code>integer(ncoeffs)</code>	Unique indices in <code>solnvec</code> for transport coefficients
<code>iFactorA</code>	<code>integer(nvars)</code>	Stores indices into <code>solnvec</code> for LHS term (weighting factors)
<code>FactorA</code>	<code>real(nvars)</code>	Constant coefficient for LHS term (weighting factor)
<code>iFactorB</code>	<code>integer(nvars,nvars,NDIM,NDIM)</code>	Indices in <code>iFactorBMap</code> for transport coefficients
<code>FactorB</code>	<code>real(nvars,nvars,NDIM,NDIM)</code>	Constant coefficients for transport/gradient terms
<code>iFactorC</code>	<code>integer(nvars,nvars)</code>	Indices in <code>solnvec</code> with inter variable coupling coefficients
<code>FactorC</code>	<code>real(nvars,nvars)</code>	Constant coefficients for coupling coefficients
<code>iFactorD</code>	<code>integer(nvars)</code>	Indices in <code>solnvec</code> to map to
<code>FactorD</code>	<code>real(nvars)</code>	Constant coefficient for each variable to multiply constant source
<code>iFactorE</code>	<code>integer(nvars,nvars,NDIM)</code>	Indices in <code>solnvec</code> to map to
<code>FactorE</code>	<code>real(nvars,nvars,NDIM)</code>	Constant coefficients for transport/gradient terms involving geometric source terms
<code>iFactorF</code>	<code>integer(nvars,nvars,NDIM)</code>	Indices in <code>solnvec</code> to map to
<code>FactorF</code>	<code>real(nvars,nvars,NDIM)</code>	Constant coefficients for transport/gradient terms involving geometric source terms
<code>iFactorG</code>	<code>integer(nvars,nvars)</code>	Indices in <code>solnvec</code> with inter variable coupling coefficients
<code>FactorG</code>	<code>real(nvars,nvars)</code>	Constant coefficients for coupling coefficients
<code>iVar</code>	<code>integer(nvars)</code>	Stores indices into <code>solnvec</code> for variables to solve
<code>iFluxMap</code>	<code>integer(nvars)</code>	Stores unique indices into <code>solnvec</code> for flux limiting variable

iFlux	integer( <b>nvars</b> )	Stores indices in iFluxMap for each variable's maximum flux
Flux	real( <b>nvars</b> )	Stores indices in iFluxMap for each variable's maximum flux
mode	integer( <b>nvars</b> )	Flux limiter mode to use for each variable
bcTypes	integer( <b>NDIM</b> , <b>nvars</b> ,2)	Boundary condition flag
bcValues	real( <b>NDIM</b> , <b>nvars</b> ,2)	Value or slope at boundary - when using Neumann or Dirichlet BC's
theta	real	Implicitness of solve
iSolution	integer	Handle so we can keep track of what equation we are solving. Should match a system PPDEFINE from GridSolvers/HYPRE/Unified/Config

---

Eq. 8.93 is evolved with a finite-volume discretization. We integrate Eq. 8.93 over a control volume  $V_\alpha$ , centered on the point  $\mathbf{x}_\alpha$ , to get

$$V_\alpha A_i \left. \frac{\partial U_i}{\partial t} \right|_{\mathbf{x}_\alpha} = V_\alpha C_{ij} U_j^* + \oint_{\partial V_\alpha} (B_{ijkl} \partial_l U_j^* + E_{ijk} U_j^*) \cdot d\mathbf{A} + V_\alpha D_i + [V_\alpha F_{ijk} \partial_k U_j^*]_{\mathbf{x}_\alpha}, \quad (8.94)$$

where the divergence theorem has been used to turn the  $B$ -factor term to be an integral over the face of our control volume and any remaining volume integrals are approximated by the value at the center of control volume  $\mathbf{x}_\alpha$ . The  $B$ -factors require derivatives integrated over the faces of mesh and the  $F$ -factors need derivatives evaluated at the cell-centers.

There are two classes of derivatives that could be need at a face, either derivatives normal to the face or those that are transverse to the face. In both cases finite-difference weights on a stencil of neighboring points are used, but differ in the approximation used for the surface integration. In the case of normal derivatives the stencil is constructed from cells that share the face of interest, and the derivative is evaluated at the center of the face, allowing a mid-point quadrature for the surface integral on that face.

$$\int_{A_{i+1/2}} B \partial_l U dA_l = B_{i+1/2} \partial_l U(\mathbf{x}_{i+1/2}) A_{i+1/2} + \mathcal{O}(\Delta^2) \quad (8.95)$$

A stencil that could approximate derivatives transverse to a cell-face would require neighboring cells that share a common edge, introducing a new coupling between diagonal elements, for example  $U_{ij} \leftarrow U_{i+1,j+1}$ . If the face-integrals are approximated with a mid-point quadrature, like in Eq. 8.97, the diagonal coupling, through the face coefficients  $B_{i+1/2,j}$  &  $B_{i+1,j+1/2}$  is no longer symmetric if the transport coefficients,  $B$ , vary in space

$$U_{ij} \xleftarrow{B_{i+1/2,j}} U_{i+1,j+1} \quad U_{i+1,j+1} \xleftarrow{B_{i+1,j+1/2}} U_{ij}. \quad (8.96)$$

The resulting systems of equations are numerically unstable, particularly when  $B_{i+1,j+1/2} \gg B_{i+1/2,j}$ . In order to symmetrize the discretization for these new edge-couplings we approximate the face-integrals with a two point trapezoid rule in the transverse direction

$$\int_{A_{i+1/2}} B \partial_l U dA_l \approx \frac{1}{2} (B_{i+1/2,j+1/2} \partial_l U_{i+1/2,j+1/2} + B_{i+1/2,j-1/2} \partial_l U_{i+1/2,j-1/2}) A_{i+1/2} \quad (8.97)$$

This way the edge-coupling is always symmetric through the edge-centered transport coefficients  $B_{i+1/2,j+1/2}$ . ■

For both edge-centered and face-centered derivatives the **Unified** HYPRE solver computes stencil weights for all possible refinement configurations at fine-coarse boundaries and handles the creation of graph objects between parts.



### 8.10.4.3 Curvilinear Coordinates

For scalar systems of equations the finite-volume formulism in Eq. 8.94 naturally incorporates any grid geometry through the volume and area factors. However, for vector systems of equations, such as velocity with viscosity or magnetic field with resistive MHD, the discretization needs to account for geometric scale factors introduced through the vector calculus operators. The **Unified HYPRE** solver currently only handles "cylindrical" geometries, where it is often convenient to express the generic system of parabolic equations as

$$A_i \frac{\partial U_i}{\partial t} = C_{ij} U_j^* + D_i + \frac{1}{r} \partial_k (r B_{ijkl} \partial_l U_j^* + E_{ijk} U_j^*) + \frac{1}{r} F_{ijk} \partial_k U_j^* + \frac{1}{r^2} G_{ij} U_j^* \quad (8.98)$$

Just as before the second derivative  $B$  &  $E$ -factor terms can be integrated with the divergence theorem to be evaluated on the cell-faces, but the volume  $F$  &  $G$ -factor terms will include a  $\frac{1}{r}$  geometric term.

## 8.11 Grid Geometry

FLASH can use various kinds of coordinates ("geometries") for modeling physical problems. The available geometries represent different (orthogonal) curvilinear coordinate systems.

The geometry for a particular problem is set at runtime (after an appropriate invocation of **setup**) through the **geometry** runtime parameter, which can take a value of "**cartesian**", "**spherical**", "**cylindrical**", or "**polar**". Together with the dimensionality of the problem, this serves to completely define the nature of the problem's coordinate axes (Table 8.9). Note that not all **Grid** implementations support all geometry/dimension combinations. Physics units may also be limited in the geometries supported, some may only work for Cartesian coordinates.

The core code of a **Grid** implementation is not concerned with the mapping of cell indices to physical coordinates; this is not required for under-the-hood **Grid** operations such as keeping track of which blocks are neighbors of which other blocks, which cells need to be filled with data from other blocks, and so on. Thus the physical domain can be logically modeled as a rectangular mesh of cells, even in curvilinear coordinates.

There are, however, some areas where geometry needs to be taken into consideration. The correct implementation of a given geometry requires that gradients and divergences have the appropriate area factors and that the volume of a cell is computed properly for that geometry. Initialization of the grid as well as AMR operations (such as restriction, prolongation, and flux-averaging) must respect the geometry also. Furthermore, the hydrodynamic methods in FLASH are finite-volume methods, so the interpolation must also be conservative in the given geometry. The default mesh refinement criteria of FLASH4 also currently take geometry into account, see Section 8.6.3 above.

**A convention:** in this section, Small letters  $x$ ,  $y$ , and  $z$  are used with their usual meaning in designating coordinate directions for specific coordinate systems: *i.e.*,  $x$  and  $y$  refer to directions in Cartesian coordinates, and  $z$  may refer to a (linear) direction in either Cartesian or cylindrical coordinates.

On the other hand, capital symbols  $X$ ,  $Y$ , and  $Z$  are used to refer to the (up to) three directions of any coordinate system, *i.e.*, the directions corresponding to the **I**AXIS, **J**AXIS, and **K**AXIS dimensions in FLASH4, respectively. Only in the Cartesian case do these correspond directly to their small-letter counterparts. For other geometries, the correspondence is given in Table 8.9.

### 8.11.1 Understanding 1D, 2D, and 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? And what does it mean to do a 1D or a 2D simulation of three-dimensional reality? Physical reality has three spatial dimensions (as far as the physical problems simulated with FLASH are concerned). In Cartesian coordinates, it is relatively straightforward to understand what a 2-d (or 1-d) simulation means: "Just leave out one (or two) coordinates." This is less obvious for other coordinate systems, therefore some fundamental discussion follows.

A reduced dimensionality (RD) simulation can be naively understood as taking a cut (or, for 1-d, a linear probe) through the real 3-d problem. However, there is also an assumption, not always explicitly



Table 8.9: 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.

name	dimensions	support	axisymmetric	X-coord	Y-coord	Z-coord
cartesian	1	pm4,pm2,UG	n	$x$		
cartesian	2	pm4,pm2,UG	n	$x$	$y$	
cartesian	3	pm4,pm2,UG	n	$x$	$y$	$z$
cylindrical	1	pm4,UG	y	$r$		
cylindrical	2	pm4,pm2,UG	y	$r$	$z$	
cylindrical	3	pm4,UG	n	$r$	$z$	$\phi$
spherical	1	pm4,pm2,UG	y	$r$		
spherical	2	pm4,pm2,UG	y	$r$	$\theta$	
spherical	3	pm4,pm2,UG	n	$r$	$\theta$	$\phi$
polar	1	pm4,UG	y	$r$		
polar "polar + $z$ "	2	pm4,pm2,UG	n	$r$	$\phi$	
(cylindrical with a different ordering of coordinates)	3	—	n	$r$	$\phi$	$z$

stated, that is implied in this kind of simulation: namely, that the cut (or line) is representative of the 3-d problem. This can be given a stricter meaning: it is assumed that the physics of the problem do not depend on the omitted dimension (or dimensions). A RD simulation can be a good description of a physical system only to the degree that this assumption is warranted. Depending on the nature of the simulated physical system, non-dependence on the omitted dimensions may mean the absence of force and/or momenta vector components in directions of the omitted coordinate axes, zero net mass and energy flow out of the plane spanned by the included coordinates, or similar.

For omitted dimensions that are lengths —  $z$  and possibly  $y$  in Cartesian, and  $z$  in cylindrical and polar RD simulations — one may think of a 2-d cut as representing a (possibly very thin) layer in 3-d space sandwiched between two parallel planes. There is no *a priori* definition of the thickness of the layer, so it is not determined what 3-d volume should be assigned to a 2-d cell in such coordinates. We can thus arbitrarily assign the length “1” to the edge length of a 3-d cell volume, making the volume equal to the 2-d area. We can understand generalizations of “volume” to 1-d, and of “face areas” to 2-d and 1-d RD simulations with omitted linear coordinates, in an equivalent way: just set the length of cell edges along omitted dimensions to 1.

For omitted dimensions that are angles — the  $\theta$  and  $\phi$  coordinates on spherical, cylindrical, and polar geometries — it is easier to think of omitting an angle as the equivalent of integrating over the full range of that angle coordinate (under the assumption that all physical solution variables are independent of that angle). Thus omitting an angle  $\phi$  in these geometries implies the assumption of axial symmetry, and this is noted in [Table 8.9](#). Similarly, omitting both  $\phi$  and  $\theta$  in spherical coordinates implies an assumption of complete spherical symmetry. When  $\phi$  is omitted, a 2-d cell actually represents the 3-d object that is generated by rotating the 2-d area around a  $z$ -axis. Similarly, when only  $r$  is included, 1-d cells (*i.e.*,  $r$  intervals) represent hollow spheres or cylinders. (If the coordinate interval begins at  $r_l = 0.0$ , the sphere or cylinder is massive instead of hollow.)

As a result of these considerations, the measures for cell (and block) volumes and face areas in a simulation depends on the chosen geometry. Formulas for the volume of a cell dependent on the geometry are given in the geometry-specific sections further below.

As discussed in [Figure 8.6](#), to ensure conservation at a jump in refinement in AMR grids, a flux correction step is taken. The fluxes leaving the fine cells adjacent to a coarse cell are used to determine more accurately the flux entering the coarse cell. This step takes the coordinate geometry into account in order to accurately determine the areas of the cell faces where fine and coarse cells touch. By way of example, an illustration is

provided below in the section on cylindrical geometry.

### 8.11.1.1 Extensive Quantities in Reduced Dimensionality

The considerations above lead to some consequences for the understanding of extensive quantities, like mass or energies, that may not be obvious.

The following discussion applies to geometries with omitted dimensions that are lengths —  $z$  and possibly  $y$  in Cartesian, and  $z$  in cylindrical and polar RD simulations. We will consider Cartesian geometries as the most common case, and just note that the remaining cases can be thought of analogously.

In 2D Cartesian, the “volume” of a cell should be  $\Delta V = \Delta x \Delta y$ . We would like to preserve the form of equations that relate extensive quantities to their densities, *e.g.*,  $\Delta m = \rho \Delta V$  for mass and  $\Delta E_{tot} = \rho e_{tot} \Delta V$  for total energy in a cell. We also like to retain the usual definitions for intensive quantities such as density  $\rho$ , including their physical values and units, so that material density  $\rho$  is expressed in  $g/cm^3$  (more generally  $M/L^3$ ), no matter whether 1D, 2D, or 3D. We cannot satisfy both desiderata without modifying the interpretation of “mass”, “energy”, and similar extensive quantities in the system of equations modeled by FLASH.

Specifically, in a 2D Cartesian simulation, we have to interpret “mass” as really representing a linear mass density, measured in  $M/L$ . Similarly, an “energy” is really a linear energy density, *etc.*

In a 1D Cartesian simulation, we have to interpret “mass” as really representing a surface mass density, measured in  $M/L^2$ , and an “energy” is really a surface energy density.

(There is a different point of view, which amounts to the same thing: One can think of the “mass” of a cell (in 2D) as the physical mass contained in a three-dimensional cell of volume  $\Delta x \Delta y \Delta z$  where the cell height  $\Delta z$  is set to be exactly 1 length unit. Always with the understanding that “nothing happens” in the  $z$  direction.)

Note that this interpretation of “mass”, “energy”, *etc.* must be taken into account not just when examining the physics in individual cells, but equally applies for quantities integrated over larger regions, including the “total mass” or “total energy” *etc.* reported by FLASH in `flash.dat` files — they are to be interpreted as (linear or surface) densities of the nominal quantities (or, alternatively, as integrals over 1 length unit in the missing Cartesian directions).

### 8.11.2 Choosing a Geometry

The user chooses a geometry by setting the `geometry` runtime parameter in `flash.par`. The default is “cartesian” (unless overridden in a simulation’s `Config` file). Depending on the `Grid` implementation used and the way it is configured, the geometry may also have to be compiled into the program executable and thus may have to be specified at configuration time; the `setup` flag `-geometry` should be used for this purpose, see [Section 5.2](#).

The `geometry` runtime parameter is most useful in cases where the geometry does not have to be specified at compile-time, in particular for the Uniform Grid. The runtime parameter will, however, always be considered at run-time during `Grid` initialization. If the `geometry` runtime parameter is inconsistent with a geometry specified at setup time, FLASH will then either override the geometry specified at setup time (with a warning) if that is possible, or it will abort.

This runtime parameter is used by the `Grid` unit and also by hydrodynamics solvers, which add the necessary geometrical factors to the divergence terms. Next we describe how user code can use the runtime parameter’s value.

### 8.11.3 Getting Geometry Information in Program Code

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

Coordinate information for the mesh can be determined via the `Grid_getCellCoords` routine. This routine can provide the coordinates of cells at the left edge, right edge, or center. The width of cells

```

#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.

#### 8.11.4 Available Geometries

Currently, all of FLASH’s physics support one-, two-, and (with a few exceptions explicitly stated in the appropriate chapters) three-dimensional Cartesian grids. Some units, including the FLASH Grid unit and PPM hydrodynamics unit ([Chapter 15](#)), support additional geometries, such as two-dimensional cylindrical  $(r, z)$  grids, one/two-dimensional spherical  $(r)/ (r, \theta)$  grids, and two-dimensional polar  $(r, \phi)$  grids. Some specific considerations for each geometry follow.

The following tables use the convention that  $r_l$  and  $r_r$  stand for the values of the  $r$  coordinate at the “left” and “right” end of the cell’s  $r$ -coordinate range, respectively (*i.e.*,  $r_l < r_r$  is always true), and  $\Delta r = r_r - r_l$ ; and similar for the other coordinates.

#### 8.11.4.1 Cartesian geometry

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

```
geometry = "cartesian"
```

in the runtime parameter file.

*Cell Volume in Cartesian Coordinates*

1-d	$\Delta x$
2-d	$\Delta x \Delta y$
3-d	$\Delta x \Delta y \Delta z$

#### 8.11.4.2 Cylindrical geometry

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

```
geometry = "cylindrical"
```

*Cell Volume in Cylindrical Coordinates*

1-d	$\pi(r_r^2 - r_l^2)$
2-d	$\pi(r_r^2 - r_l^2) \Delta z$
3-d	$\frac{1}{2}(r_r^2 - r_l^2) \Delta z \Delta \phi$

As in other non-Cartesian geometries, if the minimum radius is chosen to be zero (`xmin = 0.`), the left-hand boundary type should be reflecting (or `axisymmetric`). 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

$$A = \pi(r_r^2 - r_l^2) . \quad (8.99)$$

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

$$f_3 = \frac{A_1 f_1 + A_2 f_2}{A_3} . \quad (8.100)$$

For fluxes in the radial direction, the cross-sectional area is independent of the height,  $z$ , so the corrected flux is simply taken as the average of the flux densities in the adjacent finer cells.

#### 8.11.4.3 Spherical geometry

One or two dimensional spherical problems can be performed by specifying

```
geometry = "spherical"
```

in the runtime parameter file.

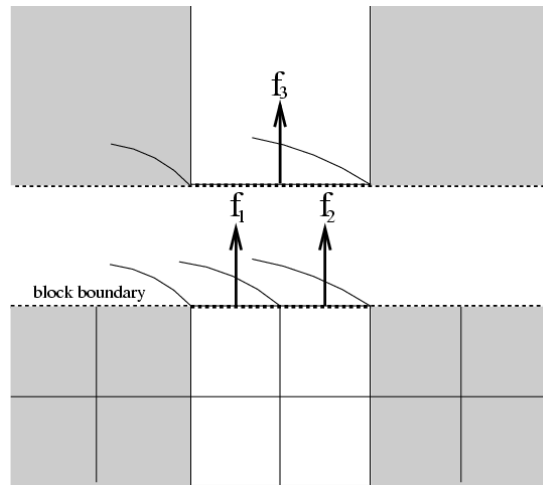


Figure 8.15: Diagram showing two fine cells and a coarse cell at a jump in refinement in the cylindrical ‘ $z$ ’ direction. The block boundary has been cut apart here for illustrative purposes. The fluxes out of the fine blocks are shown as  $f_1$  and  $f_2$ . These will be used to compute a more accurate 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*

1-d	$\frac{4}{3}\pi(r_r^3 - r_l^3)$
2-d	$\frac{2}{3}\pi(r_r^3 - r_l^3)(\cos(\theta_l) - \cos(\theta_r))$
3-d	$\frac{1}{3}(r_r^3 - r_l^3)(\cos(\theta_l) - \cos(\theta_r))\Delta\phi$

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

#### 8.11.4.4 Polar geometry

Polar geometry is a 2-D subset of 3-D cylindrical configuration without the “ $z$ ” coordinate. Such geometry is natural for studying objects like accretion disks. This geometry can be selected by specifying

```
geometry = "polar"
```

in the runtime parameter file.

*Cell Volume in Polar Coordinates*

1-d	$\pi(r_r^2 - r_l^2)$
2-d	$\frac{1}{2}(r_r^2 - r_l^2)\Delta\phi$
3-d	$\frac{1}{2}(r_r^2 - r_l^2)\Delta\phi\Delta z$ (not supported)

As in other non-Cartesian geometries, if the minimum radius is chosen to be zero (`xmin = 0.`), the left-hand boundary type should be reflecting.

### 8.11.5 Conservative Prolongation/Restriction on Non-Cartesian Grids

When blocks are refined, we need to initialize the child data using the information in the parent cell in a manner which preserves the cell-averages in the coordinate system we are using. When a block is derefined, the parent block (which is now going to be a leaf block) needs to be filled using the data in the child blocks (which are soon to be destroyed). The first procedure is called prolongation. The latter is called restriction. Both of these procedures must respect the geometry in order to remain conservative. Prolongation and restriction are also used when filling guard cells at jumps in refinement.

#### 8.11.5.1 Prolongation

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

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

These paths will be automatically added by the `setup` script when the `-gridinterpolation=monotonic` option is in effect (which is the case by default, unless `-gridinterpolation=native` was specified). The “monotonic” interpolation scheme used in both cases is taking geometry into consideration and is appropriate for all supported geometries.

#### FLASH Transition

Some more specific PARAMESH 2 interpolation schemes are included in the distribution and might be useful for compatibility with FLASH2:

- `source/Grid/GridMain/paramesh/Paramesh2/quadratic_cartesian` (for Cartesian coordinates)
- `source/Grid/GridMain/paramesh/Paramesh2/quadratic_spherical` (for spherical coordinates)

Other geometry types and prolongation schemes can be added in a manner analogous to the ones implemented here.

These routines could be included by specifying the correct path in your `Units` file, or by using appropriate `-unit=` flags for `setup`. However, their use is not recommended.

#### 8.11.5.2 Restriction

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

$$\langle f \rangle_{i,j} = \frac{V_{ic,jc} \langle f \rangle_{ic,jc} + V_{ic+1,jc} \langle f \rangle_{ic+1,jc} + V_{ic,jc+1} \langle f \rangle_{ic,jc+1} + V_{ic+1,jc+1} \langle f \rangle_{ic+1,jc+1}}{V_{i,j}}, \quad (8.101)$$

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

## 8.12 Unit Test

The `Grid` unit test has implementations to test Uniform Grid and PARAMESH. The Uniform Grid version of the test has two parts; the latter portion is also tested in PARAMESH. The test initializes the grid with a sinusoid function  $\sin(x) \times \cos(y) \times \cos(z)$ , distributed over a number of processors. Knowing the configuration

of processors, it is possible to determine the part of the sinusoid on each processor. Since guardcells are filled either from the interior points of the neighboring processor, or from boundary conditions, it is also possible to predict the values expected in guard cells on each processor. The first part of the UG unit test makes sure that the actual received values of guard cell match with the predicted ones. This process is carried out for both cell-centered and face-centered variables.

The second part of the UG test, and the only part of the **PARAMESH** test, exercises the get and put data functions. Since the **Grid** unit has direct access to all of its own data structures, it can compare the values fetched using the `getData` functions against the directly accessible values and report an error if they do not match. The testing of the **Grid** unit is not exhaustive, and given the complex nature of the unit, it is difficult to devise tests that would do so. However, the more frequently used functions are exercised in this test.





## Chapter 9

# IO Unit

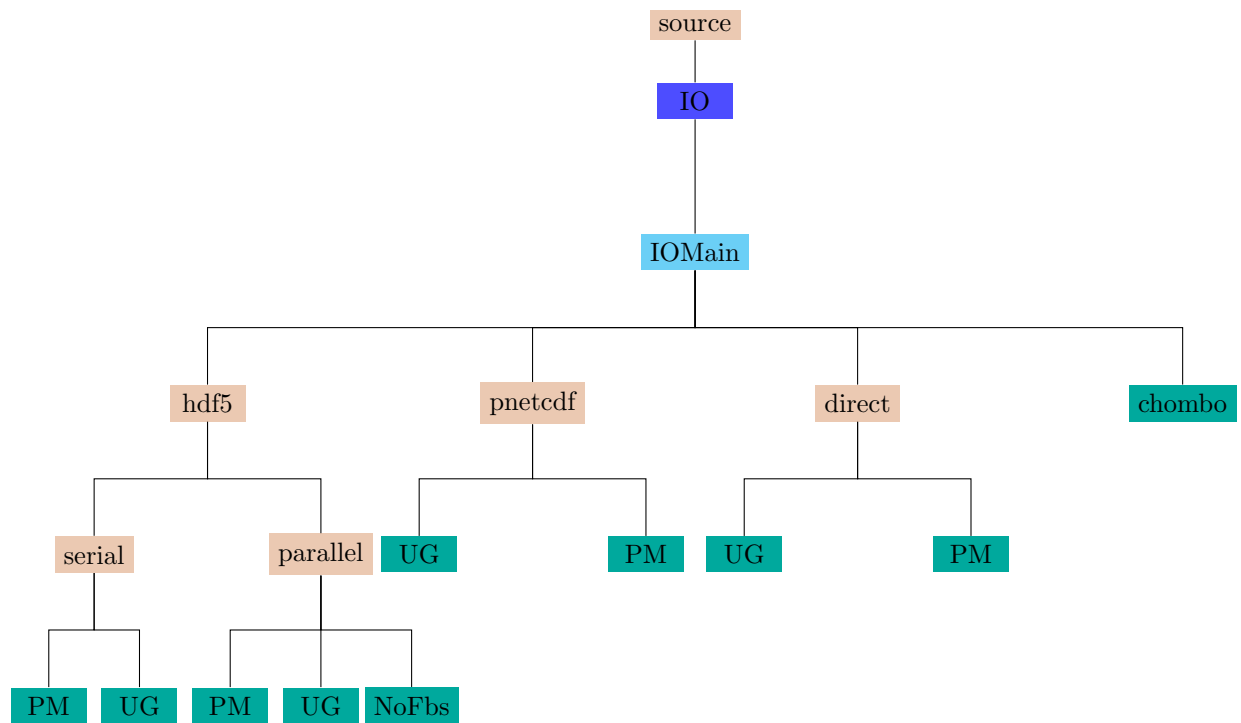


Figure 9.1: The IO unit: IOMain subunit directory tree.

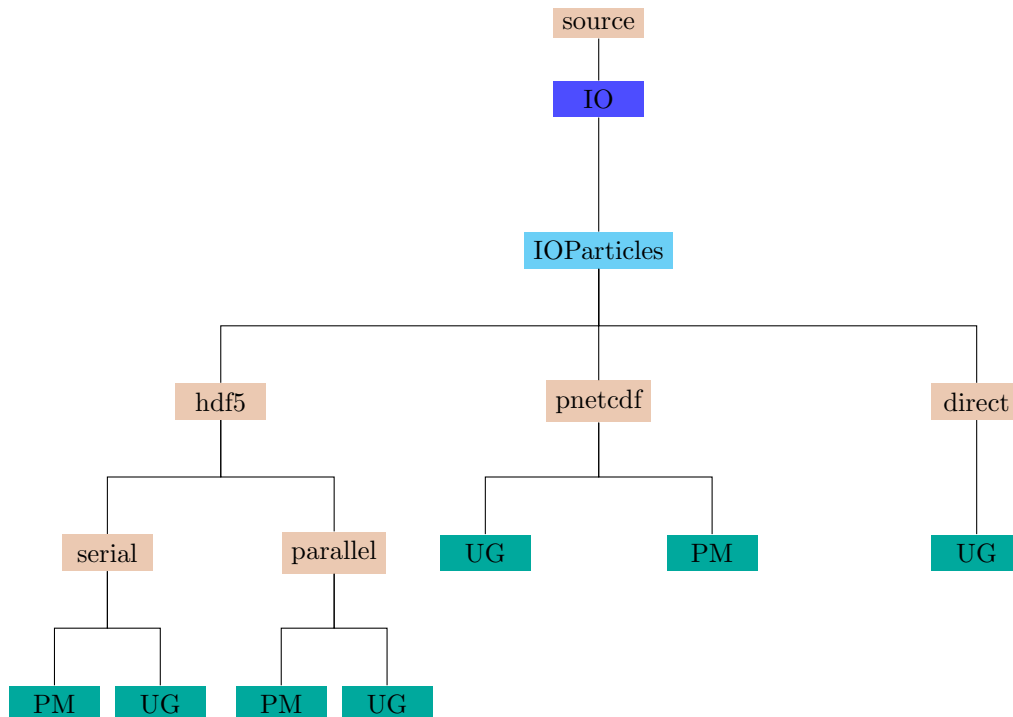


Figure 9.2: The IO unit: IOParticles subunit tree.

FLASH uses parallel input/output (IO) libraries to simplify and manage the output of the large amounts of data usually produced. In addition to keeping the data output in a standard format, the parallel IO libraries also ensure that files will be portable across various platforms. The mapping of FLASH data-structures to records in these files is controlled by the FLASH IO unit. FLASH can output data with two parallel IO libraries, HDF5 and Parallel-NetCDF. The data layout is different for each of these libraries. Since FLASH3 we also offer direct serial FORTRAN IO, which can be used as a last resort if no parallel library is available. However, FLASH post-processing tools such as *fidlr* ([Chapter 39](#)) and *sfocu* ([Chapter 37](#)) do not support the direct IO format.

**Note:**

This release supports both HDF5 and Parallel-NetCDF, including particle IO for both implementations.

Various techniques can be used to write the data to disk when running a parallel simulation. The first is to move all the data to a single processor for output; this technique is known as serial IO. Secondly, each processor can write to a separate file, known as direct IO. As a third option, each processor can use parallel access to write to a single file in a technique known as parallel IO. Finally, a hybrid method can be used where clusters of processors write to the same file, though different clusters of processors output to different files. In general, parallel access to a single file will provide the best parallel IO performance unless the number of processors is very large. On some platforms, such as Linux clusters, there may not be a parallel file system, so moving all the data to a single processor is the only solution. Therefore FLASH supports HDF5 libraries in both serial and parallel forms, where the serial version collects data to one processor before writing it, while the parallel version has every processor writing its data to the same file.

## 9.1 IO Implementations

FLASH4 supports multiple IO implementations: direct, serial and parallel implementations as well as support for different parallel libraries. In addition, FLASH4 also supports multiple ([Chapter 8](#)) **Grid** implementations. As a consequence, there are many permutations of the IO API implementation, and the selected implementation must match not only the correct IO library, but also the correct grid. Although there are many IO options, the `setup` script in FLASH4 is quite ‘smart’ and will not let the user setup a problem with incompatible **IO** and **Grid** unit implementations. [Table 9.1](#) summarizes the different implementation of the FLASH IO unit in the current release.

Table 9.1: IO implementations available in FLASH. All implementations begin at the `/source` directory.

Implementation Path	Description
<code>IO/IOMain/HDF5/parallel/PM</code>	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 <b>PARAMESH</b> grid package.
<code>IO/IOMain/hdf5/parallel/UG</code>	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.
<code>IO/IOMain/hdf5/parallel/NoFbs</code>	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.
<code>IO/IOMain/hdf5/serial/PM</code>	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 <b>PARAMESH</b> grid package.
<code>IO/IOMain/hdf5/serial/UG</code>	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.
<code>IO/IOMain/pnetcdf/PM</code>	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 <b>PARAMESH</b> grid package.

Table 9.1: FLASH IO implementations (continued).

Implementation path	Description
IO/IOMain/pnetcdf/UG	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.
IO/IOMain/direct/UG	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.
IO/IOMain/direct/PM	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.
IO/IOMain/chombo	Hierarchical Data Format (HDF) 5 output. The subroutines provide an interface to the Chombo I/O routines that are part of the standard Chombo release. This implementation can only be used in applications built with Chombo Grid. The Chombo file layout is incompatible with tools, such as <code>fidlr</code> and <code>sfocu</code> , that depend on FLASH file layout.

FLASH4 also comes with some predefined setup **shortcuts** which make choosing the correct IO significantly easier; see [Chapter 5](#) for more details about shortcuts. In FLASH4 HDF5 serial IO is included by default. Since PARAMESH 4.0 is the default grid, the included IO implementations will be compatible with PARAMESH 4.0. For clarity, a number of 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 `-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-fixedblocksize 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 FLASH 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 26](#) for a description of that format.

There are a number of runtime parameters that are used to control the output and frequency of IO files. A list of all the runtime parameters and their descriptions for the IO unit can be found online [all of them](#). Additional description is located in [Table 9.2](#) for checkpoint parameters, [Table 9.3](#) for plotfile parameters, [Table 9.4](#) for particle file parameters, [Table 9.5](#) for `flash.dat` parameters, and [Table 9.6](#) for general IO parameters.

### 9.2.1 Checkpoint files - Restarting a Simulation

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

The API routine for writing a checkpoint file is `IO_writeCheckpoint`. Users usually will not need to call this routine directly because the FLASH IO unit calls `IO_writeCheckpoint` from the routine `IO_output` which checks the runtime parameters to see if it is appropriate to write a checkpoint file at this time. There are a number of ways to get FLASH to produce a checkpoint file for restarting. Within the `flash.par`, runtime parameters can be set to dump output. A checkpoint file can be dumped based on elapsed simulation time, elapsed wall clock time or the number of timesteps advanced. A checkpoint file is also produced when

the simulation ends, when the max simulation time `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 9.2: Checkpoint IO parameters.

Parameter	Type	Default value	Description
<code>checkpointFileNumber</code>	INTEGER	0	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.
<code>checkpointFileIntervalStep</code>	INTEGER	0	The number of timesteps desired between subsequent checkpoint files.
<code>checkpointFileIntervalTime</code>	REAL	1.	The amount of simulation time desired between subsequent checkpoint files.
<code>checkpointFileIntervalZ</code>	REAL	HUGE(1.)	The amount of cosmological redshift change that is desired between subsequent checkpoint files.
<code>rolling_checkpoint</code>	INTEGER	10000	The number of checkpoint files to keep available at any point in the simulation. If a checkpoint number is greater than <code>rolling_checkpoint</code> , then the checkpoint number is reset to 0. There will be at most <code>rolling_checkpoint</code> checkpoint files kept. This parameter is intended to be used when disk space is at a premium.
<code>wall_clock_checkpoint</code>	REAL	43200.	The maximum amount of wall clock time (seconds) to elapse between checkpoints. When the simulation is started, the current time is stored. If <code>wall_clock_checkpoint</code> seconds elapse over the course of the simulation, a checkpoint file is stored. This is useful for ensuring that a checkpoint file is produced before a queue closes.

Table 9.2: Checkpoint IO parameters (continued).

Parameter	Type	Default value	Description
<b>restart</b>	BOOLEAN	<b>.false.</b>	A logical variable indicating whether the simulation is restarting from a checkpoint file ( <b>.true.</b> ) or starting from scratch ( <b>.false.</b> ).
<b>appendParRestart</b>	BOOLEAN	<b>.false.</b>	A logical variable indicating whether the parameter file is automatically appended with the appropriate values of <b>restart</b> , <b>checkpointFileNumber</b> and <b>plotFileNumber</b> after a checkpoint file is written (except for the first checkpoint file).

FLASH is capable of restarting from any of the checkpoint files it produces. The user should make sure that the checkpoint file is valid (*e.g.*, the code did not stop while outputting). To tell FLASH to restart, set the **restart** runtime parameter to **.true.** in the **flash.par**. Also, set **checkpointFileNumber** to the number of the file from which you wish to restart. If plotfiles or particle files are being produced set **plotfileNumber** and **particleFileNumber** to the number of the *next* plotfile and particle file you want FLASH to output. In FLASH4 plotfiles and particle file outputs are forced whenever a checkpoint file is written. Sometimes several plotfiles may be produced after the last valid checkpoint file. Resetting **plotfileNumber** to the first plotfile produced after the checkpoint from which you are restarting will ensure that there are no gaps in the output. See [Section 9.2.2](#) for more details on plotfiles.

### 9.2.2 Plotfiles

A plotfile contains all the information needed to interpret the grid data maintained by FLASH. The data in plotfiles, including the grid metadata such as coordinates and block sizes, are stored at single precision to preserve space. This can, however, be overridden by setting the runtime parameters **plotfileMetadataDP** and/or **plotfileGridQuantityDP** to true to set the grid metadata and the quantities stored on the grid (dens, pres, temp, etc.) to use double precision, respectively. Users must choose which variables to output with the runtime parameters **plot\_var\_1**, **plot\_var\_2**, *etc.*, by setting them in the **flash.par** file. For example:

```
plot_var_1 = "dens"
plot_var_2 = "pres"
```

Currently, we support a number of plotvars named **plot\_var\_***n* up to the number of UNKVARs in a given simulation. Similarly, scratch variables may be output to plot files [Section 9.6](#). At this time, the plotting of face centered quantities is not supported.

#### FLASH Transition

In FLASH2 a few variables like density and pressure were output to the plotfiles by default. Because FLASH4 supports a wider range of simulations, it makes no assumptions that density or pressure variables are even included in the simulation. In FLASH4 a user *must* define plotfile variables in the **flash.par** file, otherwise the plotfiles will not contain any variables.

The interface for writing a plotfile is the routine **IO\_writePlotfile**. As with checkpoint files, the user will not need to call this routine directly because it is invoked indirectly through calling **IO\_output** when,

based on runtime parameters, FLASH4 needs to write a plotfile. FLASH can produce plotfiles in much the same manner as it does with checkpoint files. They can be dumped based on elapsed simulation time, on steps since the last plotfile dump or by forcing a plotfile to be written by hand by creating a `.dump_plotfile` in the output directory. A plotfile will also be written at the termination of a simulation as well.

If plotfiles are being kept at particular intervals (such as time intervals) for purposes such as visualization or analysis, it is also possible to have FLASH denote a plotfile as “forced”. This designation places the word forced between the basename and the file format type identifier (or the split number if splitting is used). These files are numbered separately from normal plotfiles. By default, plotfiles are considered forced if output for any reason other than the change in simulation time, change in cosmological redshift, change in step number, or the termination of a simulation from reaching `nend`, `zFinal`, or `tmax`. This option can be disabled by setting `ignoreForcedPlot` to true in a simulations `flash.par` file. The following runtime parameters pertain to controlling plotfiles:

Table 9.3: Plotfile IO parameters.

Parameter	Type	Default value	Description
<code>plotFileNumber</code>	INTEGER	0	The number of the starting (or restarting) plotfile. This number is appended to the file-name.
<code>plotFileIntervalTime</code>	REAL	1.	The amount of simulation time desired between subsequent plotfiles.
<code>plotFileIntervalStep</code>	INTEGER	0	The number of timesteps desired between subsequent plotfiles.
<code>plotFileIntervalZ</code>	INTEGER	HUGE(1.)	The change in cosmological redshift desired between subsequent plotfiles.
<code>plot_var_1, ..., plot_var_n</code>	STRING	"none"	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.
<code>ignoreForcedPlot</code>	BOOLEAN	.false.	A logical variable indicating whether or not to denote certain plotfiles as forced.
<code>forcedPlotfileNumber</code>	INTEGER	0	An integer that sets the starting number for a forced plotfile.
<code>plotfileMetadataDP</code>	BOOLEAN	.false.	A logical variable indicating whether or not to output the normally single-precision grid metadata fields as double precision in plotfiles. This specifically affects <code>coordinates</code> , <code>block size</code> , and <code>bounding box</code> .
<code>plotfileGridQuantityDP</code>	BOOLEAN	.false.	A logical variable that sets whether or not quantities stored on the grid, such as those stored in <code>unk</code> , are output in single precision or double precision in plotfiles.



### 9.2.3 Particle files

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

Table 9.4: Particle File IO runtime parameters.

Parameter	Type	Default value	Description
<code>particleFileNumber</code>	INTEGER	0	The number of the starting (or restarting) particle file. This number is appended to the end of the filename.
<code>particleFileIntervalTime</code>	REAL	1.	The amount of simulation time desired between subsequent particle file dumps.
<code>particleFileIntervalStep</code>	INTEGER	0	The number of timesteps desired between subsequent particle file dumps.
<code>particleFileIntervalZ</code>	REAL	HUGE(1.)	The change in cosmological redshift desired between subsequent particle file dumps.

#### FLASH Transition

From FLASH3 on each particle dump is written to a separate file. In FLASH2 the particles data structure was broken up into real and integer parts, where as in FLASH3 all particle properties are real values. See [Section 9.9](#) and [Chapter 21](#) for more information about the particles data structure in FLASH4. Additionally, filtered particles are not implemented in FLASH4.

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

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

will include the IO unit `IO/IOMain/hdf5/serial/UG` and the correct `IOParticles` subunit `IO/IOParticles/hdf5/serial/UG`. The shortcuts `+parallelcio`, `+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 ASCII 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.

Parameter	Type	Default value	Description
<code>stats_file</code>	STRING	"flash.dat"	Name of the file to which the integral quantities are written.
<code>wr_integrals_freq</code>	INTEGER	1	The number of timesteps to elapse between outputs to the scalar/integral data file ( <code>flash.dat</code> )

### 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.

Parameter	Type	Default value	Description
<code>basenm</code>	STRING	"flash_"	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 <a href="#">Section 9.8</a> for a description of how FLASH output files are named.
<code>output_directory</code>	STRING	""	Output directory for plotfiles, particle files and checkpoint files. The default is the directory in which the executable sits. <code>output_directory</code> can be an absolute or relative path.
<code>memory_stat_freq</code>	INTEGER	100000	The number of timesteps to elapse between memory statistic dumps to the log file ( <code>flash.log</code> ).
<code>useCollectiveHDF5</code>	BOOLEAN	.true.	When using the parallel HDF5 implementation of IO, will enable collective mode for HDF5.

Table 9.6: FLASH IO parameters (continued).

Parameter	Type	Default value	Description
<code>summaryOutputOnly</code>	BOOLEAN	<code>.false.</code>	When set to <code>.true.</code> write an integrated grid quantities file only. Checkpoint, plot and particle files are not written unless the user creates a <code>.dump_plotfile</code> , <code>.dump_checkpoint</code> , <code>.dump_restart</code> or <code>.dump_particle</code> file.

### 9.3 Restarts and Runtime Parameters

FLASH4 outputs the runtime parameters of a simulation to all checkpoint files. When a simulation is restarted, these values are known by the `RuntimeParameters` unit while the code is running. On a restart, all values from the checkpoint used in the restart are stored as previous values in the lists kept by the `RuntimeParameters` unit. All current values are taken from the defaults used by FLASH4 and any simulation parameter files (*e.g.*, `flash.par`). If needed, the previous values from the checkpoint file can be obtained using the routines `RuntimeParameters_getPrev`.

### 9.4 Output Scalars

In FLASH4, each unit has the opportunity to request scalar data to be output to checkpoint or plot-files. Because there is no central database, each unit “owns” different data in the simulation. For example, the `Driver` unit owns the timestep variable `dt`, the simulation variable `simTime`, and the simulation step number `nStep`. The `Grid` unit owns the sizes of each block, `nxb`, `nyb`, and `nzb`. The `IO` unit owns the variable `checkpointFileNumber`. Each of these quantities are output into checkpoint files. Instead of hard coding the values into checkpoint routines, FLASH4 offers a more flexible interface whereby each unit sends its data to the `IO` unit. The `IO` unit then stores these values in a linked list and writes them to the checkpoint file or plotfile. Each unit has a routine called “`Unit_sendOutputData`”, *e.g.*, `Driver_sendOutputData` and `Grid_sendOutputData`. These routines in turn call `IO_setScalar`. For example, the routine `Grid_sendOutputData` calls

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

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

After restarting a simulation from a checkpoint file, a unit might call `IO_getScalar` to reset a variable value. For example, the `Driver` unit calls `IO_getScalar("dt", dr_dt)` to get the value of the timestep `dt` reinitialized from the checkpoint file. A value from the checkpoint file can be obtained by calling `IO_getPrevScalar`. This call can take an optional argument to find out if an error has occurred in finding the previous value, most commonly because the value was not found in the checkpoint file. By using this argument, the user can then decide what to do if the value is not found. If the scalar value is not found and the optional argument is not used, then the subroutine will call `Driver_abortFlash` and terminate the run.

### 9.5 Output User-defined Arrays

Often in a simulation the user needs to output additional information to a checkpoint or plotfile which is not a grid scope variable. In FLASH2 any additional information had to be hard coded into the simulation. In FLASH4, we have provided a general interface `IO_writeUserArray` and `IO_readUserArray` which allows the user to write and read any generic array needed to be stored. The above two functions do not have

any implementation and it is up to the user to fill them in with the needed calls to the HDF5 or pnetCDF C routines. We provide implementation for reading and writing integer and double precision arrays with the helper routines `io_h5write_generic_iarr`, `io_h5write_generic_rarr`, `io_ncmpi_write_generic_iarr`, and `io_ncmpi_write_generic_rarr`. Data is written out as a 1-dimensional array, but the user can write multidimensional arrays simply by passing a reference to the data and the total number of elements to write. See these routines and the simulation `StirTurb` for details on their usage.

## 9.6 Output Scratch Variables

In FLASH4 a user can allocate space for a scratch or temporary variable with grid scope using one of the Config keywords `SCRATCHVAR`, `SCRATCHCENTERVAR`, `SCRATCHFACEXVAR`, `SCRATCHFACEYVAR` or `SCRATCHFACEZVAR` (see [Section 5.5.1](#)). To output these scratch variables, the user only needs to set the values of the runtime parameters `plot_grid_var_1`, `plot_grid_var_2`, etc., by setting them in the `flash.par` file. For example to output the magnitude of vorticity with a declaration in a Config file of `SCRATCHVAR mvrt`:

```
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:

$$\text{basename\_s0000-} \left\{ \begin{array}{c} \text{hdf5} \\ \text{ncmpi} \end{array} \right\} \text{-chk.0000} ,$$

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:

$$\text{basename\_s0000-} \left\{ \begin{array}{c} \text{hdf5} \\ \text{ncmpi} \end{array} \right\} \text{-plt-} \left\{ \begin{array}{c} \text{crn} \\ \text{cnt} \end{array} \right\} \text{-0000} ,$$

where `hdf5` or `ncmpi` is picked depending on the IO implementation used, `crn` and `cnt` indicate data stored at the cell corners or centers respectively, the number following “s” is the split file number, if used, and the number at the end of the filename is the current value of `plotfileNumber`. `crn` is reserved, even though corner data output is not presently supported by FLASH4’s IO.

### FLASH Transition

In FLASH2 the correct format of the names of the checkpoint, plotfile and particle file were necessary in order to read the files with the FLASH `fidlr` visualization tool. In FLASH4 the name of the file is irrelevant to `fidlr3.0` (see [Chapter 39](#)). 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 most widely used IO library although Parallel-NetCDF is rapidly gaining acceptance among the high performance computing community. In FLASH4 we also offer a serial direct FORTRAN IO which is currently only implemented for the uniform grid. This option is intended to provide users a way to output data if they do not have access to HDF5 or PnetCDF. Additionally, if HDF5 or PnetCDF are not performing well on a given platform the direct IO implementation can be used as a last resort. Our tools, `fidlr` and `sfocu` (Part X), 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 <https://support.hdfgroup.org/documentation/>. The IO in FLASH4 implementations require HDF5 1.4.0 or later. Please note that HDF5 1.6.2 requires IDL 1.6 or higher in order to use `fidlr3.0` for post processing.

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

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

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

The default IO implementation is `IO/IOMain/hdf5/serial/PM`. It can be included simply by adding `-unit=IO` to the `setup` line. In FLASH4, the user can set up shortcuts for various implementations. See Chapter 5 for more information about creating shortcuts.

The format of the HDF5 output files produced by these various IO implementations is identical; only the method by which they are written differs. It is possible to create a checkpoint file with the parallel routines and restart FLASH from that file using the serial routines or vice-versa. (This switch would require resetting up and compiling a code to get an executable with the serial version of IO.) When outputting with the Uniform Grid, some data is stored that isn't explicitly necessary for data analysis or visualization, but is retained to keep the output format of PARAMESH the same as with the Uniform Grid. See Section 9.9.1.3 for more information on output data formats. For example, the refinement level in the Uniform Grid case is always equal to 1, as is the nodetype array. A tree structure for the Uniform Grid is 'faked' for visualization purposes. In a similar way, the non-fixedblocksize mode outputs all of the data stored by the grid as though it is one large block. This allows restarting with differing numbers of processors and decomposing the domain in an arbitrary fashion in Uniform Grid.

Parallel HDF5 mode has two runtime parameters useful for debugging: `chkGuardCellsInput` and `chkGuardCellsOutput`. When these runtime parameters are true, the FLASH4 input and output routines read and/or output the guard cells in addition to the normal interior cells. Note that the HDF5 files produced are *not* compatible with the visualization and analysis tools provided with FLASH4.

### FLASH Transition

We recommend that HDF5 version 1.6 or later be used with the HDF5 IO implementations with FLASH4. While it is possible to use any version of HDF5 1.4.0 or later, files produced with versions predating version 1.6 will not be compatible with code using the libraries post HDF5 1.6.

### Caution

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

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

#### 9.9.1.1 Collective Mode

By default, the parallel mode of HDF5 uses an independent access pattern for writing datasets and performs IO without aggregating the disk access for writing. Parallel HDF5 can also be run so that the writes to the file’s datasets are aggregated, allowing the data from multiple processors to be written to disk in fewer operations. This can greatly increase the performance of IO on filesystems that support this behavior. FLASH4 can make use of this mode by setting the runtime parameter `useCollectiveHDF5` to true.

#### 9.9.1.2 Machine Compatibility

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

#### 9.9.1.3 HDF5 Data Format

The HDF5 data format for FLASH4 is identical to FLASH2 for all grid variables and datastructures used to recreate the tree and neighbor data with the exception that `bounding box`, `coordinates`, and `block size` are now sized as `mdim`, or the maximum dimensions supported by FLASH’s grids, which is three, rather than `ndim`. `PARAMESH 4.0` and `PARAMESH 4dev`, however, do requires a few additional tree data structures to be output which are described below. The format of the metadata stored in the HDF5 files has changed to reduce the number of ‘writes’ required. Additionally, scalar data, like `time`, `dt`, `nstep`, *etc.*, are now stored in a linked list and written all at one time. Any unit can add scalar data to the checkpoint file by calling the routine `IO_setScalar`. See [Section 9.4](#) for more details. The FLASH4 HDF5 format is summarized in [Table 9.7](#).

Table 9.7: FLASH HDF5 file format.

Record label	Description of the record
<i>Simulation Meta Data: included in all files</i>	

Table 9.7: HDF5 format (continued).

Record label	Description of the record
sim info	Stores simulation meta data in a user defined C structure. Structure datatype and attributes of the structure are described below.
<pre> 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; </pre>	
<code>sim_info.file_format_version:</code>	An integer giving the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file.
<code>sim_info.setup_call:</code>	The complete syntax of the <code>setup</code> command used when creating the current FLASH executable.
<code>sim_info.file_creation_time:</code>	The time and date that the file was created.
<code>sim_info.flash_version:</code>	The version of FLASH used for the current simulation. This is returned by routine <code>setup_flashVersion</code> .
<code>sim_info.build_date:</code>	The date and time that the FLASH executable was compiled.
<code>sim_info.build_dir:</code>	The complete path to the FLASH root directory of the source tree used when compiling the FLASH executable. This is generated by the subroutine <code>setup_buildstats</code> which is created at compile time by the Makefile.
<code>sim_info.build_machine:</code>	The name of the machine (and anything else returned from <code>uname -a</code> ) on which FLASH was compiled.
<code>sim_info.cflags:</code>	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.
<code>sim_info.fflags:</code>	The f compiler flags used in the given simulation.

Table 9.7: HDF5 format (continued).

Record label	Description of the record
<code>sim_info.setup_time_stamp:</code>	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.
<code>sim_info.build_time_stamp:</code>	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.

*RuntimeParameter and Scalar data*

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

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

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

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

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

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

integer runtime parameters

`int_list_t int_list(numIntParams)`

A linked list holding the names and values of all the integer runtime parameters.

real runtime parameters

`real_list_t real_list(numRealParams)`

A linked list holding the names and values of all the real runtime parameters.

string runtime parameters

`str_list_t str_list(numStrParams)`

A linked list holding the names and values of all the string runtime parameters.



Table 9.7: HDF5 format (continued).

Record label	Description of the record
logical runtime parameters	<code>log_list_t log_list(numLogParams)</code> A linked list holding the names and values of all the logical runtime parameters.
integer scalars	<code>int_list_t int_list(numIntScalars)</code> A linked list holding the names and values of all the integer scalars.
real scalars	<code>real_list_t real_list(numRealScalars)</code> A linked list holding the names and values of all the real scalars.
string scalars	<code>str_list_t str_list(numStrScalars)</code> A linked list holding the names and values of all the string scalars.
logical scalars	<code>log_list_t log_list(numLogScalars)</code> A linked list holding the names and values of all the logical scalars.
<hr/> <i>Grid data: included only in checkpoint files and plotfiles</i>	
unknown names	<code>character*4 unk_names(nvar)</code> This array contains four-character names corresponding to the first index of the <code>unk</code> array. They serve to identify the variables stored in the ‘unknowns’ records.
refine level	<code>integer lrefine(globalNumBlocks)</code> This array stores the refinement level for each block.
node type	<code>integer nodetype(globalNumBlocks)</code> 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	<code>integer gid(nfaces+1+nchild,globalNumBlocks)</code> This is the global identification array. For a given block, this array gives the block number of the blocks that neighbor it and the block numbers of its parent and children.
coordinates	<code>real coord(mdim,globalNumBlocks)</code> This array stores the coordinates of the center of the block. <code>coord(1,blockID) = x-coordinate</code> <code>coord(2,blockID) = y-coordinate</code> <code>coord(3,blockID) = z-coordinate</code>

Table 9.7: HDF5 format (continued).

Record label	Description of the record
block size	<p><code>real size(mdim,globalNumBlocks)</code></p> <p>This array stores the dimensions of the current block.</p> <p><code>size(1,blockID) = x size</code>  <code>size(2,blockID) = y size</code>  <code>size(3,blockID) = z size</code></p>
bounding box	<p><code>real bnd_box(2,mdim,globalNumBlocks)</code></p> <p>This array stores the minimum (<code>bnd_box(1, :, :)</code>) and maximum (<code>bnd_box(2, :, :)</code>) coordinate of a block in each spatial direction.</p>
which child ( <i>Paramesh4.0 and Paramesh4dev only!</i> )	<p><code>integer which_child(globalNumBlocks)</code></p> <p>An integer array identifying which part of the parents' volume this child corresponds to.</p>
variable	<p><code>real unk(nxb,nyb,nzb,globalNumBlocks)</code></p> <p><code>nxb</code> = number of cells/block in <i>x</i>  <code>ny</code> = number of cells/block in <i>y</i>  <code>nz</code> = number of cells/block in <i>z</i></p> <p>This array holds the data for a single variable. The record label is identical to the four-character variable name stored in the record <i>unknown names</i>. Note that, for a plot file with <code>CORNERS=.true.</code> in the parameter file, the information is interpolated to the cell corners and stored.</p>
<i>Particle Data: included in checkpoint files and particle files</i>	
localnp	<p><code>integer localnp(globalNumBlocks)</code></p> <p>This array holds the number of particles on each processor.</p>
particle names	<p><code>character*24 particle_labels(NPART.PROPS)</code></p> <p>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.</p>
tracer particles	<p><code>real particles(NPART.PROPS, globalNumParticles)</code></p> <p>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.</p>

### 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 particle file will be broken into 4 files, by processor number). This feature is only available with the HDF5 parallel IO mode, and is still experimental. Users should use this at their own risk.

## 9.9.2 Parallel-NetCDF

Another implementation of the IO unit uses the Parallel-NetCDF library available at <http://www.mcs.anl.gov/parallel-netcdf/>. At this time, the FLASH code requires version 1.1.0 or higher. Our testing shows performance of PNetCDF library to be very similar to HDF5 library when using collective I/O optimizations in parallel I/O mode.

There are two different PnetCDF IO unit implementations. Both are parallel implementations, one for each supported grid, the Uniform Grid and PARAMESH. It is important to remember to match the IO implementation with the correct grid. To include PnetCDF IO in a simulation the user should add `-unit=IO/IOMain/pnetcdf.....` to the `setup` line. See examples below for the two different PnetCDF IO implementations.

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

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

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

## 9.9.3 Direct IO

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

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

## 9.9.4 Output Side Effects

In FLASH4 when plotfiles or checkpoint files are output by `IO.output`, the grid is fully restricted and user variables are computed prior to writing the file. `IO.writeCheckpoint` and `IO.writePlotfile` by default, do not do this step themselves. The restriction can be forced for all writes by setting runtime parameter `alwaysRestrictCheckpoint` to true and the user variables can always be computed prior to output by setting `alwaysComputeUserVars` to true.

## 9.10 Working with Output Files

The checkpoint file output formats offer great flexibility when visualizing the data. The visualization program does not have to know the details of how the file was written; rather it can query the file to find the number of dimensions, block sizes, variable data etc that it needs to visualize the data. IDL routines for reading HDF5 and PnetCDF formats are provided in `tools/fidlr3/`. These can be used interactively through the IDL command line (see [Chapter 39](#)). In addition, ViSit version 10.0 and higher (see [Chapter 36](#)) can natively read FLASH4 HDF5 output files by using the command line option `-assume_format FLASH`.

## 9.11 Unit Test

The IO unit test is provided to test IO performance on various platforms with the different FLASH IO implementations and parallel libraries.

### FLASH Transition

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

The `unitTest` is setup like any other FLASH4 simulation. It can be run with any IO implementation as long as the correct Grid implementation is included. This `unitTest` writes a checkpoint file, a plotfile, and if particles are included, a particle file. Particles IO can be tested simply by including particles in the simulation. Variables needed for particles should be uncommented in the `Config` file.

Example setups:

```
#setup for PARAMESH Grid and serial HDF5 io
./setup unitTest/IO -auto
```

```
#setup for PARAMESH Grid with parallel HDF5 IO (see shortcuts docs for explanation)
./setup unitTest/IO -auto +parallelIO      (same as)
./setup unitTest/IO -auto -unit=IO/IOMain/hdf5/parallel/PM
```

```
#setup for Uniform Grid with serial HDF5 IO, 3d problem, increasing default number of zones
./setup unitTest/IO -3d -auto +ug -nxb=16 -nyb=16 -nzb=16 (same as)
./setup unitTest/IO -3d -auto -unit=Grid/GridMain/UG -nxb=16 -nyb=16 -nzb=16
```

```
#setup for PM3 and parallel netCDF, with particles
./setup unitTest/IO -auto -unit=Particles +pnetcdf
```

```
#setup for UG and parallel netCDF
./setup unitTest/IO -auto +pnetcdf +ug
```

Run the test like any other FLASH simulation:

```
mpirun -np numProcs flash3
```

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

- The `Config` file in `unitTest/IO` declares some dummy grid scope variables which are stored in the `unk` array. If the user wants a more intensive IO test, more variables can be added. Variables are initialized to dummy values in `Driver_evolveFlash`.
- Variables will only be output to the plotfile if they are declared in the `flash.par` (see the example `flash.par` in the unit test).

- The only units besides the simulation unit included in this simulation are `Grid`, `I/O`, `Driver`, `Timers`, `Logfile`, `RuntimeParameters` and `PhysicalConstants`.
- If the `PARAMESH` Grid implementation is being used, it is important to note that the grid will not refine on its own. The user should set `lrefine_min` to a value  $> 1$  to create more blocks. The user could also set the runtime parameters `nblockx`, `nblocky`, `nblockz` to make a bigger problem.
- Just like any other simulation, the user can change the number of zones in a simulation using `-nxb=numZones` on the setup line.

## 9.12 Chombo

In FLASH4 we introduced a new `Grid` implementation that depends on `Chombo` library (see Section 8.7). An application built with this `Grid` implementation will include `I/O/IOMain/chombo` by default. No other I/O implementation is compatible with `Chombo Grid` implementation. As before, I/O can be excluded from an application using the `+noio` setup shortcut.

Data files are written in standard `Chombo` file layout in HDF5 format. The visualization tool `VisIt` supports `Chombo` file layout, but other tools, such as `fidlr` and `sfocu`, are incompatible with `Chombo` file layout.

## 9.13 Derived data type I/O

In FLASH4 we introduced an alternative I/O implementation for both HDF5 and Parallel-NetCDF which is a slight spin on the standard parallel I/O implementations. In this new implementation we select the data from the mesh data structures directly using HDF5 hyperslabs (HDF5) and MPI derived datatypes (Parallel-NetCDF) and then write the selected data to datasets in the file. This eliminates the need for manually copying data into a FLASH allocated temporary buffer and then writing the data from the temporary buffer to disk.

You can include derived data type I/O in your FLASH application by adding the setup shortcuts `+hdf5TypeI/O` for HDF5 and `+pnetTypeI/O` for Parallel-NetCDF to your setup line. If you are using the HDF5 implementation then you need a parallel installation of HDF5. All of the runtime parameters introduced in this chapter should be compatible with derived data type I/O.

A nice property of derived data type I/O is that it eliminates a lot of the I/O code duplication which has been spreading in the FLASH I/O unit over the last decade. The same code is used for UG, NoFBS and Paramesh FLASH applications and we have also shared code between the HDF5 and Parallel-NetCDF implementations. A technical reason for using the new I/O implementation is that we provide more information to the I/O libraries about the exact data we want to read from / write to disk. This allows us to take advantage of recent enhancements to I/O libraries such as the nonblocking APIs in the Parallel-NetCDF library. We discuss experimentation with this API and other ideas in the paper “A Case Study for Scientific I/O: Improving the FLASH Astrophysics Code” [www.mcs.anl.gov/uploads/cels/papers/P1819.pdf](http://www.mcs.anl.gov/uploads/cels/papers/P1819.pdf)

The new I/O code has been tested in our internal FLASH regression tests from before the FLASH4 release and there are no known issues, however, it will probably be in the release following FLASH4 when we will recommend using it as the default implementation. We have made the research ideas from our case study paper usable for all FLASH applications, however, the code still needs a clean up and exhaustive testing with all the FLASH runtime parameters introduced in this chapter.



## Chapter 10

# Runtime Parameters Unit

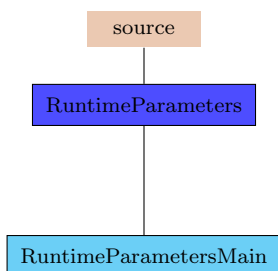


Figure 10.1: The `RuntimeParameters` unit directory tree.

The `RuntimeParameters` Unit stores and maintains a global linked lists of runtime parameters that are used during program execution. Runtime parameters can be added to the lists, have their values modified, and be queried. This unit handles adding the default runtime parameters to the lists as well as reading any overwritten parameters from the `flash.par` file.

### 10.1 Defining Runtime Parameters

All parameters must be declared in a `Config` file with the keyword declaration `PARAMETER`. In the `Config` file, assign a data type and a default value for the parameter. If possible, assign a range of valid values for the parameter. You can also provide a short description of the parameter's function in a comment line that begins with `D`.

```
#section of Config file for a Simulation
D myParameter Description of myParameter
PARAMETER myParameter REAL 22.5 [20 to 60]
```

To change the runtime parameter's value from the default, assign a new value in the `flash.par` for the simulation.

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

See [Section 5.5](#) for more information on declaring parameters in a `Config` file.

### 10.2 Identifying Valid Runtime Parameters

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

```

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

```

Figure 10.2: Section of output log showing runtime parameters values

```

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

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

```

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

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

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

## 10.3 Routine Descriptions

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

The user will typically only have to use one routine from the Runtime Parameters API, `RuntimeParameters.get`. This routine retrieves the value of a parameter stored in the linked list in the `RuntimeParameters_data` module. In FLASH4 the value of runtime parameters for a given unit are stored in that unit's `Unit_data` Fortran module and they are typically initialized in the unit's `Unit_init` routine. Each unit's 'init' routine is only called once at the beginning of the simulation by `Driver_initFlash`. For more documentation on the FLASH code architecture please see [Chapter 4](#). It is important to note that even though runtime parameters are declared in a specific unit's `Config` file, the runtime parameters linked



list is a global space and so any unit can fetch a parameter, even if that unit did not declare it. For example, the `Driver` unit declares the logical parameter `restart`, however, many units, including the `IO` unit get `restart` parameter with the `RuntimeParameters_get` interface. If a section of the code asks for a runtime parameter that was not declared in a `Config` file and thus is not in the runtime parameters linked list, the FLASH code will call `Driver_abortFlash` and stamp an error to the `logfile`. The other `RuntimeParameter` routines in the API are not generally called by user routines. They exist because various other units within FLASH need to access parts of the `RuntimeParameters` interface. For example, the input/output unit `IO` needs `RuntimeParameters_set`. There are no user-defined parameters which affect the `RuntimeParameters` unit.

### FLASH Transition

FLASH no longer distinguishes between contexts as in FLASH2. All runtime parameters are stored in the same context, so there is no need to pass a 'context' argument.

## 10.4 Example Usage

An implementation example from the `IO_init` is straightforward. First, use the module containing definitions for the unit (for `_init` subroutines, the usual `use Unit_data, ONLY:` structure is waived). Next, use the module containing interface definitions of the `RuntimeParameters` unit, *i.e.*, `use RuntimeParameters_interface, ONLY:`. Finally, read the runtime parameters and store them in unit-specific variables.

```
subroutine IO_init()

  use IO_data
  use RuntimeParameters_interface, ONLY : RuntimeParameters_get
  implicit none

  call RuntimeParameters_get('plotFileNumber',io_plotFileNumber)
  call RuntimeParameters_get('checkpointFileNumber',io_checkpointFileNumber)

  call RuntimeParameters_get('plotFileIntervalTime',io_plotFileIntervalTime)
  call RuntimeParameters_get('plotFileIntervalStep',io_plotFileIntervalStep)
  call RuntimeParameters_get('checkpointFileIntervalTime',io_checkpointFileIntervalTime)
  call RuntimeParameters_get('checkpointFileIntervalStep',io_checkpointFileIntervalStep)

  !! etc ...
```

Note that the parameters found in the `flash.par` or in the `Config` files, for example `plotFileNumber`, are generally stored in a variable of the same name with a unit prefix prepended, for example `io_plotFileNumber`. In this way, a program segment clearly indicates the origin of variables. Variables with a unit prefix (*e.g.*, `io_` for `IO`, `pt_` for particles) have been initialized from the `RuntimeParameters` database, and other variables are locally declared. When creating new simulations, runtime parameters used as variables should be prefixed with `sim_`.



# Chapter 11

## Multispecies Unit

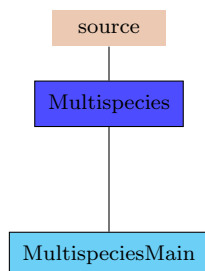


Figure 11.1: The `Multispecies` unit directory tree.

FLASH has the ability to track multiple fluids, each of which can have its own properties. The `Multispecies` unit handles setting and querying on the properties of fluids, as well as some common operations on properties. The advection and normalization of species is described in the context of the `Hydro` unit in [Chapter 15](#).

### 11.1 Defining Species

The names and properties of fluids are accessed by using their constant integer values defined in the `Flash.h` header file. The species names are defined in a `Config` file. The names of the species, for example `AIR`, `NI56`, are given in the `Config` file with keyword `SPECIES`.

In the traditional method for defining species, this `Config` would typically be the application's `Config` file in the `Simulation` unit. In the alternative method described below in [Section 11.4](#), `SPECIES` are normally not listed explicitly in the `Simulation` unit `Config`, but instead are automatically generated by `Multispecies/MultispeciesMain/Config` based on the contents of the `species` setup variable. Either way, the `setup` procedure transforms those names into accessor integers with the appended description `_SPEC`.

These names are stored in the `Flash.h` file. The total number of species defined is also defined within `Flash.h` as `NSPECIES`, and the integer range of their definition is given by `SPECIES_BEGIN` and `SPECIES_END`. To access the species in your code, use the index listed in `Flash.h`, for example `AIR_SPEC`, `NI56_SPEC`.

Note that `NSPECIES`, `SPECIES_BEGIN`, and `SPECIES_END` are always defined, whether a simulation uses multiple species or not (and whether the simulation includes the `Multispecies` unit or not). However, if `NSPECIES=0`, `SPECIES_END` will be less than `SPECIES_BEGIN`, and then neither of them should be used as an index into solution vectors.

As an illustration, [Figure 11.2](#) and [Figure 11.3](#) are snippets from a configuration file and the corresponding section of the FLASH header file, respectively. For more information on `Config` files, see

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

Figure 11.2: Sample Config file showing how to define required fluid species.

```
#define SPECIES_BEGIN (NPROP_VARS + CONSTANT_ONE)
#define AIR_SPEC 11
#define SF6_SPEC 12
#define NSPECIES 2
#define SPECIES_END (SPECIES_BEGIN + NSPECIES - CONSTANT_ONE)
```

Figure 11.3: Sample excerpt from header file `Flash.h` showing integer definition of fluid species.

[Section 5.5](#); for more information on the `setup` procedure, see [Chapter 5](#); for more information on the structure of the main header file `Flash.h`, see [Chapter 6](#).

### FLASH Transition

In FLASH2, you found the integer index of a species by using `find.fluid.index`. In FLASH4, the species index is always available because it is defined in `Flash.h`. Use the index directory, as in `xIn(NAME_SPEC - SPECIES_BEGIN + 1) = solnData(NAME_SPEC,i,j,k)`. But be careful that the species name is really defined in your simulation! You can test with

```
if (NAME_SPEC /= NONEXISTENT) then
  okVariable = solnData(NAME_SPEC,i,j,k)
endif
```

The available properties of an individual fluid are listed in [Table 11.1](#) and are defined in file `Multispecies.h`. In order to reference the properties in code, you must `#include` the file `Multispecies.h`. The initialization of properties is described in the following section.

## 11.2 Initializing Species Information in `Simulation_initSpecies`

Before you can work with the properties of a fluid, you must initialize the data in the `Multispecies` unit. Normally, initialization is done in the routine `Simulation_initSpecies`. An example procedure is shown below and consists of setting relevant properties for all fluids/SPECIES defined in the `Config` file. Fluids do not have to be isotopes; any molecular substance which can be defined by the properties shown in [Figure 11.4](#) is a valid input to the `Multispecies` unit.

### FLASH Transition

For nuclear burning networks, a `Simulation_initSpecies` routine is already predefined. It automatically initializes all isotopes found in the `Config` file. To use this shortcut, `REQUIRE` the module `Simulation/SimulationComposition` in the `Config` file.

Table 11.1: Properties available through the Multispecies unit.

Property Name	Description	Data type
A	Number of protons and neutrons in nucleus	real
Z	Atomic number	real
N	Number of neutrons	real
E	Number of electrons	real
BE	Binding Energy	real
GAMMA	Ratio of heat capacities	real
MS_ZMIN	Minimum allowed average ionization	real
MS_EOSTYPE	EOS type to use for MTMMMT EOS	integer
MS_EOSSUBTYPE	EOS subtype to use for MTMMMT EOS	integer
MS_EOSZFREEFILE	Name of file with ionization data	string
MS_EOSENERFILE	Name of file with internal energy data	string
MS_EOSPRESFILE	Name of file with pressure data	string
MS_NUMELEMS	Number of elements comprising this species	integer
MS_ZELEMS	Atomic number of each species element	array(integer)
MS_AELEMS	Mass number of each species element	array(real)
MS_FRACTIONS	Number fraction of each species element	array(real)
MS_OPLOWTEMP	Temperature at which cold opacities are used	real

```

subroutine Simulation_initSpecies()

implicit none
#include "Multispecies.h"
#include "Flash.h"

! These two variables are defined in the Config file as
! SPECIES SF6 and SPECIES AIR
call Multispecies_setProperty(SF6_SPEC, A, 146.)
call Multispecies_setProperty(SF6_SPEC, Z, 70.)
call Multispecies_setProperty(SF6_SPEC, GAMMA, 1.09)

call Multispecies_setProperty(AIR_SPEC, A, 28.66)
call Multispecies_setProperty(AIR_SPEC, Z, 14.)
call Multispecies_setProperty(AIR_SPEC, GAMMA, 1.4)
end subroutine Simulation_initSpecies

```

Figure 11.4: A Simulation\_initSpecies.F90 file showing Multispecies initialization

```

#include "Flash.h"
#include "Multispecies.h"

! Create arrays to store constituent element data. Note that these
! arrays are always of length MS_MAXELEMS.
real :: aelems(MS_MAXELEMS)
real :: fractions(MS_MAXELEMS)
integer :: zelems(MS_MAXELEMS)

call Multispecies_setProperty(H2O_SPEC, A, 18.0/3.0) ! Set average mass number
call Multispecies_setProperty(H2O_SPEC, Z, 10.0/3.0) ! Set average atomic number
call Multispecies_setProperty(H2O_SPEC, GAMMA, 5.0/3.0)
call Multispecies_setProperty(H2O_SPEC, MS_NUMELEMS, 2)

aelems(1) = 1.0 ! Hydrogen
aelems(2) = 16.0 ! Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_AELEMS, aelems)

zelems(1) = 1 ! Hydrogen
zelems(2) = 8 ! Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_ZELEMS, zelems)

fractions(1) = 2.0/3.0 ! Two parts Hydrogen
fractions(2) = 1.0/3.0 ! One part Oxygen
call Multispecies_setProperty(H2O_SPEC, MS_FRACTIONS, fractions)

```

Figure 11.5: A `Simulation_initSpecies.F90` file showing Multispecies initialization

### 11.3 Specifying Constituent Elements of a Species

A species can represent a specific isotope or a single element or a more complex material. Some units in FLASH require information about the elements that constitute a single species. For example, water is comprised of two elements: Hydrogen and Oxygen. The `Multispecies` database can store a list of the atomic numbers, mass numbers, and relative number fractions of each of the elements within a given species. This information is stored in the array properties `MS_ZELEMS`, `MS_AELEMS`, and `MS_FRACTIONS` respectively. The property `MS_NUMELEMS` contains the total number of elements for a species (`MS_NUMELEMS` would be two for water since water is made of Hydrogen and Oxygen). There is an upper bound on the number of elements for a single species which is defined using the preprocessor symbol `MS_MAXELEMS` in the “Flash.h” header file and defaults to six. The value of `MS_MAXELEMS` can be changed using the `ms_maxelems` setup variable. [Figure 11.5](#) shows an example of how the constituent elements for water can be set using the `Simulation_initSpecies` subroutine.

The constituent element information is optional and is only needed if a particular unit of interest requires it. At present, only the analytic cold opacities used in the `Opacity` unit make use of the constituent element information.

### 11.4 Alternative Method for Defining Species

[Section 11.1](#) described how species can be defined by using the `SPECIES` keyword in the `Config` file. [Section 11.2](#) then described how the properties of the species can be set using various subroutines defined in the `Multispecies` unit. There is an alternative to these approaches which uses setup variables to define the species, then uses runtime parameters to set the properties of each species. This allows users to change the number and names of species without modifying the `Config` file and also allows users to change properties without recompiling the code.

Table 11.2: Automatically Generated Multispecies Runtime Parameters

Property Name	Runtime Parameter Name
A	ms_<spec>A
Z	ms_<spec>Z
N	ms_<spec>Neutral
E	ms_<spec>Negative
BE	ms_<spec>BindEnergy
GAMMA	ms_<spec>Gamma
MS_ZMIN	ms_<spec>Zmin
MS_EOSTYPE	eos_<spec>EosType
MS_EOSSUBTYPE	eos_<spec>SubType
MS_EOSZFREEFILE	eos_<spec>TableFile
MS_EOSENERFILE	eos_<spec>TableFile
MS_EOSPRESFILE	eos_<spec>TableFile
MS_NUMELEMS	ms_<spec>NumElems
MS_ZELEMS	ms_<spec>ZElems_<N>
MS_AELEMS	ms_<spec>AElems_<N>
MS_FRACTIONS	ms_<spec>Fractions_<N>
MS_OPLOWTEMP	op_<spec>LowTemp

Species can be defined using the `species` setup variable. For example, to create two species called AIR and SF6 one would specify `species=air,sf6` in the simulation setup command. Using this setup variable and using the `SPECIES` keyword in the `Config` file are mutually exclusive. Thus, the user must choose which method they wish to use for a given simulation. Certain units, such as the `Opacity` unit, requires the use of the setup variable.

When species are defined using the setup variable approach, the `Multispecies` unit will automatically define several runtime parameters for each species. These runtime parameters can be used set the properties shown in [Table 11.1](#). The runtime parameter names contain the species name. [Table 11.2](#) shows an example of the mapping between runtime parameters and `Multispecies` properties, where `<spec>` is replaced by the species name as specified in the species setup argument list. Some of these runtime parameters are arrays, and thus the `<N>` is a number ranging from 1 to `MS_MAXELEMS`. The `Simulation_initSpecies` subroutine can be used to override the runtime parameter settings.

## 11.5 Routine Descriptions

We now briefly discuss some interfaces to the multifluid database that are likely of interest to the user. Many of these routines include optional arguments.

- **Multispecies.setProperty** This routine sets the value species property. It should be called within the subroutine **Simulation\_initSpecies** for all the species of interest in the simulation problem, and for all the required properties (any of A, Z, N, E, EB, GAMMA).

### FLASH Transition

In FLASH2, you could set multiple properties at once in a call to `add_fluid_to_db`. In FLASH4, individual calls are required. If you are setting up a nuclear network, there is a precoded **Simulation\_initSpecies** to easily initialize all necessary species. It is located in the unit `Simulation/SimulationComposition`, which must be listed in your simulation `Config` file.

- **Multispecies.getProperty** Returns the value of a requested property.
- **Multispecies.getSum** Returns a weighted sum of a chosen property of species. The total number of species can be subset. The weights are optional, but are typically the mass fractions  $X_i$  of each of the fluids at a point in space. In that case, if the selected property (one of  $A_i$ ,  $Z_i$ ,  $\dots$ ,  $\gamma_i$ ) is denoted  $\mathcal{P}_i$ , the sum calculated is

$$\sum_i X_i \mathcal{P}_i \quad .$$

- **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  $A_i$ ,  $Z_i$ ,  $\dots$ ,  $\gamma_i$ ) is denoted  $\mathcal{P}_i$ , the average calculated is

$$\frac{1}{N} \sum_i^N w_i \mathcal{P}_i \quad ,$$

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  $A_i$ ,  $Z_i$ ,  $\dots$ ,  $\gamma_i$ ) is denoted  $\mathcal{P}_i$ , the sum calculated is

$$\sum_i^N \frac{w_i}{\mathcal{P}_i} \quad .$$

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

$$\frac{1}{\bar{A}} = \sum_i \frac{X_i}{A_i} \quad , \tag{11.1}$$

where  $X_i$  is the mass fraction of species  $i$ , and  $A_i$  is the atomic mass of that species. To compute  $\bar{A}$  using the multifluid database, one would use the following lines

```
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  $A_i$ ,  $Z_i$ ,  $\dots$ ,  $\gamma_i$ ) is denoted  $\mathcal{P}_i$ , the sum calculated is

$$\sum_i \frac{X_i}{A_i} \mathcal{P}_i \quad .$$

- **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  $A_i$ ,  $Z_i$ ,  $\dots$ ,  $\gamma_i$ ) is denoted  $\mathcal{P}_i$ , the sum calculated is

$$\sum_i^N w_i \mathcal{P}_i^2 \quad .$$

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



## 11.6 Example Usage

In general, to use Multispecies properties in a simulation, the user must only properly initialize the species as described above in the `Simulation_init` routine. But to program with the Multispecies properties, you must do three things:

- `#include` the `Flash.h` file to identify the defined species
- `#include` the `Multispecies.h` file to identify the desired property
- use the Fortran interface to the Multispecies unit because the majority of the routines are overloaded.

The example below shows a snippet of code to calculate the electron density.

```
...
#include Flash.h
#include Multispecies.h

USE Multispecies_interface, ONLY: Multispecies_getSumInv, Multispecies_getSumFrac
...
do k=blkLimitsGC(LOW,KAXIS),blkLimitsGC(HIGH,KAXIS)
  do j=blkLimitsGC(LOW,JAXIS),blkLimitsGC(HIGH,JAXIS)
    do i=blkLimitsGC(LOW,IAXIS),blkLimitsGC(HIGH,IAXIS)
      call Multispecies_getSumInv(A,abar_inv)
      abar = 1.e0 / abar_inv
      call Multispecies_getSumFrac(Z,zbar)
      zbar = abar * zbar
      ye(i,j,k) = abar_inv*zbar
    enddo
  enddo
enddo
...
```

## 11.7 Unit Test

The unit test for `Multispecies` provides a complete example of how to call the various API routines in the unit with all variations of arguments. Within `Multispecies_unitTest`, incorrect usage is also indicated within commented-out statements.



## Chapter 12

# Physical Constants Unit

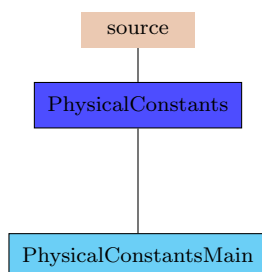


Figure 12.1: The `PhysicalConstants` unit directory tree.

The Physical Constants unit provides a set of common constants, such as Pi and the gravitational constant, in various systems of measurement units. The default system of units is CGS, so named for having a length unit in centimeters, a mass unit in grams, and a time unit in seconds. In CGS, the charge unit is the esu, and the temperature unit is the Kelvin. The constants can also be obtained in the standard MKS system of units, where length is in meters, mass in kilograms, and time in seconds. For MKS units, charge is in Coloumbs, and temperature in Kelvin.

### FLASH Transition

For ease of usage, the constant  $\text{PI}=3.14159\dots$  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  $\text{Mpc}^3 \text{Gyr}^{-2} M_{\odot}^{-1}$  by calling

```
call PhysicalConstants_get ("Newton", G, len_unit="Mpc",  
                           time_unit="Gyr", mass_unit="Msun")
```

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

Physical constants are taken from K. Nahamura *et al.* (Particle Data Group), J. Phys. G **37**, 075021 (2010).

Table 12.1: Available Physical Constants

String Constant	Description
Newton	Gravitational constant G
speed of light	Speed of light
Planck	Planck's constant
electron charge	charge of an electron
electron mass	mass of an electron
proton mass	Mass of a proton
fine-structure	fine-structure constant
Avogadro	Avogadro's Mole Fraction
Boltzmann	Boltzmann's constant
ideal gas constant	ideal gas constant
Wien	Wien displacement law constant
Stefan-Boltzmann	Stefan-Boltzman constant
pi	Pi
e	e
Euler	Euler-Mascheroni constant

## 12.1 Available Constants and Units

There are many constants and units available within FLASH3, see [Table 12.1](#) and [Table 12.2](#). Should the user wish to add additional constants or units to a particular setup, the routine `PhysicalConstants_init` should be overridden and the new constants added within the directory of the setup.

## 12.2 Applicable Runtime Parameters

There is only one runtime parameter used by the Physical Constants unit: `pc_unitsBase` selects the default system of units for returned constants. It is a three-character string set to "CGS" or "MKS"; the default is CGS.

## 12.3 Routine Descriptions

The following routines are supplied by this unit.

- `PhysicalConstants_get` Request a physical constant given by a string, and returns its real value. This routine takes optional arguments for converting units from the default. If the constant name or any of the optional unit names aren't recognized, a value of 0 is returned.
- `PhysicalConstants_init` Initializes the Physical Constants Unit by loading all constants. This routine is called by `Driver_initFlash` and must be called before the first invocation of `PhysicalConstants_get`. In general, the user does not need to invoke this call.
- `PhysicalConstants_list` Lists the available physical constants in a snappy table.
- `PhysicalConstants_listUnits` Lists all the units available for optional conversion.
- `PhysicalConstants_unitTest` Lists all physical constants and units, and tests the unit conversion routines.

Table 12.2: Available Units for Conversion of Physical Constants

Base unit	String Constant	Value in CGS units	Description
length	cm	1.0	centimeter
time	s	1.0	second
temperature	K	1.0	degree Kelvin
mass	g	1.0	gram
charge	esu	1.0	ESU charge
length	m	1.0E2	meter
length	km	1.0E5	kilometer
length	pc	3.0856775807E18	parsec
length	kpc	3.0856775807E21	kiloparsec
length	Mpc	3.0856775807E24	megaparsec
length	Gpc	3.0856775807E27	gigaparsec
length	Rsun	6.96E10	solar radius
length	AU	1.49597870662E13	astronomical unit
time	yr	3.15569252E7	year
time	Myr	3.15569252E13	megayear
time	Gyr	3.15569252E16	gigayear
mass	kg	1.0E3	kilogram
mass	Msun	1.9889225E33	solar mass
mass	amu	1.660538782E-24	atomic mass unit
charge	C	2.99792458E9	Coulomb
Cosmology-friendly units using $H_0 = 100$ km/s/Mpc:			
length	LFLY	3.0856775807E24	1 Mpc
time	TFLY	2.05759E17	$\frac{2}{3H_0}$
mass	MFLY	9.8847E45	5.23e12 Msun

### FLASH Transition

The header file `PhysicalConstants.h` must be included in the calling routine due to the optional arguments of `PhysicalConstants_get`.

## 12.4 Unit Test

The `PhysicalConstants` unit test `PhysicalConstants_unitTest` is a simple exercise of the functionality in the unit. It does not require time stepping or the grid. “Correct” usage is indicated, as is erroneous usage.



## Chapter 13

# Cpp & Python units

The **Cpp** unit collects various interfaces provided by other FLASH units that expose their API through explicit C bindings. The main purpose of these bindings is to allow for FLASH applications to interplay with a python interpreter through the C++ header library pybind11. This enables

- Controlling runtime parameters using a python script flashPar.py as a replacement for flash.par parfiles.
- Reading setup arguments from lists defined in flashPar.py
- Drop in replacements for simulation customizations that previously were done with Fortran source files

### 13.1 Cpp

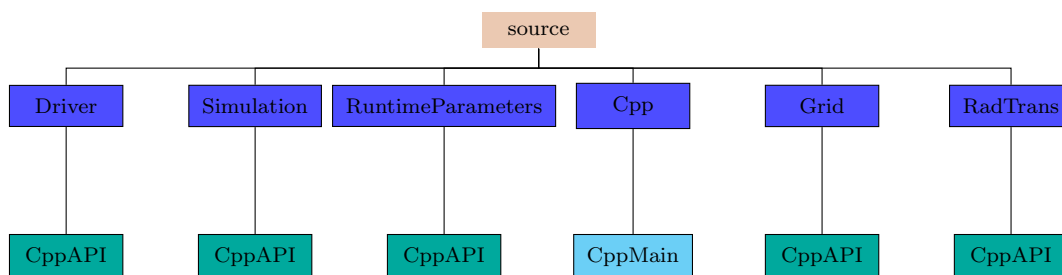


Figure 13.1: The **Cpp** unit directory tree, along with the implemented APIs.

The **Cpp** unit primarily exists to collect the various **CppAPI** implementations across the various units as depicted in Fig. 13.1, which are enumerated in **Cpp/CppMain/Config**. The **Cpp** unit also holds some subroutines that help in the fortran-C interoperability, in particular for binding strings between the two languages. This includes **Cpp\_fStr** and **Cpp\_cStrPtr**, which allows the passing of **character(len=\*)** on the fortran side and **char\*** on the C side.

Each **Unit/CppAPI** unit implements a **Unit\_cAPI.F90** interface that uses the **bind(C)** fortran feature to expose the unit's API to C, and a **UNIT\_cppAPI.hxx** header that declares those bindings as **extern "C"**. Additionally, if the unit contains C++ code that should be called from the fortran side it will implement a **Unit\_cInterface.hxx** & **Unit\_cInterface.F90** which bind in the opposite direction. This is primarily intended for implementing code in python for reading runtime parameters and user defined **Simulation** routines. The unit may be included at setup with the argument **+cppapi**.

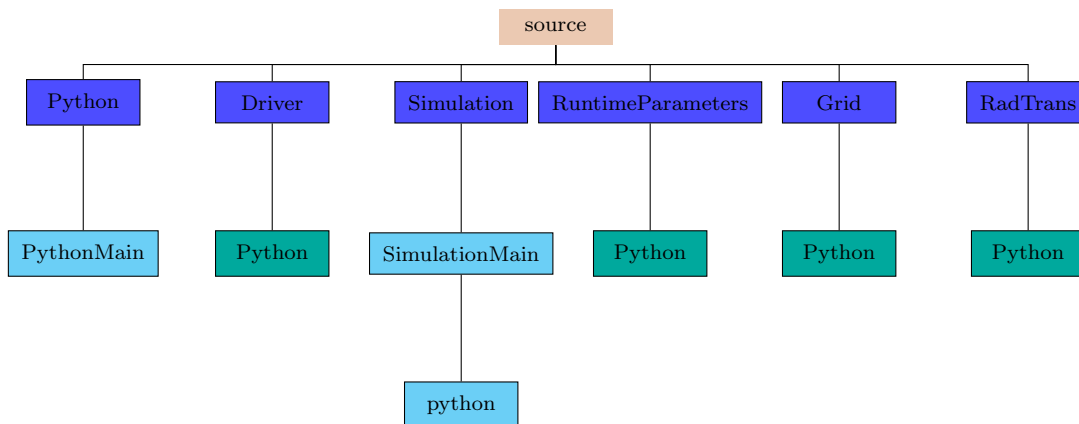


Figure 13.2: The `Python` unit directory tree, along with the implemented APIs.

## 13.2 Python

FLASH allows for binding certain unit API functionality to python through the compilation of a `pyFlash` module that is linked against the FLASH source and which the `FLASH` executable is in turn linked against using the `pybind11` library to generate the bindings. This enables the code to embed a python interpreter in the application allowing for python scripts to interact with a FLASH simulation at runtime. Currently the python bindings are only supported through the `cmake` build system.

Like the `Cpp` unit, the `Python` unit collects various other units' implementations of their own `Python` subunit. Each `Python` subunit contains a source file `Unit_py11.cxx` that is responsible for generating a python module for that unit. The full API as well as additional usage documentation can be found in the [pyFlash documentation](#).

### 13.2.1 Setup Generated Files

<code>FlashH.py</code>	Maps the simulation specific preprocessor macros from <code>Flash.h</code> to python variables. Array indices are decremented to be 0-indexed.
<code>constantsH.py</code>	Maps the preprocessor macros from <code>constants.h</code> to python variables.
<code>parm.py</code>	Creates <code>rp_[Int Real Log Str]</code> python variables that wrap around all the runtime parameters encountered in the simulation's <code>Config</code> files.



**Part V**

**Physics Units**



## Chapter 14

# 3T Capabilities for Simulation of HEDP Experiments

The FLASH code has been extended with numerous capabilities to allow it to simulate laser-driven High Energy Density Physics (HEDP) experiments. These experiments often require a multi-temperature treatment of the plasma where the ion temperature,  $T_{\text{ion}}$  and the electron temperature  $T_{\text{ele}}$  are not necessarily equal. Thermal radiation effects are also important in many High Energy Density (HED) plasmas. If the radiation field has a total energy density given by  $u_{\text{rad}}(\mathbf{x}, t)$  then the radiation temperature is defined as  $T_{\text{rad}} = (u_{\text{rad}}/a)^{1/4}$ . The radiation field is not in equilibrium with the plasma and thus  $T_{\text{rad}} \neq T_{\text{ele}} \neq T_{\text{ion}}$ . We refer to this treatment, where these three temperatures are not necessarily equal, as a 3T treatment. This chapter is intended to describe the basic theory behind FLASH's 3T implementation to direct users to other parts of the manual and simulations which provide further details on how to use these new capabilities in FLASH.

The term “3T” is not meant to imply in any way that a gray treatment of the radiation field is being assumed. The radiation temperature is only used to represent the total energy density, which is integrated over all photon frequencies. The radiation temperature never directly enters the calculation. Thus, 3T refers to the fact that FLASH is being run in a mode where 3 independent components (ions, electrons, radiation) are being modeled. The radiation field is usually treated in a frequency dependent way through multigroup radiation diffusion as described below and in [Chapter 25](#).

The equations which FLASH solves to describe the evolution of an unmagnetized 3T plasma are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \quad (14.1a)$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P_{\text{tot}} = 0, \quad (14.1b)$$

$$\frac{\partial}{\partial t}(\rho E_{\text{tot}}) + \nabla \cdot [(\rho E_{\text{tot}} + P_{\text{tot}}) \mathbf{v}] = Q_{\text{las}} - \nabla \cdot \mathbf{q}, \quad (14.1c)$$

where:

- $\rho$  is the total mass density
- $\mathbf{v}$  is the average fluid velocity
- $P_{\text{tot}}$  is the total pressure defined as the sum over the ion, electron, and radiation pressures:

$$P_{\text{tot}} = P_{\text{ion}} + P_{\text{ele}} + P_{\text{rad}} \quad (14.2)$$

- $E_{\text{tot}}$  is the total specific energy which includes the specific internal energies of the electrons, ions, and radiation field along with the specific kinetic energy. Thus:

$$E_{\text{tot}} = e_{\text{ion}} + e_{\text{ele}} + e_{\text{rad}} + \frac{1}{2} \mathbf{v} \cdot \mathbf{v} \quad (14.3)$$

- $\mathbf{q}$  is the total heat flux which is assumed to have a radiation and electron conductivity component:

$$\mathbf{q} = \mathbf{q}_{\text{ele}} + \mathbf{q}_{\text{rad}} \quad (14.4)$$

- $Q_{\text{las}}$  represents the energy source due to laser heating

Since the plasma is not assumed to have a single temperature, additional equations must be evolved to describe the change in specific internal energies of the ions, electrons, and radiation field. For the electrons and ions these equations are:

$$\frac{\partial}{\partial t}(\rho e_{\text{ion}}) + \nabla \cdot (\rho e_{\text{ion}} \mathbf{v}) + P_{\text{ion}} \nabla \cdot \mathbf{v} = \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}), \quad (14.5a)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{ele}}) + \nabla \cdot (\rho e_{\text{ele}} \mathbf{v}) + P_{\text{ele}} \nabla \cdot \mathbf{v} = \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}) - \nabla \cdot \mathbf{q}_{\text{ele}} + Q_{\text{abs}} - Q_{\text{emis}} + Q_{\text{las}}, \quad (14.5b)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{rad}}) + \nabla \cdot (\rho e_{\text{rad}} \mathbf{v}) + P_{\text{rad}} \nabla \cdot \mathbf{v} = \nabla \cdot \mathbf{q}_{\text{rad}} - Q_{\text{abs}} + Q_{\text{emis}}, \quad (14.5c)$$

where:

- $c_{v,\text{ele}}$  is the electron specific heat
- $\tau_{ei}$  is the ion/electron equilibration time
- $Q_{\text{abs}}$  represents the increase in electron internal energy due to the total absorption of radiation
- $Q_{\text{emis}}$  represents the decrease in electron internal energy due to the total emission of radiation

The 3T equation of state in FLASH connects the internal energies, temperatures, and pressures of the components. Many different equations of state options exist in FLASH. These are described in [Section 17.4](#). A full-physics HEDP simulation using FLASH will solve equations (14.1) and (14.5) using the 3T equation of state. These equations are somewhat redundant since (14.1c) can be written as a sum of the other equations. These equations are also not yet complete, since it has not been described how many of the terms above are defined and computed in FLASH. The remainder of this chapter will describe this and direct readers to the appropriate sections of the manual where examples and further information can be found.

A series of operator splits is used to solve (14.1) and (14.5) in FLASH. First, all of the terms on the left hand sides of these equations are split off and solved in various code units. The remaining equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \quad (14.6a)$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P_{\text{tot}} = 0, \quad (14.6b)$$

$$\frac{\partial}{\partial t}(\rho E_{\text{tot}}) + \nabla \cdot [(\rho E + P_{\text{tot}}) \mathbf{v}] = 0, \quad (14.6c)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{ion}}) + \nabla \cdot (\rho e_{\text{ion}} \mathbf{v}) + P_{\text{ion}} \nabla \cdot \mathbf{v} = 0, \quad (14.6d)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{ele}}) + \nabla \cdot (\rho e_{\text{ele}} \mathbf{v}) + P_{\text{ele}} \nabla \cdot \mathbf{v} = 0, \quad (14.6e)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{rad}}) + \nabla \cdot (\rho e_{\text{rad}} \mathbf{v}) + P_{\text{rad}} \nabla \cdot \mathbf{v} = 0, \quad (14.6f)$$

describe the advection of conserved quantities and the effect of work. (14.6) is solved by the **Hydro** unit. [Chapter 15](#) describes the hydrodynamics solvers in a general way where only equations for conservation of total mass, momentum, and energy are considered. The extension of the FLASH hydrodynamics solvers to 3T is described in [Section 15.1.5](#). Note that these equations are not in a conservative form because of the presence of the work terms in (14.6d), (14.6e), and (14.6f). These work terms are divergent at shocks and cannot be directly evaluated. Two techniques are described in [Section 15.1.5](#) for coping with this issue.

The electron, ion, and radiation internal energy equations in the absence of the hydrodynamic terms are shown in (14.7). The density is updated in the hydrodynamic update so for the remaining equations the density is assumed to be constant, and we remove the density from the time derivatives.

$$\rho \frac{\partial e_{\text{ion}}}{\partial t} = \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}), \quad (14.7a)$$

$$\rho \frac{\partial e_{\text{ele}}}{\partial t} = \rho \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}) - \nabla \cdot \mathbf{q}_{\text{ele}} + Q_{\text{abs}} - Q_{\text{emis}} + Q_{\text{las}}, \quad (14.7b)$$

$$\rho \frac{\partial e_{\text{rad}}}{\partial t} = \nabla \cdot \mathbf{q}_{\text{rad}} - Q_{\text{abs}} + Q_{\text{emis}}. \quad (14.7c)$$

The first term on the right hand side of (14.7a) and (14.7b) describes the exchange of internal energy between ions and electrons through collisions. This term will force the ion and electron temperatures to equilibrate over time. The **Heatexchange** unit, described in [Section 18.5.1](#), solves for this part of (14.7). Specifically, it updates the ion and electron temperatures according to:

$$\frac{\partial e_{\text{ion}}}{\partial t} = \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}), \quad (14.8a)$$

$$\frac{\partial e_{\text{ele}}}{\partial t} = \frac{c_{v,\text{ele}}}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}). \quad (14.8b)$$

The electron specific heat,  $c_{v,\text{ele}}$  is computed through calls to the equation of state. The ion/electron equilibration time is computed in the **Heatexchange** unit.

The second term on the right hand side of (14.7b) represents the transport of energy through electron thermal conduction. Thus the heat flux is defined as:

$$\mathbf{q}_{\text{ele}} = -K_{\text{ele}} \nabla T_{\text{ele}}, \quad (14.9)$$

where  $K_{\text{ele}}$  is the electron thermal conductivity and is computed in the **Conductivity** unit (see [Section 23.1](#)). The **Diffuse** unit, described in [Chapter 19](#), is responsible for including the effect of conduction in FLASH simulations. Again, using operator splitting, the **Diffuse** unit solves the following equation over a time step:

$$\rho \frac{\partial e_{\text{ele}}}{\partial t} = \nabla \cdot K_{\text{ele}} \nabla T_{\text{ele}}. \quad (14.10)$$

This equation can be solved implicitly over the time step to avoid time-step constraints. The electron conductivity is evaluated using a flux limiter to give more a physically realistic heat flux in regions where the electron temperature gradient is very large.

The remaining terms describe radiation transport. FLASH incorporates radiation effects using multigroup diffusion (MGD) theory. The total radiation flux, emission, and absorption terms which appear in (14.7) contain contributions from each energy group. For group  $g$ , where  $1 \leq g \leq N_g$ , the total quantities can be written as summations over each group:

$$Q_{\text{abs}} = \sum_{g=1}^{N_g} Q_{\text{ele},g}, \quad Q_{\text{emis}} = \sum_{g=1}^{N_g} Q_{\text{emis},g}, \quad \mathbf{q}_{\text{rad}} = \sum_{g=1}^{N_g} \mathbf{q}_g. \quad (14.11)$$

The change in the radiation energy density for each group,  $u_g$ , is described by:

$$\frac{\partial u_g}{\partial t} + \nabla \cdot (u_g \mathbf{v}) + \left( \frac{u_g}{e_{\text{rad}} \rho} \right) P_{\text{rad}} \nabla \cdot \mathbf{v} = -\nabla \cdot \mathbf{q}_g + Q_{\text{emis},g} - Q_{\text{abs},g} \quad (14.12)$$

The total specific radiation energy is related to  $u_g$  through:

$$\rho e_{\text{rad}} = \sum_{g=1}^{N_g} u_g \quad (14.13)$$

The **RadTrans** unit is responsible for solving the radiation diffusion equations for each energy group. The **RadTrans** unit solves these diffusion equations implicitly by using the **Diffuse** unit. While the work term for the total radiation energy is computed in the **Hydro** unit, the distribution of that work amongst each energy group is performed in the **RadTrans** unit. [Chapter 25](#) describes in detail how the multigroup radiation diffusion package in FLASH functions. The group radiation flux, emission, and absorption terms are all defined in that chapter. These terms are functions of the material opacity which is computed by the **Opacity** unit and is described in [Section 23.5](#).

The only remaining term in (14.7) is  $Q_{\text{las}}$  which represents the deposition of energy by lasers into the electrons. The **Laser** implementation in the **EnergyDeposition** unit is responsible for computing  $Q_{\text{las}}$ . The geometrics optics approximation to laser energy deposition is used in FLASH. [Section 18.4](#) describes the theory and usage of the laser ray-tracing model in FLASH in detail.

As has been described above, the HEDP capabilities in FLASH are divided amongst many units including:

- **Hydro**: Responsible for the 3T hydrodynamic update
- **Eos**: Computes 3T equation of state
- **Heatexchange**: Implements ion/electron equilibration
- **Diffuse**: Responsible for implementing implicit diffusion solvers and computes effect of electron conduction
- **RadTrans**: Implements multigroup radiation diffusion
- **Opacity**: Computes opacities for radiation diffusion
- **Conductivity**: Computes electron thermal conductivities
- **EnergyDeposition**: Computes the laser energy deposition

Several simulations are included with FLASH which demonstrate the usage of the HEDP capabilities and, taken together, exercise all of the units listed above. These simulations are described briefly below. [Chapter 35](#) describes all of the simulations in detail. Below, the relevant simulations listed with brief descriptions.

- **MGDInfinite** simulation, described in [Section 35.6.1](#): Simple 0D test of the exchange of energy between electrons, ions, and the radiation field
- **MGDStep** simulation, described in [Section 35.6.2](#): Simple 1D test of electron conduction, ion/electron equilibration, and MGD
- **ShafranovShock** simulation, described in [Section 35.8.1](#): Simple 1D verification test of the structure of a shock in a radiationless plasma with  $T_{\text{ele}} \neq T_{\text{ion}}$ .
- **GrayDiffRadShock** simulation, described in [Section 35.8.2](#): Simple 1D verification test of the structure of a radiating shock
- **ReinickeMeyer** simulation, described in [Section 35.8.3](#): Verification test of a spherical blast wave with thermal conduction
- **LaserSlab** simulation, described in [Section 35.7.5](#): Full physics 2D simulation which includes 3T hydrodynamics, tabulated EOS and opacity, MGD, electron conduction, and laser ray-tracing. This simulation is meant to demonstrate how to set up a complex simulation of an HEDP experiment

## Chapter 15

# Hydrodynamics Units

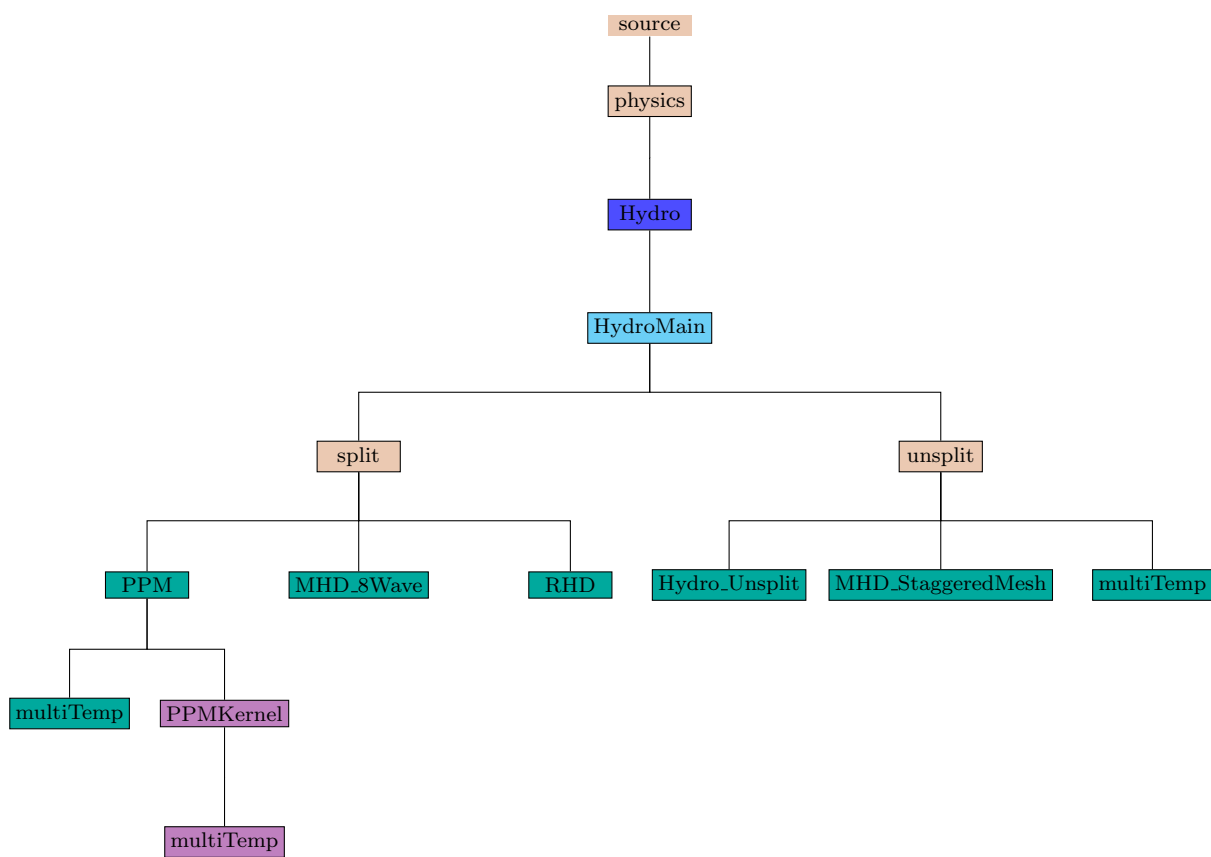


Figure 15.1: The `Hydro` unit directory tree.

The `Hydro` unit solves Euler’s equations for compressible gas dynamics in one, two, or three spatial dimensions. We first describe the basic functionality; see implementation sections below for various extensions.

The Euler equations can be written in conservative form as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \quad (15.1)$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P = \rho \mathbf{g} \quad (15.2)$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} , \quad (15.3)$$

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,

$$E = \epsilon + \frac{1}{2} |\mathbf{v}|^2 , \quad (15.4)$$

$\mathbf{g}$  is the acceleration due to gravity, and  $t$  is the time coordinate. The pressure is obtained from the energy and density using the equation of state. For the case of an ideal gas equation of state, the pressure is given by

$$P = (\gamma - 1) \rho \epsilon , \quad (15.5)$$

where  $\gamma$  is the ratio of specific heats. More general equations of state are discussed in [Section 17.2](#) and [Section 17.3](#).

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

$$\epsilon = E - \frac{1}{2} |\mathbf{v}|^2 \quad (15.6)$$

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

$$\frac{\partial \rho \epsilon}{\partial t} + \nabla \cdot [(\rho \epsilon + P) \mathbf{v}] - \mathbf{v} \cdot \nabla P = 0 . \quad (15.7)$$

If the internal energy is a small fraction of the kinetic energy (determined via the runtime parameter `eintSwitch`), then the total energy is recomputed using the internal energy from (15.7) and the velocities from the momentum equation. Numerical experiments using the PPM solver included with FLASH showed that using (15.7) 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

$$\frac{\partial \rho X_\ell}{\partial t} + \nabla \cdot (\rho X_\ell \mathbf{v}) = 0 , \quad (15.8)$$

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 `irenorm` equal to 1. Otherwise, FLASH will only restrict the abundances to fall between `smallx` and 1. The quantity  $\rho X_\ell$  represents the partial density of the  $\ell$ th fluid. The code does not explicitly track interfaces between the fluids, so a small amount of numerical mixing can be expected during the course of a calculation.

The `hydro` unit has a capability to advect mass scalars. Mass scalars are field variables advected with density, similar to species mass fractions,

$$\frac{\partial \rho \phi_\ell}{\partial t} + \nabla \cdot (\rho \phi_\ell \mathbf{v}) = 0 , \quad (15.9)$$

where  $\phi_\ell$  is the  $\ell$ th mass scalar. Note that mass scalars are optional variables; to include them specify the name of each mass scalar in a `Config` file using the `MASS_SCALAR` keyword. Mass scalars are not renormalized in order to sum to 1, except when they are declared to be part of a renormalization group. See [Section 5.5.1](#) for more details.



Table 15.1: Runtime parameters used with the hydrodynamics (**Hydro**) unit.

Variable	Type	Default	Description
<b>eintSwitch</b>	real	0	If $\epsilon < \text{eintSwitch} \cdot \frac{1}{2} \mathbf{v} ^2$ , use the internal energy equation to update the pressure
<b>irenorm</b>	integer	0	If equal to one, renormalize multifluid abundances following a hydro update; else restrict their values to lie between <b>smallx</b> and 1.
<b>cfl</b>	real	0.8	Courant-Friedrichs-Lewy (CFL) factor; must be less than 1 for stability in explicit schemes

Table 15.2: Solution variables used with the hydrodynamics (**Hydro**) unit.

Variable	Type	Description
<b>dens</b>	PER_VOLUME	density
<b>velx</b>	PER_MASS	$x$ -component of velocity
<b>vely</b>	PER_MASS	$y$ -component of velocity
<b>velz</b>	PER_MASS	$z$ -component of velocity
<b>pres</b>	GENERIC	pressure
<b>ener</b>	PER_MASS	specific total energy ( $T + U$ )
<b>temp</b>	GENERIC	temperature

## 15.1 Gas hydrodynamics

### 15.1.1 Usage

The two gas hydrodynamic solvers supplied in the release of FLASH4 are organized into two different operator splitting methods: directionally split and unsplit. The directionally split piecewise-parabolic method (PPM) makes use of second-order Strang time splitting, and the new directionally unsplit solver is based on Monotone Upstream-centered Scheme for Conservation Laws (MUSCL) Hancock type second-order scheme.

The algorithms are described in [Section 15.1.2](#) and [Section 15.1.3](#) and implemented in the directory tree under `physics/Hydro/HydroMain/split/PPM` and `physics/Hydro/HydroMain/unsplit/Hydro_Unsplit`. Extensions for multitemperature applications are described in [Section 15.1.5](#).

Current and future implementations of **Hydro** use the runtime parameters and solution variables described in [Table 15.1](#) and [Table 15.2](#). Additional runtime parameters used either solely by the PPM method or the unsplit hydro solver are described in [HydroMain](#).

### 15.1.2 The piecewise-parabolic method (PPM)

FLASH includes a directionally split piecewise-parabolic method (PPM) solver descended from the **PROMETHEUS** code (Fryxell, Müller, and Arnett 1989). The basic PPM algorithm is described in detail in Woodward and Colella (1984) and Colella and Woodward (1984). It is a higher-order version of the method developed by Godunov (1959). FLASH implements the Direct Eulerian version of PPM.

Godunov’s method uses a finite-volume spatial discretization of the Euler equations together with an explicit forward time difference. Time-advanced fluxes at cell boundaries are computed using the numerical solution to Riemann’s shock tube problem at each boundary. Initial conditions for each Riemann problem are determined by assuming the non-advanced solution to be piecewise-constant in each cell. Using the Riemann solution has the effect of introducing explicit nonlinearity into the difference equations and permits the calculation of sharp shock fronts and contact discontinuities without introducing significant nonphysical oscillations into the flow. Since the value of each variable in each cell is assumed to be constant, Godunov’s method is limited to first-order accuracy in both space and time.

PPM improves on Godunov’s method by representing the flow variables with piecewise-parabolic functions. It also uses a monotonicity constraint rather than artificial viscosity to control oscillations near discontinuities, a feature shared with the MUSCL scheme of van Leer (1979). Although these choices could lead to a method which is accurate to third order, PPM is formally accurate only to second order in both space and time, as a fully third-order scheme proved not to be cost-effective. Nevertheless, PPM is considerably more accurate and efficient than most formally second-order algorithms.

PPM is particularly well-suited to flows involving discontinuities, such as shocks and contact discontinuities. The method also performs extremely well for smooth flows, although other schemes which do not perform the extra work necessary for the treatment of discontinuities might be more efficient in these cases. The high resolution and accuracy of PPM are obtained by the explicit nonlinearity of the scheme and through the use of intelligent dissipation algorithms, such as monotonicity enforcement and interpolant flattening. These algorithms are described in detail by Colella and Woodward (1984).

A complete description of PPM is beyond the scope of this guide. However, for comparison with other codes, we note that the implementation of PPM in FLASH uses the Direct Eulerian formulation of PPM and the technique for allowing non-ideal equations of state described by Colella and Glaz (1985). For multidimensional problems, FLASH uses second-order operator splitting (Strang 1968). We note below the extensions to PPM that we have implemented.

The PPM algorithm includes a steepening mechanism to keep contact discontinuities from spreading over too many cells. Its use requires some care, since under certain circumstances, it can produce incorrect results. For example, it is possible for the code to interpret a very steep (but smooth) density gradient as a contact discontinuity. When this happens, the gradient is usually turned into a series of contact discontinuities, producing a stair step appearance in one-dimensional flows or a series of parallel contact discontinuities in multi-dimensional flows. Under-resolving the flow in the vicinity of a steep gradient is a common cause of this problem. The directional splitting used in our implementation of PPM can also aggravate the situation. The contact steepening can be disabled at runtime by setting `use_steepening = .false..`

The version of PPM in the FLASH code has an option to more closely couple the hydrodynamic solver with a gravitational source term. This can noticeably reduce spurious velocities caused by the operator splitting of the gravitational acceleration from the hydrodynamics. In our ‘modified states’ version of PPM, when calculating the left and right states for input to the Riemann solver, we locally subtract off from the pressure field the pressure that is locally supporting the atmosphere against gravity; this pressure is unavailable for generating waves. This can be enabled by setting `ppm_modifystates = .true..`

The interpolation/monotonization procedure used in PPM is very nonlinear and can act differently on the different mass fractions carried by the code. This can lead to updated abundances that violate the constraint that the mass fractions sum to unity. Plewa and Müller (1999) (henceforth CMA) describe extensions to PPM that help prevent overshoots in the mass fractions as a result of the PPM advection. We implement two of the modifications they describe, the renormalization of the average mass fraction state as returned from the Riemann solvers (CMA eq. 13), and the (optional) additional flattening of the mass fractions to reduce overshoots (CMA eq. 14-16). The latter procedure is off by default and can be enabled by setting `use_cma_flattening = .true..`

Finally, there is an odd-even instability that can occur with shocks that are aligned with the grid. This was first pointed out by Quirk (1997), who tested several different Riemann solvers on a problem designed to demonstrate this instability. The solution he proposed is to use a hybrid Riemann solver, using the regular solver in most regions but switching to an HLLE solver inside shocks. In the context of PPM, such a hybrid implementation was first used for simulations of Type II supernovae. We have implemented such a procedure, which can be enabled by setting `hybrid_riemann = .true..`

### 15.1.3 The unsplit hydro solver

A directionally unsplit pure hydrodynamic solver (unsplit hydro) is an alternate gas dynamics solver to the split PPM scheme. The method basically adopts a predictor-corrector type formulation (zone-edge data-extrapolated method) that provides second-order solution accuracy for smooth flows and first-order accuracy for shock flows in both space and time. Recently, the order of spatial accuracy in data reconstruction for the normal direction has been extended to implement the 3rd order PPM and 5th order Weighted ENO (WENO) methods. This unsplit hydro solver can be considered as a reduced version of the Unsplit Staggered Mesh

Table 15.3: Additional runtime parameters for *Interpolation Schemes* in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro\_Unsplit)

Variable	Type	Default	Description
<code>order</code>	integer	2	Order of method in data reconstruction: 1st order Godunov (FOG), 2nd order MUSCL-Hancock (MH), 3rd order PPM, 5th order WENO.
<code>transOrder</code>	integer	1	Interpolation order of accuracy of taking upwind biased transverse flux derivatives in the unsplit data reconstruction: 1st, 2nd, 3rd. The choice of using <code>transOrder=4</code> adopts a slope limiter between the 1st and 3rd order accurate methods to minimize oscillations in upwinding at discontinuities.
<code>slopeLimiter</code>	string	“vanLeer”	Slope limiter: “MINMOD”, “MC”, “VANLEER”, “HYBRID”, “LIMITED”
<code>LimitedSlopeBeta</code>	real	1.0	Slope parameter specific for the “LIMITED” slope by Toro
<code>charLimiting</code>	logical	.true.	Enable/disable limiting on characteristic variables (.false. will use limiting on primitive variables)
<code>use_steepening</code>	logical	.false.	Enable/disable contact discontinuity steepening for PPM and WENO
<code>use_flattening</code>	logical	.false.	Enable/disable flattening (or reducing) numerical oscillations for MH, PPM, and WENO
<code>use_avisc</code>	logical	.false.	Enable/disable artificial viscosity for FOG, MH, PPM, and WENO
<code>cvisc</code>	real	0.1	Artificial viscosity coefficient
<code>use_upwindTVD</code>	logical	.false.	Enable/disable upwinded TVD slope limiter PPM. NOTE: This requires NGUARD=6
<code>use_hybridOrder</code>	logical	.false.	Enable an adaptively varying reconstruction order scheme reducing its order from a high-order to first-order depending on monotonicity constraints
<code>use_gravHalfUpdate</code>	logical	.false.	On/off gravitational acceleration source terms at the half time Riemann state update
<code>use_3dFullCTU</code>	logical	.true.	Enable a full CTU (e.g., similar to the standard 12-Riemann solve) algorithm that provides full CFL stability in 3D. If .false., then the theoretical CFL bound for 3D becomes less than 0.5 based on the linear Fourier analysis.

(USM) MHD solver (see details in [Section 15.3.3](#)) that has been available in previous FLASH3 releases.

The unsplit hydro implementation can solve 1D, 2D and 3D problems with added capabilities of exploring various numerical implementations: different types of Riemann solvers; slope limiters; first, second, third and fifth reconstruction methods; a strong shock/rarefaction detection algorithm as well as two different entropy fix routines for Roe’s linearized Riemann solver.

One of the notable features of the unsplit hydro scheme is that it particularly improves the preservation of flow symmetries as compared to the splitting formulation. Also, the scheme used in this unsplit algorithm can take a wide range of CFL stability limits (e.g.,  $CFL < 1$ ) for all three dimensions, which is based on using upwinded transverse flux formulations developed in the multidimensional USM MHD solver (Lee, 2006; Lee and Deane, 2009; Lee, 2013).

The above set of runtime parameters provide various types of different combinations that help in obtaining numerical accuracy, efficiency and stability. However, there are some important tips users should know before using them.

- [Extended stencil]: When NGUARD=6 is used, users should also use `nxb`, `nyb`, and `nzb` larger than

Table 15.4: Additional runtime parameters for *Riemann Solvers* in the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro\_Unsplit)

Variable	Type	Default	Description
<code>RiemannSolver</code>	string	"Roe"	Different choices for Riemann solver. "LLF (local Lax-Friedrichs)", "HLL", "HLLC", "HYBRID", "ROE", and "Marquina"
<code>shockDetect</code>	logical	.false.	On/off attempting to detect strong shocks/rarefactions (and saving flag in " <b>shok</b> " variable)
<code>shockLowerCFL</code>	logical	.false.	On/off lowering of CFL factor where strong shocks are detected, automatically sets <b>shockDetect</b> if on.
<code>EOSforRiemann</code>	logical	.false.	Enable/disable calling EOS in computing each Godunov flux
<code>entropy</code>	logical	.false.	On/off entropy fix algorithm for Roe solver
<code>entropyFixMethod</code>	string	"HARTENHYMAN"	Entropy fix method for the Roe solver. "HARTEN", "HARTENHYMAN"

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, PPM, or WENO without changing `NGUARD` back to 4.

- [transOrder]: The first order method `transOrder=1` is a default and only supported method that is stable according to the linear Fourier stability analysis. The choices for higher-order interpolations are no longer available in this release.
- [EOSforRiemann]: `EOSforRiemann = .true.` will call (expensive) EOS routines to compute consistent adiabatic indices (*i.e.*, `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.` to reduce computation time. 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 the 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.
- [Gravity coupling with Unplit Hydro Solvers]: When gravity is included in a simulation using the unsplit hydro and MHD solvers, users can choose to include gravitational source terms in the Riemann state update at  $n + 1/2$  time step (*i.e.*, `use_gravHalfUpdate=.true.`). This will provide an improved second-order accuracy with respect to coupling gravitational accelerations to hydrodynamics. It should be noted that current optimized unsplit hydro/MHD codes (*e.g.*, those selected with `+uhd`, `+usm`) do not support the runtime parameters `use_gravPotUpdate` and `use_gravConsv` of some previous FLASH versions any more.
- [Reduced CTU vs. Full CTU for 3D in the unsplit hydro (UHD) and staggered mesh (USM) solvers]: `use_3dFull1CTU` is a new switch that enhances a numerical stability for 3D simulations in the unsplit solvers using the corner transport upwind (CTU) algorithm by Colella. The unsplit solvers of FLASH are different from many other shock capturing codes, in that neither UHD nor USM solvers need intermediate Riemann solver solutions for updating transverse fluxes in multidimensional problems. This provides a computational efficiency because there is a reduced number of calls to Riemann solvers per cell per time step. The total number of required Riemann solver solutions are two for 2D and three for 3D (except for extra Riemann calls for constraint-transport (CT) update in USM). This is smaller than the usual stability requirement in many other codes which needs four for 2D and twelve for 3D in order to provide a full CFL limit (*i.e.*,  $CFL < 1$ ).

In general for 3D, there is another computationally efficient approach that only uses six Riemann solutions (aka, 6-CTU) instead of solving twelve Riemann problems (aka, 12-CTU). In this efficient 6-CTU, however, the numerical stability limit becomes  $CFL < 0.5$ .

For solving 3D problems in UHD and USM, enabling the new switch `use_3dFull1CTU=.true.` (i.e., full-CTU) will make the solution evolution scheme similar to 12-CTU while requiring to solve three Riemann problems only (again, except for the CT update in USM). On the other hand, `use_3dFull1CTU=.false.` (i.e., reduced-CTU) will be similar to the 6-CTU integration algorithm with a reduced CFL limit (i.e.,  $CFL < 0.5$ ).

### Unsplit Hydro Solver vs. Unsplit Staggered MHD Mesh Solver

One major difference between the unsplit hydro solver and the USM MHD solver is the presence of magnetic and electric fields. The associated staggered mesh configuration required for the USM MHD solver is not needed in the unsplit hydro solver, and all hydrodynamic variables are stored at cell centers.

### Stability Limits for both Unsplit Hydro Solver and Unsplit Staggered Mesh Solver

As mentioned above, the two unsplit solvers can take a wide range of CFL limits in all three dimensions (i.e.,  $CFL < 1$ ). However, in some circumstances where there are strong shocks and rarefactions, `shockLowerCFL=.true.` could be useful to gain more numerical stability by lowering the CFL accordingly (e.g., default settings provide 0.45 for 2D and 0.25 for 3D for the Donor scheme). This approach will automatically revert such reduced stability conditions to any given original condition set by users when there are no significant shocks and rarefactions detected.

### Setting up a simulation with the unsplit hydro solver

The default hydro implementation has changed from split to unsplit in FLASH4.4. One can still specify `+unsplitHydro` (or `+uhd` for short) in the setup line in order to explicitly request the unsplit hydro solver for a simulation. One needs to specify `+splitHydro` in the setup line if a split hydro solver is required instead. For instance, a setup call `./setup Sedov -2d -auto +splitHydro` will run a Sedov 2D problem using the split PPM hydro solver. Without specifying `+unsplitHydro`, the default unsplit hydro solver will be selected.

### Diffusion terms

Non-ideal terms, such as viscosity and heat conduction, can be included in the unsplit hydro solver for simulating diffusive processes. Please see related descriptions in [Section 15.3.5](#).

### Non-Cartesian Grid Support

Grid support for non-Cartesian geometries has been revised in the unsplit hydro and MHD solvers in the current release. The supported geometries are (i) 1D spherical (ii) 2D cylindrical in r-z. Please see related descriptions in [Section 8.11](#).

### 15.1.3.1 Implementation of Stationary Rigid Body in a Simulation Domain for Unsplit Hydro Solver

An approach to include a single or multiple stationary rigid body (bodies) in a simulation domain has been newly introduced in the unsplit hydro solver. Using this new feature it is possible to add any numbers of solid bodies that are of any shapes inside a computational domain, where a reflecting boundary condition is to be applied at each solid surface. Due to the nature of regular box-like grid structure in FLASH, the surface of rigid body looks like stair steps at best rather than smooth or round shapes. High refinement levels are recommended at such stair shaped interfaces around the rigid body.

In order to add a rigid body in a simulation, users first need to add a variable called `BDRY_VAR` in a simulation `Config` file. The next step is to initialize `BDRY_VAR` in `Simulation_initBlock.F90` in such a way that a positive one is assigned to cells in a rigid body (i.e., `solnData(BDRY_VAR,i,j,k)=1.0`); otherwise a negative one for all other cells (i.e., `solnData(BDRY_VAR,i,j,k)=-1.0`).

Users can allow high resolutions around the rigid body by promoting `BDRY_VAR` to be one of the refinement variables (i.e., `refine_var_1='bdry'` in `flash.par`).

The implementation automatically adapts `order` (a spatial reconstruction order; see [Table 15.3](#)) in fluid cells that are near the rigid body, reducing any `order` ( $> 1$ ) to `order=1` at those fluid cells adjacent to the body. This prohibits any high order (higher than 1) interpolation algorithms from reaching the rigid body data which should not be used when reconstructing high order Riemann states in the adjacent fluid cells.

For this reason there is one stability issue during simulations when `order` in the fluid cells becomes 1 and hence the local reconstruction scheme becomes a first order Godunov method. For these cells, the multidimensional local data reconstruction-evolution integration scheme reduces to a donor cell method (otherwise globally the corner-transport-upwind method by Colella) which requires a reduced CFL limit (i.e.,  $CFL < 1/2$  for 2D;  $CFL < 1/3$  for 3D). In FLASH4.3, a reduced CFL factor is automatically used in such cases; the theoretical reduced CFL limit of  $1/NDIM$  is further adjusted by `hy_cflFallbackFactor`.

Two example simulations can be found in [Section 35.1.12.1](#) and [Section 35.1.12.2](#).

## 15.1.4 The Volume-of-Fluid Solver

A volume-of-fluid (VOF) solver is an optional capability to the unsplit hydrodynamic solver for maintaining sharp material interfaces. The method adopts the unsplit VOF advection algorithm (Rider & Kothe 1998) for compressible gasses (Miller & Puckett 1996). We assume only two vof fluid components (or multifluid components) that share a common velocity field. In mixed cells the two components can have different densities, temperatures and internal energies, but are assumed to be in pressure equilibrium. This is in contrast to the `MultiSpecies` unit where the every species shares a common temperature and each species contributes a partial pressure given by its mass fraction.

In the VOF framework we don't track the material interface explicitly. Instead we evolve the volume fraction of constituent fluid components in each computational cell. Interfaces are modelled as piecewise linear in each cell and are reconstructed from the volume fractions in surrounding cells (Youngs 1984, Rider & Kothe 1998). The evolution of the volume fraction for a compressible gas with components labelled by  $\alpha$  is

$$\frac{\partial f^\alpha}{\partial t} + \nabla \cdot (\mathbf{u} f^\alpha) = \frac{f^\alpha \gamma}{\gamma^\alpha} \nabla \cdot \mathbf{u}. \quad (15.10)$$

$\gamma$  and  $\gamma^\alpha$  are the equivalent single fluid and multifluid component adiabatic indices as described in [Section 17.5](#).

Unlike in the rest of the hydrodynamics unit where PDEs are solved algebraically, Eq. 15.10 is solved geometrically. First the unsplit hydrodynamic solver operates as previously described to update the hydro variables for the equivalent single fluid as parameterized in [Section 17.5](#). The mass flux at each face will imply an advected volume through that face in the computational cell, which when cut with the reconstructed piecewise linear material interface partitions the volume flux to the VOF fluid components. The advective update to the volume fraction is then

$$f^{\alpha,adv} = f^{\alpha,n} - \frac{1}{V_{\text{cell}}} \sum_{\{i,j,k\}} \left( V_{i+1/2}^\alpha - V_{i-1/2}^\alpha \right). \quad (15.11)$$

$\nabla \cdot \mathbf{u}$  is inferred from the sum of advected volume fraction's difference with unity. The ratio of component volume fluxes can then be used to partition the flux of density and internal energy.

### 15.1.5 Multitemperature extension for Hydro

**Chapter 14** described the new capabilities in FLASH for modeling High Energy Density Physics (HEDP) experiments and describes the basic theory. These experiments often require a 3T treatment of the plasma as described in **Chapter 14**. Equation (14.1) shows the full set of 3T equations solved by the current version of FLASH. A series of operator splits are used to isolate the hydrodynamic terms from the various source terms. The **Hydro** unit is responsible for solving the system of equations which includes the hydrodynamic terms, shown in (14.6). These terms describe advection and work. Note this system contains a redundant equation since the total energy equation, (14.6c), can be written as a sum which includes (14.6b), (14.6d), (14.6e), and (14.6f).

A significant challenge exists in solving (14.6) since (14.6d), (14.6e), and (14.6f) contain source terms that involve velocity divergences; these are the work terms. The quantity  $\nabla \cdot \mathbf{v}$  is not defined at a shock, and thus directly evaluating this source term is not possible. Two techniques have been implemented in FLASH for solving (14.6) without evaluating the work terms directly. These will be referred to as the entropy advection approach and the RAGE-like approach. These approaches exploit the fact that the existing hydrodynamic solvers (the split and unsplit solvers) already solve the conservation equations for total mass, momentum, and energy. These equations retain the same form in the 3T case, however they are not complete and must be augmented with other equations to close the system since the total pressure is computed using an equation of state that requires knowledge of the properties of ions and electrons independently. For example, in many equations of state, we can write that  $P_{\text{ele}} = P_{\text{ele}}(\rho, e_{\text{ele}})$  and  $P_{\text{ion}} = P_{\text{ion}}(\rho, e_{\text{ion}})$ . Thus  $P = P_{\text{ele}} + P_{\text{ion}} = P(\rho, e_{\text{ele}}, e_{\text{ion}})$ . In the 1T case, the system can be closed by assuming  $T_{\text{ele}} = T_{\text{ion}}$ .

#### 15.1.5.1 The Entropy Advection Approach

The total entropy is not continuous at a shock. To conserve mass, momentum, and energy, shocks must irreversibly convert kinetic energy to internal energy. A good approximation that can be made is that the electron entropy is continuous across a shock. The entropy advection approach makes this assumption to solve for the state of the ions. The entropy advection approach has had limited testing. It has not been extended to incorporate radiation. Thus, there is no need to include (14.6f).

The entropy advection approach solves the first three equations of (14.6) for conservation of total mass, momentum, and energy. Now the system can be closed by solving either (14.6d) or (14.6e). However, this cannot be done, since those equations have terms that are divergent at shocks. The solution is to add an additional equation which states that electron entropy is advected with the fluid:

$$\frac{\partial}{\partial t}(\rho s_{\text{ele}}) + \nabla \cdot (\rho s_{\text{ele}} \mathbf{v}) = 0 \quad (15.12)$$

The electron internal energy, temperature and other properties are then computed using the EOS in a mode which accepts specific electron entropy  $s_{\text{ele}}$  in addition to specific combined internal energy  $e_{\text{tot}}$  and  $\rho$  as inputs. Thus, the solution procedure for the entropy advection scheme is:

- Solve the system of equations defined by (14.6a), (14.6b), (14.6c), and (15.12)
- Compute the total specific internal energy,  $e_{\text{tot}}$ , by subtracting the kinetic energy from the total energy:  
 $e_{\text{tot}} = E_{\text{tot}} - \mathbf{v} \cdot \mathbf{v} / 2$
- Compute the electron specific internal energy using the 3T equation of state:  $e_{\text{ele}} = \text{EOS}(\rho, s_{\text{ele}}, e_{\text{tot}})$
- Compute the ion specific internal energy from the total specific internal energy using the kinetic energy and the electron internal energy:  $e_{\text{ion}} = e_{\text{tot}} - e_{\text{ele}}$



### 15.1.5.2 The RAGE-like Approach

The RAGE-like approach is so named because it is identical to the method implemented in the radiation hydrodynamics code RAGE (Gittings, 2008). Verification tests comparing the two codes have shown nearly identical behavior. The RAGE-like is physically accurate in smooth flow, but does not distribute internal energy correctly among the ions, electrons, and radiation field at shocks. This is in contrast to the entropy advection approach which does distribute energy correctly, but has its own limitations.

In general, let  $\Delta e_s$  be the change in the internal energy at a particular location over some short time,  $\Delta t$ . The subscript  $s$  refers to either ions, electrons, radiation, or the total specific internal energy. When considering only the hydrodynamic effects,  $\Delta e_s = \Delta e_s^{\text{adv}} + \Delta e_s^{\text{work}} + \Delta e_s^{\text{shock}}$ , where:

1.  $\Delta e_s^{\text{adv}}$  refers to the change in internal energy due to the advection of internal energy with the fluid
2.  $\Delta e_s^{\text{work}}$  refers to the change in internal energy due to hydrodynamic work. This term is not well defined near shocks.
3.  $\Delta e_s^{\text{shock}}$  refers to changes in internal energy due to shock heating. Shock heating is the necessarily, irreversible conversion of kinetic energy to internal energy which occurs at shock as a consequence of conserving mass, momentum, and energy.

One physically accurate approximation is that  $\Delta e_{\text{ion}}^{\text{shock}} = \Delta e_{\text{tot}}^{\text{shock}}$ . The challenge of 3T hydrodynamics lies in dividing these components so that they can be correctly apportioned among the ions, electrons, and radiation field. The entropy advection approach avoids maintains the physically accurate result by solving an equation for electron entropy.

The RAGE-like approach does not apportion shock heating to only the ions. Rather it apportions the quantity  $\Delta e_{\text{tot}}^{\text{work}} + \Delta e_{\text{tot}}^{\text{shock}}$  among the ions, electrons, and radiation field in proportion to the partial pressures of these components. This is consistent with (14.6) in smooth flow, but it not accurate near shocks where the shock heating should only contribute to the change in the ion internal energy. Note that it is possible to isolate internal energy changes due to advection by solving as set of advection equations for the ions, electrons, and radiation field.

Thus, the solution procedure for the RAGE-like approach is:

1. Solve the system of equations defined by (14.6a), (14.6b), and (14.6c) and simultaneously solve:

$$\frac{\partial}{\partial t}(\rho e_{\text{ele}}) + \nabla \cdot (\rho e_{\text{ele}} \mathbf{v}) = 0, \quad (15.13a)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{ion}}) + \nabla \cdot (\rho e_{\text{ion}} \mathbf{v}) = 0, \quad (15.13b)$$

$$\frac{\partial}{\partial t}(\rho e_{\text{rad}}) + \nabla \cdot (\rho e_{\text{rad}} \mathbf{v}) = 0, \quad (15.13c)$$

to update the internal energies by including only advection related changes.

2. Compute the change in total specific internal energy over the time step for each computational cell:  
 $\Delta e_{\text{tot}} = \Delta E_{\text{tot}} - \mathbf{v} \cdot \mathbf{v}/2$
3. Compute  $\Delta e_{\text{tot}}^{\text{work}} + \Delta e_{\text{tot}}^{\text{shock}}$  by subtracting the advected energy changes from the total change in internal energy
4. Divide  $\Delta e_{\text{tot}}^{\text{work}} + \Delta e_{\text{tot}}^{\text{shock}}$  amongst the ions, electrons, and radiation field according to the ratio of pressures

### 15.1.5.3 Use, Implications, and Limitations of Multitemperature Hydro Approaches

Each approach has its own strengths and weaknesses. The entropy advection approach is more accurate in the sense that it correctly includes all shock heating in the ion internal energy. The RAGE-like approach is less accurate near shocks for this reason in terms of predicting the correct downstream ion temperature and electron temperature. In general, immediately downstream of a shock, the RAGE-like approach will



Table 15.5: Additional solution variables used with the hydrodynamics (**Hydro**) unit extended for multitemperature. Note that “specific” variables are understood as per mass unit of the *combined* fluid.

Variable	Corresponding 1T Variable	Type	Description
<b>tion</b>	<b>temp</b>	GENERIC	ion temperature
<b>tele</b>	<b>temp</b>	GENERIC	electron temperature
<b>trad</b>	<b>temp</b>	GENERIC	radiation temperature
<b>pion</b>	<b>pres</b>	GENERIC	ion pressure
<b>pele</b>	<b>pres</b>	GENERIC	electron pressure
<b>prad</b>	<b>pres</b>	GENERIC	radiation pressure
<b>eion</b>	<b>eint/ener</b>	PER_MASS	specific ion internal energy
<b>eele</b>	<b>eint/ener</b>	PER_MASS	specific electron internal energy
<b>erad</b>	<b>eint/ener</b>	PER_MASS	specific radiation energy
<b>sele</b>		PER_MASS (mass scalar)	specific electron entropy

predict an electron temperature that is too large and an ion temperature that is too small. However, the density and velocity in the vicinity of shocks will be accurate. Furthermore, ion/electron equilibration (see [Chapter 14](#)) will quickly act to equalize the ion and electron temperatures. Therefore, the RAGE-like approach is reasonable when the ion/electron equilibration time is small as is the case in many physical scenarios. The Shafranov Shock simulation (see [Section 35.8.1](#)) compares the temperatures produced by the RAGE-like approach through these two approaches.

The entropy advection approach has some practical limitations that prevent its use in FLASH for general problems. First, many of the 3T EOS models in FLASH do not support the calculation of electron entropy. The only EOS that does is the gamma-law model which is not appropriate modeling many HEDP experiments. Furthermore, oscillations in the electron and ion temperatures have been observed when using this approach. These oscillations can lead to negative ion temperatures. For these reasons, the entropy advection approach has not yet been used for production FLASH simulations of HEDP experiments.

Users can select which 3T hydro approach to use by setting the **hy\_eosModeAfter** runtime parameter. When set to “dens\_ie\_gather” a RAGE-like approach is used. When set to “dens\_ie\_sele\_gather”, the entropy advection approach is used. Due to the limitations described above, the RAGE-like approach is currently the default option. The **+uhd3t** setup shortcut can be used to setup a simulation using the 3T extension of the unsplit hydrodynamics solver. The split solver is included by default whenever a multitemperature EOS is included in the simulation. The Shafranov Shock simulation ([Section 35.8.1](#)) and radiative shock ([Section 35.8.2](#)) simulations demonstrate the use of the split solver. The Laser Slab simulation ([Section 35.7.5](#)) demonstrates the use of the 3T unsplit hydrodynamics solver.

The multitemperature hydro extensions are implemented in several implementation directories named **multiTemp** within the normal hydrodynamic solvers. These contain source files that replace some of the source files of the same name that reside higher in the source tree, and related source and **Config** files. Together, these files add code functionality and solution variables that are appropriate for multitemperature simulations. The FLASH configuration mechanism automatically takes care of building with the right versions of source files when the **multiTemp** directories are included.

The **Hydro** unit thus adds variables that are used to describe the state of each of the three components. In the current version, these are maintained *in addition* to corresponding variables for the state of the fluid as a whole. There is thus redundancy in the state description. As a benefit of this redundancy, some code units that were written without multitemperature in mind may continue to function in a multitemperature context by referring to the usual 1T set of variables (like **temp**, **pres**, **ener**, **eint**). The set of variables may be optimized in future code revisions. [Table 15.5](#) shows some of the additional variables.

**Hydro** needs a multitemperature implementation of **Eos** in order to work for multitemperature setups. Thus when a simulation is configured to use the **multiTemp** **Hydro** code, it also needs to include one of the **Eos** implementations under **physics/Eos/EosMain/multiTemp**. The **Config** files under **multiTemp** in **Hydro** will take care of this, but the simulation may have to request a specific implementation under **physics/Eos/EosMain/multiTemp**. A simulation also needs to control which EOS modes to use during a

simulation. Thus the runtime parameters `eosMode`, `eosModeInit`, and the new `hy_eosModeAfter` need to be set appropriately.

### 15.1.6 Chombo compatible Hydro

This is a slightly modified version of the split Hydro solver (Section [Section 15.1.2](#)) which must be used when including `Chombo Grid`. It is available at `physics/Hydro/HydroMain/split/PPM/chomboCompatible` and is automatically selected by the setup script whenever an application includes `Chombo Grid`.

It is required because the `Chombo` class `LevelFluxRegister` which performs flux correction at fine-coarse boundaries has a very different interface to the subroutines in `Paramesh`. A `LevelFluxRegister` conserves fluxes at fine-coarse boundaries and then corrects the coarse solution data. In contrast, the `Paramesh` subroutine `Grid_conserveFluxes` only conserves fluxes and then FLASH updates the solution data in `hy_ppm_updateSoln`.

This implementation has not been tested extensively and should not be used in a production simulation. We have encountered worse than expected mass and total energy conservation when it has been used with `Chombo Grid`. It is not yet known whether this is a problem with `Chombo compatible Hydro` or whether our AMR refinement is too aggressive and there is fine-coarse interpolation where there are sharp shock fronts. Aggressive AMR refinement could be the problem because the availability of cell-by-cell refinement in `Chombo` is very different to our prior experience with `Paramesh` which refines entire blocks at a time.

An example of bad mass and total energy conservation happens in a Sod problem that uses the included parameter file named `test_regular_fluxes_fbs_troublesome.par`. The boundary conditions are setup to be periodic, so that mass and total energy should be conserved. Integrated mass and total energy at the end of test problem change by order  $1 \times 10^{-5}\%$ . Increasing `lrefine_max` from 2 to 4 does not lead to worse conservation. We have found that we can eliminate the conservation problems in this test problem by setting `BRMeshRefineBlockFactor = 8` and `maxBlockSize = 8`, however, such a change has not fixed every test problem configuration which has shown poor conservation. In general, we get better conservation when using mesh parameter values `maxBlockSize >= 8`, `BRMeshRefineBlockFactor >= 8`, `refRatio = 2` and `tagRadius >= 2`, and so have added warning messages in the FLASH log file when these values are outside of the given range. We expect to learn more about the conservation issue soon after the FLASH4 release.

This Hydro implementation can be manually included in an application with `Paramesh` mesh package as follows:

```
./setup Sod -auto chomboCompatibleHydro=True -parfile=test_regular_fluxes_fbs_troublesome.par
```

The logical parameter `chomboLikeUpdateSoln` controls whether or not FLASH uses the standard `hy_ppm_updateSoln` for the solution update. A value of `.true.` indicates that it will not be used and is the only acceptable value for `Chombo` based applications. A value of `.false.` means it is used and is a possibility for `Paramesh` based applications.

#### Using Chombo Compatible Hydro

The runtime parameter `eintSwitch` should be set to 0.0 when using `physics/Hydro/-HydroMain/split/PPM/chomboCompatible`.

## 15.2 Relativistic hydrodynamics (RHD)

### 15.2.1 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 15.2.2](#), 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.

### 15.2.2 Equations

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

$$\frac{\partial}{\partial t} \begin{pmatrix} D \\ \mathbf{m} \\ E \end{pmatrix} + \nabla \cdot \begin{pmatrix} D\mathbf{v} \\ \mathbf{mv} + p\mathbf{I} \\ \mathbf{m} \end{pmatrix} = 0, \quad (15.14)$$

where  $D$ ,  $\mathbf{m}$ ,  $E$ ,  $\mathbf{v}$  and  $p$  define, respectively, the fluid density, momentum density, total energy density, three-velocity and pressure of the fluid. (15.14) is written in units of  $c = 1$ , where  $c$  is the speed of light. The same convention is used throughout this section, and is also adopted in the FLASH code.

At present, only Cartesian (1, 2 and 3-D), 2-D cylindrical ( $x = r$ ,  $y = z$ ) and 1-D spherical (1-D,  $x = r$ ) geometries are supported by FLASH. Gravity is not included, although it can be easily added with minor modifications.

An equation of state (Eos) provides an additional relation between thermodynamic quantities and closes the system of conservation laws ((15.14)). The current version of FLASH supports only the ideal equation of state, for which the specific enthalpy  $h$  may be expressed as

$$h = 1 + \frac{\Gamma}{\Gamma - 1} \frac{p}{\rho} \quad (15.15)$$

where  $\Gamma$  (constant) is the specific heat ratio and  $\rho$  is the proper rest mass density. Causality ( $c_s < c$ ) is preserved for  $\Gamma < 2$ . The specific heat ratio is specified as a runtime parameter ("gamma").

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

$$D = \gamma\rho, \quad \mathbf{m} = \rho h \gamma^2 \mathbf{v}, \quad E = \rho h \gamma^2 - p, \quad (15.16)$$

where  $\gamma = (1 - v^2)^{-1/2}$  is the Lorentz factor. Notice that the total energy density includes the rest mass contribution. The inverse relation, giving  $\mathbf{V}$  in terms of  $\mathbf{U}$ , is

$$\rho = \frac{D}{\gamma}, \quad \mathbf{v} = \frac{\mathbf{m}}{E + p}, \quad p = Dh\gamma - E. \quad (15.17)$$

This inverse map is not trivial due to the non-linearity introduced by the Lorentz factor  $\gamma$ ; it can be shown, in fact, that (15.17) can be combined together to obtain the following implicit expression for  $p$ :

$$p = Dh(p, \tau(p))\gamma(p) - E. \quad (15.18)$$

(15.18) has to be solved at least once per time step in order to recover the pressure from a set of conservative variables  $\mathbf{U}$ . Notice that  $\tau = \tau(p)$  depends on the pressure  $p$  through  $\tau = \gamma(p)/D$  and that the specific enthalpy  $h$  is, in general, a function of both  $p$  and  $\tau$ ,  $h = h(p, \tau(p))$ . The conversion routines are implemented in the `rdh_conserveToPrimitive.F90` and `rdh_primitiveToConserve.F90` source files.

### 15.2.3 Relativistic Equation of State

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

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

For this specific purpose, the current RHD implementation supports `MODE_DENS_EI` (a default mode) and `MODE_DENS_PRE` only (but not `MODE_DENS_TEMP`) in making a `Eos_wrapped` call.

### 15.2.4 Additional Runtime Parameter

One additional runtime parameter used with the RHD unit is

Table 15.6: Additional parameters in the RHD unit.

Variable	Type	Default	Description
<code>rhd_reconType</code>	integer	1	Order of reconstruction scheme: 1 for piecewise linear; 2 for piecewise parabolic

## 15.3 Magnetohydrodynamics (MHD)

### 15.3.1 Description

The FLASH4 code includes two magnetohydrodynamic (MHD) units that represent two different algorithms. The first is the eight-wave model (**8Wave**) by Powell *et al.* (1999) that is already present in FLASH2. The second is a newly implemented unsplit staggered mesh algorithm (USM or **StaggeredMesh**). It should be noted that there are several major differences between the two MHD units. The first difference is how each algorithm enforces the solenoidal constraint of magnetic fields. The eight-wave model basically uses the truncation-error method, which effectively removes the effects of unphysical magnetic monopoles if they are generated during simulations. It does not, however, completely eliminate monopoles that are spurious in a strict physical law. To improve such unphysical effects in simulations, the unsplit staggered mesh algorithm uses the constrained transport method (Evans and Hawley, 1988) to enforce divergence-free constraints of magnetic fields. This method is shown to maintain magnitudes of  $\nabla \cdot \mathbf{B}$  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

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \quad (15.19)$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v} - \mathbf{B} \mathbf{B}) + \nabla p_* = \rho \mathbf{g} + \nabla \cdot \boldsymbol{\tau} \quad (15.20)$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\mathbf{v}(\rho E + p_*) - \mathbf{B}(\mathbf{v} \cdot \mathbf{B})) = \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot (\mathbf{v} \cdot \boldsymbol{\tau} + \sigma \nabla T) + \nabla \cdot (\mathbf{B} \times (\eta \nabla \times \mathbf{B})) \quad (15.21)$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v} \mathbf{B} - \mathbf{B} \mathbf{v}) = -\nabla \times (\eta \nabla \times \mathbf{B}) \quad (15.22)$$

where

$$p_* = p + \frac{B^2}{2}, \quad (15.23)$$

$$E = \frac{1}{2}v^2 + \epsilon + \frac{1}{2}\frac{B^2}{\rho}, \quad (15.24)$$

$$\tau = \mu \left( (\nabla \mathbf{v}) + (\nabla \mathbf{v})^T - \frac{2}{3}(\nabla \cdot \mathbf{v})\mathbf{I} \right) \quad (15.25)$$

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 15](#)). In addition, both the eight-wave and the unsplit staggered mesh units declare additional plasma variables and runtime parameters, which are listed in [Table 15.7](#) and [Table 15.8](#).

### 15.3.2 Usage

In the current release, the eight-wave unit serves as a default MHD solver. In order to choose the unsplit staggered mesh unit for MHD simulations, users need to include `+usm` in a setup script. The default eight-wave unit will be automatically chosen if there is no such specification included.

#### A word of caution

The eight-wave solver is only compatible with native grid interpolation in AMR simulations. This is because the solver only uses two layers of guard cells in each coordinate direction. The choice `-gridinterpolation=native` is automatically adopted if `+8wave` is specified in setup, otherwise, `-gridinterpolation=native` should be explicitly included in order to use the eight-wave solver without specifying `+8wave`. For instance, running a script `./setup magnetoHD/BrioWu -1d -auto +8wave` will properly setup the BrioWu problem for the 8Wave solver, and `./setup magnetoHD/BrioWu -1d -auto +usm` for the StaggeredMesh solver.

#### Supported configurations

Both MHD units currently support the uniform grid with `FIXEDBLOCKSIZE` and `NONFIXEDBLOCKSIZE` modes, and the adaptive grid with `PARAMESH` on Cartesian geometries, as well as 2D cylindrical (R-Z). When using AMR grids, the eight-wave unit supports both `PARAMESH 2` and `PARAMESH 4`, while only `PARAMESH 4` is supported in the unsplit staggered mesh solver because face-centered variables are only fully supported in `PARAMESH 4`.

Table 15.7: Additional solution variables used in the MHD units.

Variable	Type	Description
<b>magx</b>	PER_VOLUME	$x$ -component of magnetic field
<b>magy</b>	PER_VOLUME	$y$ -component of magnetic field
<b>magz</b>	PER_VOLUME	$z$ -component of magnetic field
<b>magp</b>	(GENERIC)	magnetic pressure
<b>divb</b>	(GENERIC)	divergence of magnetic field

Table 15.8: Additional runtime parameters used in the MHD units.

Variable	Type	Default	Description
<b>UnitSystem</b>	string	“none”	System of units in which MHD calculations are to be performed. Acceptable values are “none” “CGS” and “SI”.
<b>killdivb</b>	logical	.true.	Enable/disable divergence cleaning.
<b>flux_correct</b>	logical	.true.	Enable/disable flux correction on AMR grid.

### 15.3.3 Algorithm: The Unsplit Staggered Mesh Solver

A directionally unsplit staggered mesh algorithm (USM), which solves ideal and non-ideal MHD governing equations (15.19)  $\sim$  (15.22) in multiple dimensions, is a new MHD solver. Since FLASH4-beta, a full 3D implementation has been included as a new official release (Lee, 2013). 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; Lee, 2013):

- *Quasi-linearization:* This step replaces the nonlinear system (15.19)  $\sim$  (15.22) 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 15.9](#)) includes the `minmod` limiter as well as the compressible limiters such as `vanLeer` or `mc` limiter. Another choice, called `hybrid` limiter, can be used to provide a mixed type of limiters as described in Balsara (2004). In this choice, one uses a compressible limiter to produce a crisp representation for linearly degenerate waves (e.g., an entropy wave and left- and right-going Alfvén waves). To this end, a compressible limiter can be applied to the density and the magnetic fields variables, where these variables contribute much of the variations in such linearly degenerate waves. Other variables, the velocity field components and pressure, constitute four genuinely nonlinear wave families (*i.e.*, 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):

$$b_{x,i+1/2,j}^{n+1} = b_{x,i+1/2,j}^n - \frac{\Delta t}{\Delta y} \left\{ E_{z,i+1/2,j+1/2}^{n+1/2} - E_{z,i+1/2,j-1/2}^{n+1/2} \right\}, \quad (15.26)$$

$$b_{y,i,j+1/2}^{n+1} = b_{y,i,j+1/2}^n - \frac{\Delta t}{\Delta x} \left\{ -E_{z,i+1/2,j+1/2}^{n+1/2} + E_{z,i-1/2,j+1/2}^{n+1/2} \right\}. \quad (15.27)$$

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 15.2](#).

One of the main advantages of using the CT-type of scheme is that the cell face-centered magnetic fields  $b_{x,i+1/2,j}^{n+1}$  and  $b_{y,i,j+1/2}^{n+1}$ , which are updated via the above induction equations, satisfy the divergence-free constraint locally. The numerical divergence of the magnetic fields is defined as

$$(\nabla \cdot \mathbf{B})_{i,j}^{n+1} = \frac{b_{x,i+1/2,j}^{n+1} - b_{x,i-1/2,j}^{n+1}}{\Delta x} + \frac{b_{y,i,j+1/2}^{n+1} - b_{y,i,j-1/2}^{n+1}}{\Delta y} \quad (15.28)$$

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 15.9](#) below).

To solve the above induction equations (15.26) and (15.27) 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:

$$\begin{aligned} E_{z,i+1/2,j+1/2}^{n+1/2} &= \frac{1}{4} \left\{ -F_{B_y,i+1/2,j}^{n+1/2} - F_{B_y,i+1/2,j+1}^{n+1/2} + G_{B_x,i,j+1/2}^{n+1/2} + G_{B_x,i+1,j+1/2}^{n+1/2} \right\} \\ &= \frac{1}{4} \left\{ E_{z,i+1/2,j}^{n+1/2} + E_{z,i+1/2,j+1}^{n+1/2} + E_{z,i,j+1/2}^{n+1/2} + E_{z,i+1,j+1/2}^{n+1/2} \right\}, \end{aligned} \quad (15.29)$$

where  $F_{B_y}$  and  $G_{B_x}$  represent the  $x$  and  $y$  high-order Godunov flux components corresponding to the magnetic fields  $B_y$  and  $B_x$ , respectively (see details in Balsara and Spicer, 1999).

A high-order accurate version is also available by turning on a logical switch **E\_modification** in the unsplit staggered mesh scheme, which takes Taylor series expansions of the cell-cornered electric field  $E_{z,i+1/2,j+1/2}^{n+1/2}$  in all directions, followed by taking an arithmetic average of them (Lee, 2006; Lee and Deane, 2009).

The last step in the unsplit staggered mesh scheme is to reconstruct the cell-centered magnetic fields  $B_{x,i,j}$  and  $B_{y,i,j}$  from the divergence-free face-centered magnetic fields. The unsplit staggered mesh scheme takes arithmetic averages of the face-centered fields variables to obtain the cell-centered magnetic fields, which is sufficient for second order accuracy. After obtaining the new cell-centered magnetic fields, the total plasma energy may need to be corrected in order to preserve the positivity of the thermal temperature and pressure (Balsara and Spicer, 1999; Tóth, 2000). This energy correction is very useful especially in problems involving very low  $\beta$  plasma flows.

There are several choices available for calculating high order Godunov fluxes in the unsplit staggered mesh scheme. The default solver is Roe’s linearized approximate solver, which takes into account all seven waves that arise in the MHD equations. The Roe solver can adopt one of the two entropy fix routines (Harten, 1983; Harten and Hyman, 1983) in `ordetblrefr` to avoid unphysical states near strong rarefaction regions. As all seven waves are considered in Roe’s solver, high numerical resolutions can be achieved in most cases. However, Roe’s solver still can fail to maintain positive states near very low densities even with the entropy fix. In this case, computationally efficient and positively conservative Riemann solvers such as HLL (Einfeldt *et al.*, 1991), HLLC (S. Li, 2005), or HLLD (Miyoshi and Kusano, 2005) can be used to maintain positive states in simulations. A hybrid type of Riemann solver which combines using the Roe solver for high accuracy and HLLD for stability is also available.

The USM solver has been recently extended also for 2D and 2.5D cylindrical (R-Z) geometries, both for uniform grids and AMR. In the cylindrical implementation, we followed the guidelines of Mignone *et al.* (2007) and Skinner & Ostriker (2010). Special care was also taken to ensure a divergence free interpolation of the staggered magnetic field components, when grid movements occur in AMR. This novel prolongation scheme is based on the methods described in Balsara (2001, 2004) and Li & Li (2004). More information regarding the cylindrical implementation can be found in Tzeferacos *et al.* (2012, in print) whereas new test problems, provided with this release, are available at Sections [35.2.3](#) and [35.2.4](#). Handling of different geometries will be available in future releases.

### 15.3.3.1 Slowly moving shock handling in PPM

A new dissipative mechanism is a hybridized slope limiter for PPM that combines a new upwind biased slope limiter with a conventional TVD slope limiter (Lee, D., *Astronum Proc.* 2010). This hybrid upwind limiter reduces spurious numerical oscillations near discontinuities, and therefore can compute sharp, monotone profiles in compressible flows when using PPM, especially in Magnetohydrodynamics (MHD) slowly moving shock regions. (See more in Chapter [35.2.1](#).)



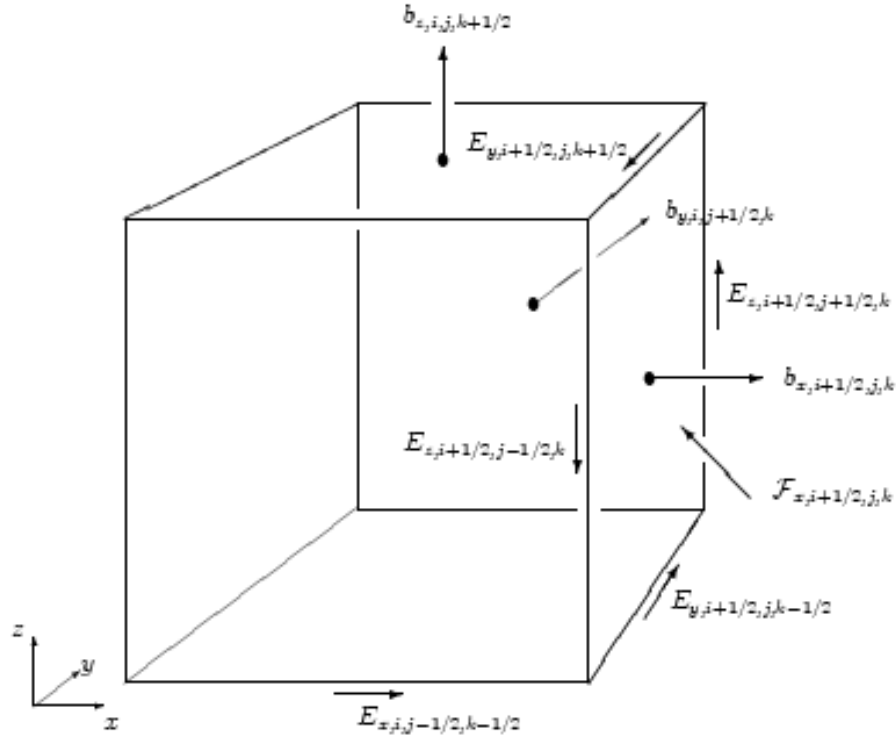


Figure 15.2: A 3D control volume on the staggered grid with the cell center at  $(i, j, k)$ . The magnetic fields are collocated at the cell face centers and the electric fields at the cell edge centers. The line integral of the electric fields  $\int \partial \mathcal{F}_n \mathbf{E} \cdot \mathbf{T} dl$  along the four edges of the face  $\mathcal{F}_{x,i+1/2,j,k}$  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 (*i.e.*, the area of  $\mathcal{F}_{x,i+1/2,j,k}$ ).

By the nature of very small numerical dissipations in PPM, unphysical oscillations in discontinuous MHD solutions can appear in a specific flow region, referred to as a *slowly moving shock* (SMS) relative to the grid. This new approach handles numerical non-oscillatory MHD solutions using PPM in SMS regions. The SMS should not be confused with so-called "slow MHD shock" which corresponds to two slow waves (*i.e.*,  $u \pm c_s$ ) in MHD, where  $c_s$  is the slow magneto-acoustic velocity.

The method first detects a local, slowly moving shock, and considers an upwind direction to compute a monotonicity-preserving slope limiter. This new approach, in addition to improving the numerical solutions in MHD to levels that reduce (or eliminate) such oscillatory behaviors while preserving sharp discontinuities in MHD, is also simple to implement. The method has been verified against the results from other high-resolution shock-capturing (HRSC) methods such as MUSCL and WENO schemes.

In order to enable the SMS treatment for PPM, users set a runtime parameter `upwindTVD=.true.` in `flash.par`. The SMS method does require to have an extended stencil, and users should specify `+supportPPMUpwind` in setup. See more in Section 15.1.3.

Table 15.9: Runtime parameters used in the unsplit staggered mesh MHD (physics/Hydro/HydroMain/unsplit/MHD\_StaggeredMesh) solver additional to those described for the unsplit hydro solver (physics/Hydro/HydroMain/unsplit/Hydro\_Unsplit).

Variable	Type	Default	Description
<code>killdivb</code>	logical	<code>.true.</code>	On/off $\nabla \cdot \mathbf{B} = 0$ handling on the staggered grid
<code>E_modification</code>	logical	<code>.true.</code>	Enable/disable high-order electric field construction
<code>E_upwind</code>	logical	<code>.false.</code>	Enable/disable an upwind update for induction equations
<code>energyFix</code>	logical	<code>.true.</code>	Enable/disable energy correction
<code>facevar2ndOrder</code>	logical	<code>.true.</code>	Turn on/off a second-order facevar update
<code>ForceHydroLimit</code>	logical	<code>.false.</code>	On/off pure Hydro mode
<code>prolMethod</code>	string	"INJECTION_PROL"	Use either direct injection method ("INJECTION_PROL") or Balsara's method ("BALSARA_PROL") in prolonging divergence-free magnetic fields stored in face-centered variables
<code>RiemannSolver</code>	string	"ROE"	"HLLD" is additionally available for MHD, "Hybrid" is also available for MHD.

### Stability limit

As described in the unsplit hydro solver unit (physics/Hydro/HydroMain/unsplitHydro\_Unsplit), the USM MHD solver can take a wide range of CFL limits in all three dimensions (*i.e.*,  $\text{CFL} < 1$ ). However, in some circumstances where there are strong shocks and rarefactions, `shockLowerCFL=.true.` could be useful to gain more numerical stability by using. It is also helpful to use (1) artificial viscosity and flattening, or (2) lower order reconstruction scheme (e.g., MH), or (2) diffusive Riemann solver such as HLL-type, or LLF solvers, or (3) a reduced CFL accordingly.

### Divergence-free prolongation of magnetic fields on AMR in the unsplit staggered mesh solver

It is of importance to preserve divergence-free evolutions of magnetic fields in MHD simulations. Moreover, some special cares are required in prolonging divergence-free magnetic fields on AMR grids. One simple straightforward way in this aspect is to prolong divergence-free fields to newly created children blocks using direct injection. This injection method therefore inherently preserves divergence-free properties on AMR block structures and works well in most cases. This method is a default in the unsplit staggered mesh solver and can also be enabled by setting a runtime parameter `prolMethod = "INJECTION_PROL"`. Another way, proposed by Balsara (2001), is also available in the unsplit staggered mesh solver and can be chosen by setting `prolMethod = "BALSARA_PROL"`. Both prolongation methods are supported in MHD's 2.5D and 3D simulations. In 2 and 2.5D cylindrical geometry however, since neither method takes into account geometrical factors, we use a modified prolongation algorithm based on Balsara (2004) and Li&Li (2004). This is the default option and is activated by choosing `prolMethod = "BALSARA_PROL"`. The need for this special refinement requires to have an MHD's own customized implementation of `Simulation_customizeProlong.F90` placed in the `source/Simulation/SimulationMain/magnetoHD/`.

### 15.3.4 Algorithm: The Eight-wave Solver

The eight-wave magnetohydrodynamic (MHD) unit is based on a finite-volume, cell-centered method that was proposed by Powell *et al.* (1999). The unit uses directional splitting to evolve the magnetohydrodynamics equations. Like the PPM and RHD units, this MHD unit makes one sweep in each spatial direction to advance physical variables from one time level to the next. In each sweep, the unit uses AMR functionality to fill in guard cells and impose boundary conditions. Then it reconstructs characteristic variables and uses these variables to compute time-averaged interface fluxes of conserved quantities. In order to enforce conservation at jumps in refinement, the unit makes flux conservation calls to AMR, which redistributes affected fluxes using the appropriate geometric area factors. Finally, the unit updates the solution and calls the EOS unit to ensure thermodynamical consistency.

After all sweeps are completed, the MHD unit enforces magnetic field divergence cleaning. In the current release only a diffusive divergence cleaning method (truncation-error method), which was the default method in FLASH2, is supported and the later release of the code will incorporate an elliptic projection cleaning method.

The ideal part of the MHD equations are solved using a high-resolution, finite-volume numerical scheme with MUSCL-type (van Leer, 1979) limited gradient reconstruction. In order to maximize the accuracy of the solver, the reconstruction procedure is applied to characteristic variables. Since this may cause certain variables, such as density and pressure, to fall outside of physically meaningful bounds, extra care is taken in the limiting step to prevent this from happening. All other variables are calculated in the unit from the interpolated characteristic variables.

In order to resolve discontinuous Riemann problems that occur at computational cell interfaces, the code by default employs a Roe-type solver derived in Powell *et al.* (1999). This solver provides full characteristic decomposition of the ideal MHD equations and is, therefore, particularly useful for plasma flow simulations that feature complex wave interaction patterns. An alternative Riemann solver of HLLE type, provided in FLASH2, has not been incorporated into FLASH4 yet.

Time integration in the MHD unit is done using a second-order, one-step method due to Hancock (Toro, 1999). For linear systems with unlimited gradient reconstruction, this method can be shown to coincide with the classic Lax-Wendroff scheme.

A difficulty particularly associated with solving the MHD equations numerically lies in the solenoidality of the magnetic field. The notorious  $\nabla \cdot \mathbf{B} = 0$  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 current release, the truncation-error cleaning methods is provided for the eight-wave MHD unit, and the constrained transport method is implemented for the unsplit staggered mesh MHD units (see [Section 15.3.3](#) for details).

In the truncation-error method, the solenoidality of the magnetic field is enforced by including several terms proportional to  $\nabla \cdot \mathbf{B}$ . 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

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v} \mathbf{B} - \mathbf{B} \mathbf{v}) = -\mathbf{v} \nabla \cdot \mathbf{B} + \eta_a \nabla \nabla \cdot \mathbf{B} , \quad (15.30)$$

with the artificial diffusion coefficient  $\eta_a$  chosen to match that of grid numerical diffusion. In the FLASH code,  $\eta_a = \frac{\lambda}{2} \left( \frac{1}{\Delta x} + \frac{1}{\Delta y} + \frac{1}{\Delta z} \right)^{-1}$ , 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 FLASH4 and will be supported in a later release.

### 15.3.5 Extended MHD

FLASH includes a complete generalized Ohm's law that incorporates all extended MHD terms of the Braginskii formulation. In other words, Hall, Biermann battery, anisotropic magnetic resistivity, Seebeck, Nernst, and cross-field terms are all included as shown in the generalized Ohm's Law below (Eq. 15.31):

$$\begin{aligned} \mathbf{E} = & -\mathbf{u} \times \mathbf{B} + \frac{\mathbf{J}}{n_e e} \times \mathbf{B} - \frac{\nabla P_e}{n_e e} \\ & + \frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{b}(\mathbf{b} \cdot \mathbf{J}) + \frac{\eta_{\perp}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{b} \times (\mathbf{J} \times \mathbf{b}) - \frac{\eta_{\wedge}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \\ & - \frac{k\beta_{\parallel}}{e} \mathbf{b}(\mathbf{b} \cdot \nabla T_e) - \frac{k\beta_{\perp}}{e} \mathbf{b} \times (\nabla T_e \times \mathbf{b}) - \frac{k\beta_{\wedge}}{e} (\mathbf{b} \times \nabla T_e) \end{aligned} \quad (15.31)$$

where  $\mathbf{E}$  is the electric field,  $\mathbf{B}$  the magnetic field,  $\mathbf{u}$  the plasma velocity,  $\mathbf{J}$  the current density,  $n_e$  the electron number density,  $e$  the electron charge,  $P_e$  the electron pressure,  $k$  is Boltzmann's constant,  $\hat{\mathbf{b}}$  the unit vector in the direction of the magnetic field,  $\omega_{pe}$  the electron plasma frequency,  $\epsilon_0$  the permittivity of vacuum,  $\tau_{ei}$  the electron-ion collision time,  $T_e$  the electron temperature, and  $\eta$  and  $\beta$  are respectively the resistivity and the thermoelectric coefficients in dimensionless form.

Each of these terms can be enabled or disabled in FLASH at run time via the `flash.par` file. For example, a typical `flash.par` file for extended MHD runtime parameters should look more or less like this:

```
# Hall effect -----
use_Hall = .true.

# Biermann battery (Source implementation)-----
use_Biermann = .true.
hy_biermannSource = .true.

# Biermann battery (Flux implementation)-----
use_Biermann = .true.

# Biermann battery (Flux implementation with thermal magnetic wave term)-----
use_Biermann = .true.
use_Biermann3T = .true.

# Magnetic Resistivity -----
useMagneticResistivity = .true.
resistivitySolver = "explicit"
resistivityForm = "fullaniso"
useCrossMagRes = .true. # the eta_wedge term
```

```
# Thermoelectric Terms -----
use_Seebeck = .true.
use_CrossField = .true.
use_Nernst = .true.
```

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:

```
# Diffusive time step calculation and implicit solver(s)
REQUIRES physics/Diffuse/DiffuseMain/Unsplit

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

# Thermoelectric
REQUIRES physics/materialProperties/Thermoelectric/ThermoelectricMain/DaviesWen
```

For Hall effect, Biermann battery, and Thermoelectric transport, there is no specific units to add, the only requirement is to require "source/physics/Hydro/HydroMain/unsplit/MHD'StaggeredMesh" (+usm) and for Biermann battery the MultiTemp implementation (+3t). Typically, for HEDP simulations, the setup will be done using "+usm3t". Also, for realistic Thermoelectric transport, a coefficients implementation should be chosen in the `Config` file as shown above. Below we detail the Hall, Biermann, Resistivity, and Thermoelectric implementations.

#### Including diffusive terms

New treatments has been made in including the non-ideal diffusive terms in the USM-MHD solver in the FLASH3.2 release and remain unchanged since then. In the previous releases before FLASH3.2, all the non-ideal diffusive terms had to be grouped together in the "resistive MHD" part of the unit by turning the flag variable `useMagneticResistivity` on, and the non-ideal terms were included all together. In the FLASH3.2 release, each individual term can be separately included by turning each corresponding logical variables in run time. The Diffusion time step is computed using the `Diffuse_computeDt.F90` routine in `Driver_computeDt.F90` to provide a more consistent way of computing a non-ideal time step.

#### 15.3.5.1 Hall effect

**15.3.5.1.1 How to use it ?** As indicated above (15.3.5), the user needs to include at least the Unsplit Staggered Mesh MHD solver (" +usm" during setup) in order to call the Hall implementation. Then the runtime parameter `use_Hall` has to be set to `= .true.`. Adding the Hall effect in the MHD model introduces two new kind of waves (see 15.3.5.1.5) that need to be taken into account for stability. It is why the `cf1` runtime parameter will generally have to be modified by the user for simulations including Hall physics. It is common for Hall simulations to use `cf1` values smaler than 0.1. The Hall effect is available in FLASH in 1D/2D/3D Cartesian geometries as well as in 2D Cylindrical geometry. We also require the user to set the runtime parameter `conserveAngField` to `=.true.` when using Hall effect in 2D cylindrical geometry.

**15.3.5.1.2 The Hall effect in the MHD model** Implementing the Hall effect in a MHD code consists to add one term in the induction equation:

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} \quad \text{where} \quad \mathbf{E} = \mathbf{E}_{\text{ind}} + \mathbf{E}_{\text{Fric}} + \mathbf{E}_{\text{Bier}} + \mathbf{E}_{\text{Hall}} \quad (15.32)$$

where each electric field is obtained from the generalized Ohm's law.  $\mathbf{E}_{\text{ind}} = -\mathbf{u} \times \mathbf{B}$  is the inductive electric field,  $\mathbf{E}_{\text{Fric}}$  is the electric field associated to the existence of friction between electrons and ions (one component of this field, and the one implemented in the FLASH code, is the ohmic electric field  $\mathbf{E}_{\text{ohm}} = \eta \mathbf{j}$ ),  $\mathbf{E}_{\text{Bier}} = -\nabla p_e / en_e$  is the Biermann electric field (discussed in the next section) and finally:

$$\mathbf{E}_{\text{Hall}} = \frac{\mathbf{j} \times \mathbf{B}}{en_e} \quad (15.33)$$

is the Hall electric field. We note that other fields exist in the generalized ohm's law (inertial...) but are not described here.

Because the electrical current  $\mathbf{j}$  is defined as  $\mathbf{j} = en_e(\mathbf{u}_i - \mathbf{u}_e)$ , the sum of the inductive and Hall terms can be written as  $\mathbf{E}_{\text{ind}} + \mathbf{E}_{\text{Hall}} = -\mathbf{u}_e \times \mathbf{B}$ . This indicates that the "frozen-in" behavior is actually valid only for the electron population. However, in many cases (see next section), it is a good approximation to consider the field to be frozen in the bulk (ion) population. Here we are interested in cases where this approximation fails and thus where the presence of the Hall term is necessary.

In FLASH, the Hall effect is implemented by updating the face-centered fluxes for magnetic field and total energy equations. By first writing the magnetic energy equation (in cgs):

$$\frac{\partial(B^2/8\pi)}{\partial t} + \nabla \cdot \left( \frac{c}{4\pi} \mathbf{E} \times \mathbf{B} \right) = -\mathbf{j} \cdot \mathbf{E} \quad (15.34)$$

we notice that the Hall electric field is not responsible for energy generation source  $\mathbf{j} \cdot \mathbf{E}_{\text{Hall}} = 0$  and only the Poynting vector ( $c\mathbf{E} \times \mathbf{B}/4\pi$ ) is affected by the presence of this additional electric field. This specificity of the Hall effect means that there is no need to modify other equations in the MHD model in order to satisfy total energy conservation (something we will have to do for Biermann). The Poynting contribution is added in FLASH in the face-centered total energy fluxes. Below we detail the flux formulation for the magnetic field equation, both in cartesian and cylindrical coordinates.

**15.3.5.1.3 Formulation for a cartesian implementation** In cartesian coordinates, the induction equation 15.32 can be written as:

$$\frac{\partial \mathbf{B}}{\partial t} + \underbrace{\frac{\partial}{\partial x} \begin{pmatrix} 0 \\ -E_z \\ E_y \end{pmatrix}}_{\text{x-flux}} + \underbrace{\frac{\partial}{\partial y} \begin{pmatrix} E_z \\ 0 \\ -E_x \end{pmatrix}}_{\text{y-flux}} + \underbrace{\frac{\partial}{\partial z} \begin{pmatrix} -E_y \\ E_x \\ 0 \end{pmatrix}}_{\text{z-flux}} = \mathbf{0} \quad (15.35)$$

or equivalently, using matrix notation:

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot \bar{\bar{\mathbf{F}}} = \mathbf{0} \quad (15.36)$$

where the flux matrix is expressed in terms of the electric field components:

$$\bar{\bar{\mathbf{F}}} = \begin{bmatrix} 0 & -E_z & E_y \\ E_z & 0 & -E_x \\ -E_y & E_x & 0 \end{bmatrix} \quad (15.37)$$

with the tensor divergence defined as  $\nabla \cdot \bar{\bar{\mathbf{F}}}|_i = \partial_j F_{ji}$ .

Expressing the Hall electric field components (from 15.33):

$$\mathbf{E}_{\text{Hall}} = \begin{pmatrix} (j_y B_z - j_z B_y)/en_e \\ (j_z B_x - j_x B_z)/en_e \\ (j_x B_y - j_y B_x)/en_e \end{pmatrix} \quad (15.38)$$

the Hall flux matrix is explicitly given by:

$$\bar{\bar{\mathbf{F}}}_{Hall} = \begin{bmatrix} 0 & -(j_x B_y - j_y B_x)/en_e & (j_z B_x - j_x B_z)/en_e \\ (j_x B_y - j_y B_x)/en_e & 0 & -(j_y B_z - j_z B_y)/en_e \\ -(j_z B_x - j_x B_z)/en_e & (j_y B_z - j_z B_y)/en_e & 0 \end{bmatrix} \quad (15.39)$$

**15.3.5.1.4 Formulation for a cylindrical implementation** In cylindrical (r,θ,z) coordinates, the induction equations 15.32 writes as:

$$\frac{\partial \mathbf{B}}{\partial t} = - \left( \begin{array}{c} \frac{1}{r} \frac{\partial E_z}{\partial \theta} - \frac{\partial E_\theta}{\partial z} \\ \frac{\partial E_r}{\partial z} - \frac{\partial E_z}{\partial r} \\ \frac{1}{r} \frac{\partial (r E_\theta)}{\partial r} - \frac{1}{r} \frac{E_r}{\partial \theta} \end{array} \right) \quad (15.40)$$

which can be written as:

$$\frac{\partial \mathbf{B}}{\partial t} + \underbrace{\frac{1}{r} \frac{\partial}{\partial r} \begin{pmatrix} 0 \\ -r E_z \\ r E_\theta \end{pmatrix}}_{\text{r-flux}} + \underbrace{\frac{1}{r} \frac{\partial}{\partial \theta} \begin{pmatrix} E_z \\ 0 \\ -E_r \end{pmatrix}}_{\text{θ-flux}} + \underbrace{\frac{\partial}{\partial z} \begin{pmatrix} -E_\theta \\ E_r \\ 0 \end{pmatrix}}_{\text{z-flux}} = - \underbrace{\frac{1}{r} \begin{pmatrix} 0 \\ E_z \\ 0 \end{pmatrix}}_{\text{source}} \quad (15.41)$$

the "source term" on the right side here (often called "geometrical" source term) comes from the fact that we can write  $\partial E_z / \partial r = 1/r \partial(r E_z) / \partial r - E_z / r$ . Eq.15.41 can also be written without including the source term and rather having everything written in terms of fluxes. It's actually what is being done currently in FLASH and that's why it requires the user to set the runtime parameter `conserveAngField` to `= .true.` for the 2D cylindrical case.

The Hall electric field in 2D axisymmetric cylindrical is explicitly given by:

$$\mathbf{E}_{Hall} = \begin{pmatrix} (j_\theta B_z - j_z B_\theta)/en_e \\ (j_z B_r - j_r B_z)/en_e \\ (j_x B_\theta - j_\theta B_x)/en_e \end{pmatrix} \quad (15.42)$$

**15.3.5.1.5 Time step limitation due to the Hall effect** As mentionned before, the addition of the Hall term in the MHD model is responsible for the possibility for the plasma to sustain two additional propagation modes. The phase velocity of these modes are computed in `Hydro_computeDt.F90` and compared with other present speeds (sound, Alfvén, fast and slow magneto-acoustic...) to determine the temporal time step `dt`. To study the possibility for the plasma to generate Hall-induced waves, we keep only the Hall term in the induction equation and express it in this form:

$$\frac{\partial \mathbf{B}}{\partial t} = \underbrace{-\frac{1}{en_e} \nabla \times (\mathbf{j} \times \mathbf{B})}_{\text{Whistler waves}} + \underbrace{\frac{1}{en_e^2} \nabla n_e \times (\mathbf{j} \times \mathbf{B})}_{\text{Hall drift waves}} \quad (15.43)$$

the first right side term is responsible for the generation of the so called "Whistler waves" whereas the second term is responsible for the generation of the "Hall drift waves".

It is easy to show that (see Huba, 2003), applying the plane wave perturbation on equation 15.43 as well as using the Maxwell-Ampere law in the non-relativistic limit ( $\mu_0 \mathbf{j} = \nabla \times \mathbf{B}$ ), the dispersion relation (in parallel propagation) for the whistler waves is given by:

$$\omega_{wh} = k_{\parallel} v_A \frac{k_{\parallel} c}{\omega_{pi}} = k_{\parallel} v_A (k_{\parallel} L_i) \quad (15.44)$$

It can be clearly seen from this relation that the Whistler waves (Hall-induced) will become faster than the Alfvén waves (inductively-induced) if  $k_{\parallel} L_i > 1$  or expressed in an other way if the spatial scale of the perturbation ( $\sim k_{\parallel}^{-1}$ ) is smaller than the ion inertial length  $L_i$ . This demonstrate why hall physics becomes really relevant for a physical problem on scales smaller than  $L_i$ .

By looking at the second right side term in equation 15.43, it can be shown that the Hall drift waves propagates in inhomogeneous plasmas and in the  $\mathbf{B} \times \nabla n_e$  direction (assumed to be the direction of the  $\mathbf{k}$  vector). A simple dispersion relation can be obtained in the limit  $kL_{n_e} \ll 1$ :

$$\omega_{hd} = kv_A \frac{c}{L_{n_e} \omega_{pi}} = kv_A \frac{L_i}{L_{n_e}} \quad (15.45)$$

with  $L_{n_e} = n_e/|\nabla n_e|$  being the spatial scale of the density gradient. Here again, the Hall-induced Hall drift waves will propagates faster than the Alfvén speed only if  $L_{n_e} < L_i$ , that is to say if the spatial scale of the density gradient is smaller than the ion inertial length.

### 15.3.5.2 Biermann battery mechanism

**15.3.5.2.1 How to use it ?** As indicated above (15.3.5), the user needs to include the Unsplit Staggered Mesh MHD solver ("usm" during setup) in order to call the Biermann implementation. We currently offer support only for the 3T implementation ("3t" during setup). Users wanting to use Biermann should thus setup their simulation with "usm3t". Then the runtime parameter `hy_useBiermann` has to be set to `= .true.`. Two different versions of the Biermann battery mechanism are currently available in FLASH: a source version which should be considered as the most robust and reliable implementation as well as a flux-based version which should be used for specific problems with user-specified boundaries conditions. To use the source version, the user needs to set the runtime parameter `hy_biermannSource` to `= .true.` whereas for the flux-based version this runtime parameter should remain set to its default value of `= .false.`. A strong difference between the two versions lies in the fact that the flux-based one can introduce a new kind of thermal magnetic wave and thus new stability considerations that have to be taken into account by the user through the runtime parameter `cfl`. The term associated with this wave can be turned on or off with the logical runtime parameter `use_Biermann3T`. The Biermann battery mechanism, for both versions, is available in FLASH in 2D Cartesian and 2D Cylindrical geometries, but only the flux-based version will work accurately in 3D Cartesian. We also require the user to set the runtime parameter `conserveAngField` to `= .true.` when using Biermann battery mechanism in 2D cylindrical geometry. Finally, we recommend to the user to always set the runtime parameter `shockDetect` to `= .true.` with the consequence of removing Biermann B-field generation in cells tagged as shock.

**15.3.5.2.2 The Biermann battery mechanism in the MHD model** Implementing the Biermann battery mechanism in a MHD code consists of adding terms in the induction equation as well as in the internal energy equations. Regarding only the Biermann term, the induction equation writes (see eq.15.32) as:

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}_{\text{Bier}} \quad (15.46)$$

with the Biermann electric field given by:

$$\mathbf{E}_{\text{Bier}} = -\frac{\nabla p_e}{en_e} \quad (15.47)$$

This mechanism describes the possibility for a plasma to self-generate magnetic fields if there exist some crossed gradients of density and temperature. It can be seen by rewriting eq.15.46 (and inserting 15.47) as:

$$\frac{\partial \mathbf{B}}{\partial t} = \frac{k_B}{en_e} \nabla T_e \times \nabla n_e \quad (15.48)$$

From this last equation it's also clear that Biermann-generated B-fields will be possible only in 2D and 3D geometries. As mentionned above, including Biermann in a MHD model should always be accompanied by the addition of one or several terms in the internal energy equations because of energy conservation considerations. This can be understood by looking at the magnetic energy equation (eq.15.34) and noting that the Biermann electric field is responsible for a source term  $-\mathbf{j} \cdot \mathbf{E}_{\text{Bier}} \neq 0$ . To maintain the conservation of total energy, one can then understand that this source terms must be compensated somewhere in another form of energy sink (where does the energy necessary to generate the B-fields comes from ?). This point is addressed in two different ways in FLASH:



- In the source version, we just aim to keep track of the energy by considering that the increase of magnetic energy due to the Biermann mechanism in a given cell is compensated directly by a decrease of electron internal energy.
- In the flux version, we follow a more physical argument by observing that the presence of the Biermann electric field in the Ohm's law is a direct consequence of keeping track of the electron pressure work on the electron population (which is, in MHD, assumed to be equilibrated by the work of an neutralizing electric field:  $E_{Bier}$ ). Therefore, it seems logical that one needs to take into account the full electron enthalpy transport in this case and this transport happens with the velocity of the electron population  $\mathbf{v}_e = \mathbf{v} - \mathbf{j}/en_e$ . In conclusion, the presence of the Biermann term in the induction equation should be accompanied by the following electron internal energy equation (without any source, collision or conduction process, for simplicity):

$$\frac{\partial \epsilon_e}{\partial t} + \nabla(\epsilon_e \mathbf{v}_e) + p_e \nabla \cdot \mathbf{v}_e = \frac{\partial \epsilon_e}{\partial t} + \nabla(\epsilon_e(\mathbf{v} - \mathbf{j}/en_e) + p_e \nabla \cdot (\mathbf{v} - \mathbf{j}/en_e) = 0 \quad (15.49)$$

It is then easy to show that with such an equation for electron internal energy and in the presence of  $E_{Bier}$  in the Ohm's law, the total energy equation (magnetic + kinetic + internal) can be written in a fully conservative form. As we will demonstrate in 15.3.5.2.5, the addition of these new terms introduces a new kind of wave: the Thermal-magnetic waves and a corresponding phase velocity that needs to be taken into account for stability.

**15.3.5.2.3 Formulation for a Cartesian implementation** Using the formalism described in 15.3.5.1.3, we write explicitly the Biermann electric field in cartesian coordinates:

$$\mathbf{E}_{Bier} = \begin{pmatrix} -\partial_x p_e / en_e \\ -\partial_y p_e / en_e \\ -\partial_z p_e / en_e \end{pmatrix} \quad (15.50)$$

and thus the Biermann flux matrix is explicitly given by:

$$\bar{\bar{\mathbf{F}}}_{Biermann} = \begin{bmatrix} 0 & \partial_z p_e / en_e & -\partial_y p_e / en_e \\ -\partial_z p_e / en_e & 0 & -\partial_x p_e / en_e \\ \partial_y p_e / en_e & \partial_x p_e / en_e & 0 \end{bmatrix} \quad (15.51)$$

**15.3.5.2.4 Formulation for a Cylindrical implementation** Assuming the electric field being composed only of the Biermann one, from 15.40 and in the case of an axisymmetric configuration ( $\partial/\partial\theta = 0$ ), eq.15.41 can be written as (here  $E_\theta = 0$ ):

$$\frac{\partial \mathbf{B}}{\partial t} + \underbrace{\frac{\partial}{\partial r} \begin{pmatrix} 0 \\ -E_z \\ 0 \end{pmatrix}}_{\text{r-flux}} + \underbrace{\frac{\partial}{\partial z} \begin{pmatrix} 0 \\ E_r \\ 0 \end{pmatrix}}_{\text{z-flux}} = \mathbf{0} \quad (15.52)$$

In this case also we require the user to set the runtime parameter `conserveAngField` to `= .true.` for the 2D cylindrical case. It can be seen that in this geometry only a  $\theta$ -magnetic field can be generated. The Biermann electric field in 2D axisymmetric cylindrical is explicitly given by:

$$\mathbf{E}_{Biermann} = - \begin{pmatrix} \partial_r p_e / en_e \\ 0 \\ \partial_z p_e / en_e \end{pmatrix} \quad (15.53)$$

**15.3.5.2.5 Time step limitation due to the Biermann battery mechanism** As mentioned previously, the flux version of Biermann (see 15.3.5.2.2) allows the propagation of a new kind of waves: the Thermal-magnetic waves (Pert, 1977). To derive the dispersion relation of these waves, suppose a cylindrical axisymmetric situation where the density depends only on the coordinate  $r$  whereas the background temperature is homogeneous. Then consider this set of equations:

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times \left( \frac{\nabla p_e}{en_e} \right) \quad (15.54)$$

$$\frac{\partial p_e}{\partial t} = \nabla \cdot \left( \frac{p_e}{en_e} \mathbf{j} \right) + (\gamma - 1)p_e \nabla \cdot \left( \frac{\mathbf{j}}{en_e} \right) \quad (15.55)$$

where we assumed  $\epsilon_e = p_e/(\gamma - 1)$ . Then we apply perturbation theory in cylindrical coordinates (with  $T_{e0} = cst$  and  $\beta_0(r) = \ln(n_{e0}(r))$ ):

$$\frac{\partial B_{1\theta}}{\partial t} = -\frac{1}{en_{e0}} \frac{\partial \beta_0}{\partial r} \frac{\partial p_{e1}}{\partial z} \quad (15.56)$$

$$\frac{\partial p_{e1}}{\partial t} = -(\gamma - 1) \frac{k_B T_{e0}}{\mu_0 e} \frac{\partial \beta_0}{\partial r} \frac{\partial B_{1\theta}}{\partial z} \quad (15.57)$$

Considering plane waves  $\propto \exp(i(\omega t - kz))$ , we get:

$$\omega B_{1\theta} = \frac{1}{en_{e0}} \frac{\partial \beta_0}{\partial r} k p_{e1} \quad (15.58)$$

$$\omega p_{e1} = (\gamma - 1) \frac{k_B T_{e0}}{\mu_0 e} \frac{\partial \beta_0}{\partial r} k B_{1\theta} \quad (15.59)$$

and we finally get the dispersion relation for the Thermal-magnetic waves:

$$\omega^2 = (\gamma - 1) \frac{k_B T_{e0}}{\mu_0 e^2 n_{e0}} \left( \frac{\partial \beta_0}{\partial r} \right)^2 k^2 \quad (15.60)$$

from which we can identify the phase velocity  $v_{TM}$  of Thermal-magnetic waves:

$$v_{TM} = \sqrt{\frac{\gamma - 1}{\gamma}} \frac{L_i}{L_{c,n_{e0}}} c_s \quad (15.61)$$

where  $c_s = \sqrt{\gamma Z k_B T_{e0}/m_i}$  is the ion speed of sound,  $L_i$  is the ion inertial length and  $L_{c,n_{e0}} = |\partial \beta_0 / \partial r|^{-1}$  is the characteristic length scale of the density gradient. It can clearly be seen that the Thermal-magnetic waves will be come faster than sound waves on scales smaller than the ion inertial length, in a similar way whistler and Hall drift waves become faster than Alfvén waves on this same scale. These non-dispersive waves propagate in inhomogeneous plasmas on iso-density surfaces. In FLASH, the time step related to this new characteristic speed is computed in `Hydro_computeDt.F90` and compared with others speeds in order to get the stable correct global time step. The `cf1` condition will generally have to be lowered in the presence of the flux-based version of Biermann.

### 15.3.5.3 Resistive MHD

**15.3.5.3.1 How to use it ?** As indicated above, in order to use resistive MHD in FLASH, the user needs to set the runtime parameter `useMagneticResistivity` to `= .true..` Currently, three implementations of resistive MHD are available in FLASH: the "Constant" version, "SpitzerHighZ", as well as the "DaviesWen" one. Depending on which one the user wants to use, the proper path needs to be required in the Config file:

```
# For Constant resistivity
REQUIRES physics/materialProperties/MagneticResistivity/MagneticResistivityMain/Constant
```

```
# For SpitzerHighZ resistivity
```

REQUIRES physics/materialProperties/MagneticResistivity/MagneticResistivityMain/SpitzerHighZ

# For DaviesWen resistivity

REQUIRES physics/materialProperties/MagneticResistivity/MagneticResistivityMain/DaviesWen

See 23.2 for more details on each coefficients implementation. It is important for the user to know that for all implementations the quantity actually computed and used is the magnetic diffusivity (in units of  $l^2 t^{-1}$ ). Therefore, in the standard FLASH unit system (CGS), the value of the `resistivity` parameter should be in  $cm^2 s^{-1}$ . In the case of user-specific implementations of the “MagneticResistivity.F90” and “MagneticResistivityFullState.F90” files, the values returned to the code should also always be in units of  $l^2 t^{-1}$ .

There are three parameters that control how resistive MHD is solved in the code: `resistivityForm`, `useCrossMagRes`, and `resistivitySolver`. The `resistivityForm` parameter accepts values of “parallel”, “perpendicular” (default), or “fullaniso”, and this controls the form of the resistivity terms in the generalized Ohm’s law (eq. 15.31). A value of “parallel” solves  $\mathbf{E} = \frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J}$ , and a value of “perpendicular” solves  $\mathbf{E} = \frac{\eta_{\perp}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J}$ . A value of “fullaniso” will solve  $\mathbf{E} = \frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J} + \frac{(\eta_{\perp} - \eta_{\parallel})}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \times \mathbf{b}$ . Note that this has been rewritten as compared to eq. 15.31. When `useCrossMagRes` is set to `.true.`, the  $\eta_{\perp}$  term is included and solved for by adding it to the Hall term.

The `resistivitySolver` parameter accepts values of “explicit” (default), “implicit”, or “hybrid”. This controls which type of solver(s) is being used to solve the resistive terms. The explicit solver is a flux-based approach, and requires that the time step be limited for numerical stability. Thus, when using this explicit solver, the user should always set `useDiffuse` and `useDiffuseComputeDtMagnetic` to `.true.` (as well as give a reasonable value to `dt_diff_factor`). See 19.1.6 for details on the implicit solver and how to use it, but note that the implicit solver is currently limited to only work generally on the uniform grid implementation (setup shortcut `+ug`).

The “hybrid” solver splits the resistive terms and solves  $\frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J}$  implicitly, while solving  $\frac{(\eta_{\perp} - \eta_{\parallel})}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \times \mathbf{b}$  explicitly. Hence it only makes sense to use the hybrid solver with `resistivityForm = "fullaniso"`.

#### 15.3.5.4 Thermoelectric transport

**15.3.5.4.1 How to use it ?** Currently, three implementations of thermoelectric coefficients are available in FLASH: the “Constant” version, “EppHain”, as well as the “DaviesWen” one. More details on these coefficients can be found in 23.4. As indicated above, in order to use thermoelectric transport in FLASH, the user needs to set various runtime parameters to `.true.` to turn on different terms. For convenience, the thermoelectric terms in eq. 15.31 are rewritten as:

$$\mathbf{E} = -\frac{k\beta_{\parallel}}{e} \nabla T_e - \frac{k(\beta_{\perp} - \beta_{\parallel})}{e} (\mathbf{b} \times \nabla T_e) \times \mathbf{b} + \frac{k\beta_{\perp}}{e} (\nabla T_e \times \mathbf{b}). \quad (15.62)$$

These terms are each turned on/off, respectively, with runtime parameters `use_Seebeck`, `use_CrossField`, and `use_Nernst`.

The same runtime parameters also turn on/off analogous terms in the heat flux equation:

$$\mathbf{q} = -\frac{kT_e\beta_{\parallel}}{e} \mathbf{J} - \frac{kT_e(\beta_{\parallel} - \beta_{\perp})}{e} \mathbf{b} \times (\mathbf{b} \times \mathbf{J}) - \frac{kT_e\beta_{\perp}}{e} (\mathbf{b} \times \mathbf{J}). \quad (15.63)$$

Hence, in the current version of FLASH, the user cannot run a simulation with a term on in the induction equation (eq. 15.62) while leaving it off in the heat flux equation (eq. 15.63).

## 15.4 Radiation-Flux-Limiter Aware Hydrodynamics

RadFLAH is a variant of hydrodynamics meant to be used together with an implementation of radiation transport that is based on flux-limited diffusion. The hydro is “aware” of the flux limiter and has modified and additional terms.

The implementation is included under `physics/Hydro/HydroMain/unsplit_rad`. This variant implements additional terms in the FLD Rad-Hydro equations to handle streaming and transition-to-streaming regimes better (including in particular effects of radiation pressure) than the regular version of `unsplit Hydro` used for HEDP simulations and similar applications.

Setup with shortcut `+uhd3tR` instead of `+uhd3t`. This has had limited testing, mostly in 1D spherical geometry.

The current implementation assumes that the fluid is described by two temperatures (matter+radiation), and thus the “electron” component really represents “matter”; we hope to remove reliance on this assumption in the future. With this assumption in place, this code has to be used together with a “MatRad3” implementation of multiTemp Eos. See also `physics/Eos/Eos/EosMain/multiTemp/MatRad3/README`.

The current implementation assumes there is only one radiation group (gray opacities). We may extend this in the future.

Currently there is no equivalent variant for MHD - this should be straightforward to implement, and we plan to provide this in the future (probably under `unsplit_rad/MHD.StaggeredMesh`).

More information will be included here after draft publication of Chatzopoulos and Weide (2017).

Several example setups are available under `Simulation/SimulationMain/radflaHD/`.

## Chapter 16

# Incompressible Navier-Stokes Unit

The `IncompNS` unit solves incompressible Navier-Stokes equations in two or three spatial dimensions. The currently released implementation assumes constant density throughout the simulation domain.

Multistep and Runge-Kutta explicit projection schemes are used for time integration. These methods are described in Armfield & Street 2002, Yang & Balaras 2006, and Vanella *et al.* 2010. Implementations using a staggered grid arrangement are provided for both uniform grid (UG) and PARAMESH adaptive mesh refinement `Grid` implementations. The `MultigridMC` and `BiPCGStab` Poisson solvers can be employed for AMR cases, whereas the homogeneous trigonometric solver + PFFT can be used in UG. Typical velocity boundary conditions for this problem are implemented.

*More documentation to appear later.*

As of FLASH4.4, Simulations that use the `IncompNS` unit should be configured to use a special implementation of `Driver_evolveFlash` located in `Driver/DriverMain/INSfracstep`. (Including `IncompNS` in the units of a simulation will automatically pull this directory into the configuration.) This requirement may change in the future.



## Chapter 17

# Equation of State Unit

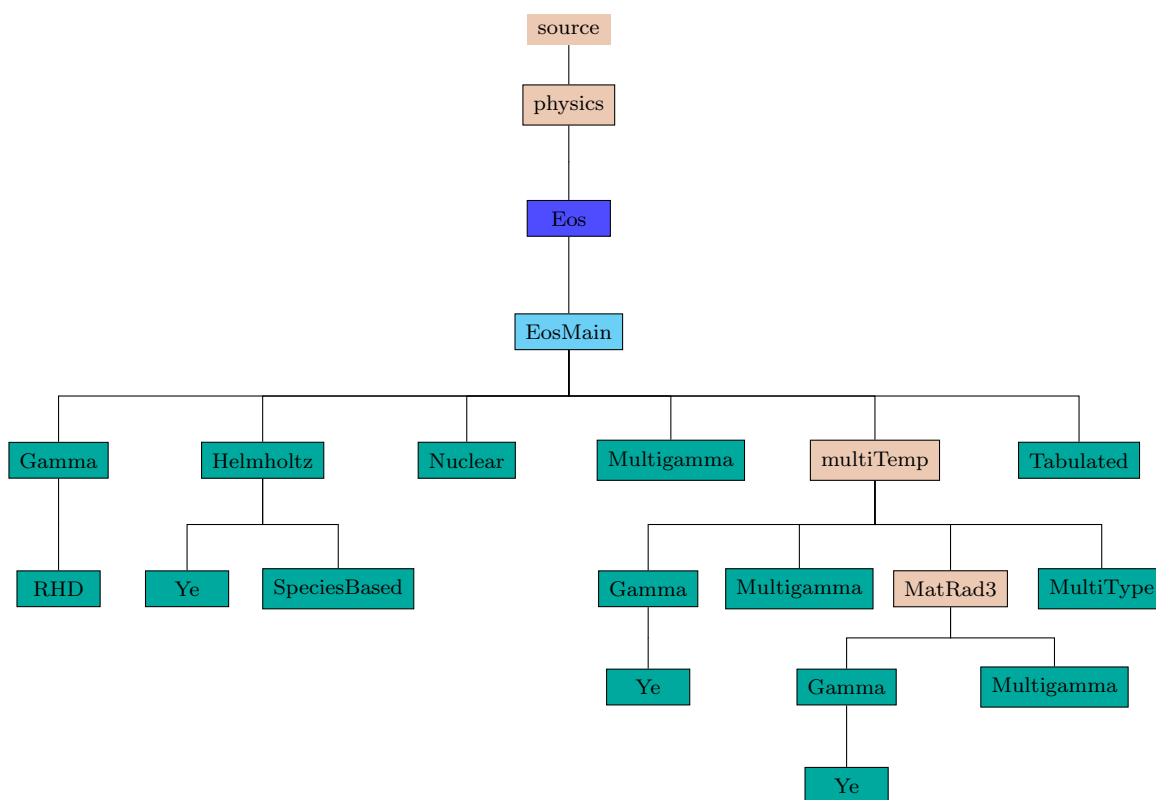


Figure 17.1: The Eos directory tree.

### 17.1 Introduction

The `Eos` unit implements the equation of state needed by the hydrodynamics and nuclear burning solvers. The function `Eos` provides the interface for operating on a one-dimensional vector. The same interface can be used for a single cell by reducing the vector size to 1. Additionally, this function can be used to find the thermodynamic quantities either from the density, temperature, and composition or from the density, internal energy, and composition. For user's convenience, a wrapper function (`Eos_wrapped`) is provided, which takes a section of a block and translates it into the data format required by the `Eos` function, then

calls the function. Upon return from the `Eos` function, the wrapper translates the returned data back to the same section of the block.

Four implementations of the (`Eos`) unit are available in the current release of FLASH4: `Gamma`, which implements a perfect-gas equation of state; `Gamma/RHD`, which implements a perfect-gas equation taking relativistic effects into account; `Multigamma`, which implements a perfect-gas equation of state with multiple fluids, each of which can have its own adiabatic index ( $\gamma$ ); and `Helmholtz`, which uses a fast Helmholtz free-energy table interpolation to handle degenerate/relativistic electrons/positrons and includes radiation pressure and ions (via the perfect gas approximation).

As described in previous sections, FLASH evolves the Euler equations for compressible, inviscid flow. This system of equations must be closed by an additional equation that provides a relation between the thermodynamic quantities of the gas. This relationship is known as the equation of state for the material, and its structure and properties depend on the composition of the gas.

It is common to call an equation of state (henceforth EOS) routine more than  $10^9$  times during a two-dimensional simulation and more than  $10^{11}$  times during the course of a three-dimensional simulation of stellar phenomena. Thus, it is very desirable to have an EOS that is as efficient as possible, yet accurately represents the relevant physics. While FLASH is capable of using any general equation of state, we discuss here the three equation of state routines that are supplied: an ideal-gas or gamma-law EOS, an EOS for a fluid composed of multiple gamma-law gases, and a tabular Helmholtz free energy EOS appropriate for stellar interiors. The two gamma-law EOSs consist of simple analytic expressions that make for a very fast EOS routine both in the case of a single gas or for a mixture of gases. The Helmholtz EOS includes much more physics and relies on a table look-up scheme for performance.

## 17.2 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

$$\gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial \rho} . \quad (17.1)$$

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

$$\gamma_4 = 1 + \frac{P}{\rho \epsilon} . \quad (17.2)$$

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 (17.2).

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

$$P = (\gamma - 1) \rho \epsilon . \quad (17.3)$$

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

$$P = \frac{N_a k}{\bar{A}} \rho T , \quad (17.4)$$

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

$$\frac{1}{\bar{A}} = \sum_i \frac{X_i}{A_i} , \quad (17.5)$$

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

$$\epsilon = \frac{1}{\gamma - 1} \frac{N_a k}{\bar{A}} T . \quad (17.6)$$



The relativistic variant of the ideal gas equation is explained in more detail in [Section 15.2](#).

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

$$\frac{1}{(\gamma - 1)} = \bar{A} \sum_i \frac{1}{(\gamma_i - 1)} \frac{X_i}{A_i} . \quad (17.7)$$

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.

### 17.2.1 Ideal Gamma Law for Relativistic Hydrodynamics

The relativistic variant of the ideal gas equation is explained in more detail in [Section 15.2](#).

## 17.3 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

$$P_{\text{tot}} = P_{\text{rad}} + P_{\text{ion}} + P_{\text{ele}} + P_{\text{pos}} + P_{\text{coul}} \quad (17.8)$$

$$\epsilon_{\text{tot}} = \epsilon_{\text{rad}} + \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{pos}} + \epsilon_{\text{coul}} . \quad (17.9)$$

Here the subscripts “rad,” “ion,” “ele,” “pos,” and “coul” represent the contributions from radiation, nuclei, electrons, positrons, and corrections for Coulomb effects, respectively. The radiation portion assumes a blackbody in local thermodynamic equilibrium, the ion portion (nuclei) is treated as an ideal gas with  $\gamma = 5/3$ , and the electrons and positrons are treated as a non-interacting Fermi gas.

The blackbody pressure and energy are calculated as

$$P_{\text{rad}} = \frac{aT^4}{3} \quad (17.10)$$

$$\epsilon_{\text{rad}} = \frac{3P_{\text{rad}}}{\rho} \quad (17.11)$$

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 [17.3](#) – [17.4](#)) 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

$$N_{\text{ele}} = \frac{8\pi\sqrt{2}}{h^3} m_e^3 c^3 \beta^{3/2} [F_{1/2}(\eta, \beta) + F_{3/2}(\eta, \beta)] \quad (17.12)$$

$$N_{\text{pos}} = \frac{8\pi\sqrt{2}}{h^3} m_e^3 c^3 \beta^{3/2} [F_{1/2}(-\eta - 2/\beta, \beta) + \beta F_{3/2}(-\eta - 2/\beta, \beta)] , \quad (17.13)$$

where  $h$  is Planck's constant,  $m_e$  is the electron rest mass,  $\beta = kT/(m_e c^2)$  is the relativity parameter,  $\eta = \mu/kT$  is the normalized chemical potential energy  $\mu$  for electrons, and  $F_k(\eta, \beta)$  is the Fermi-Dirac integral

$$F_k(\eta, \beta) = \int_0^\infty \frac{x^k (1 + 0.5 \beta x)^{1/2} dx}{\exp(x - \eta) + 1} . \quad (17.14)$$

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

$$N_{\text{ele,matter}} = \frac{\bar{Z}}{\bar{A}} N_a \rho = \bar{Z} N_{\text{ion}} , \quad (17.15)$$

and charge neutrality requires

$$N_{\text{ele,matter}} = N_{\text{ele}} - N_{\text{pos}} . \quad (17.16)$$

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,

$$F = \epsilon - T S \quad (17.17)$$

$$dF = -S dT + \frac{P}{\rho^2} d\rho , \quad (17.18)$$

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 (17.14) 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 (17.16). All partial derivatives of the pressure, entropy, and internal energy are formed analytically. Searches through the free energy table are avoided by computing hash indices from the values of any given  $(T, \rho\bar{Z}/\bar{A})$  pair. No computationally expensive divisions are required in interpolating from the table; all of them can be computed and stored the first time the EOS routine is called.

We note that the Helmholtz free energy table is constructed for only the electron-positron plasma, and it is a 2-dimensional function of density and temperature, *i.e.*  $F(\rho, T)$ . It is made with  $\bar{A} = \bar{Z} = 1$  (pure hydrogen), with an electron fraction  $Y_e = 1$ . One reason for not including contributions from photons and ions in the table is that these components of the Helmholtz EOS are very simple (Equations 17.10 – 17.11), 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  $10^{-10} < \rho < 10^{11}$  (g cm<sup>-3</sup>) and  $10^4 < T < 10^{11}$  (K). As with the gamma-law EOS, the Helmholtz EOS provides both forward and backward relations. In the case of the forward relation ( $\rho, T$ , given along with the composition) the table lookup scheme and analytic

Table 17.1: 1T and multitemperature Eos modes. The symbols are defined in `constants.h`. The column “1-T” indicates whether invoking `Eos` with this mode implies and/or ensures that the three components are in temperature equilibrium; a “yes” here implies that all temperatures provided as outputs are equal, and equal to  $T$  if  $T$  is listed as an input. For modes where component temperatures are allowed to stay different, the combined  $T$  has no physical meaning but indicates an “average” temperature to which the components would equilibrate if cells were held fixed without exchange of heat with neighboring cells. The columns RS indicates modes where, in the current multitemperature `Eos` implementations, a Newton-Raphson root search may be performed internally in order to compute some combined fluid quantities, in particular  $T$ . In this column “(y)” means the search is only necessary to get  $T$  and can otherwise be omitted. Subscripts i, e, and r stand for ions, electrons, and radiation, respectively,  $\epsilon$  for specific internal energy,  $s$  for specific entropy. Note that combined density  $\rho$  is always assumed an input and omitted from the table. Material composition and ionization levels are currently also just inputs.

symbol	inputs	outputs	1-T	RS
MODE_DENS_TEMP	$T$	$\epsilon, P$	yes	no
MODE_DENS_TEMP_EQUI	$T$	$\epsilon, P, T_i=T_e=T_r, \epsilon_i, \epsilon_e, \epsilon_r, s_e, P_i, P_e, P_r, \dots$	yes	no
MODE_DENS_TEMP_GATHER	$T_i, T_e, T_r$	$T, \epsilon, P, \epsilon_i, \epsilon_e, \epsilon_r, s_e, P_i, P_e, P_r, \dots$	no	(y)
MODE_DENS_EI	$\epsilon$	$T, P$	yes	yes
MODE_DENS_EI_SCATTER	$\epsilon$	$T, P, T_i=T_e=T_r, \epsilon_i, \epsilon_e, \epsilon_r, s_e, P_i, P_e, P_r, \dots$	yes	yes
MODE_DENS_EI_GATHER	$\epsilon_i, \epsilon_e, \epsilon_r$	$T, \epsilon, P, T_i, T_e, T_r, s_e, P_i, P_e, P_r, \dots$	no	(y)
MODE_DENS_EI_SELE_GATHER	$\epsilon, s_e, \epsilon_r$	$T, P, T_i, T_e, T_r, \epsilon_i, \epsilon_e, P_i, P_e, P_r, \dots$	no	(y)
MODE_DENS_PRES	$P$	$T, \epsilon$	yes	yes

formulae directly provide relevant thermodynamic quantities. In the case of the backward relation ( $\rho, \epsilon$ , and composition given), the routine performs a Newton-Raphson iteration to determine temperature. It is possible for the input variables to be changed in the iterative modes since the solution is not exact. The returned quantities are thermodynamically consistent.

## 17.4 Multitemperature extension for Eos

Extended functionality is required of the `Eos` in order to work a multitemperature simulation. Such simulations need to use one of the implementations under the `multiTemp` directory. When the `Hydro multiTemp` code is used, as described in [Section 15.1.5](#), one of the implementations under `physics/Eos/EosMain/multiTemp` must be used. Additional functionality includes

- Support for additional `Hydro` variables that describe the state for the fluid components (ions, electrons, radiation).
- Additional `Eos` modes. See [Table 17.1](#).

### 17.4.1 Gamma

The multitemperature `Gamma` EOS implementation models the ion as well as the electron components as ideal gases, and the radiation component as black body thermal radiation (like the Helmholtz 1T implementation, see [Section 17.3](#)). It is a limitation of this implementation that the ionization state of ions is assumed to be fixed.

This `Eos` implementation has several runtime parameters, in part inherited from the 1T implementation, that can technically be changed but for which different values can lead to a nonsensical or inconsistent model. In particular, the values of `gamma`, `gammaIon`, `gammaEle`, and `gammaRad` should not be changed from their default values (5./3. for the first four and 4./3. for the latter). Other values have not been tested.

This EOS can include contributions from radiation, partially or completely ionized nuclei, and electrons. The desired (constant) ionization level  $\bar{Z}$  of the nuclei should be specified with the `eos_singleSpeciesZ`

runtime parameter. The `eos_singleSpeciesA` runtime parameter specifies the average atomic mass  $\bar{A}$  and should be set to the mass of one atom of the material in atomic mass units. For example, for a plasma of fully ionized Carbon-12, one would set `eos_singleSpeciesA` = 12.0 and `eos_singleSpeciesZ` = 6.0.

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

$$P_{\text{tot}} = P_{\text{ion}} + P_{\text{ele}} + P_{\text{rad}} \quad (17.19)$$

$$\epsilon_{\text{tot}} = \epsilon_{\text{ion}} + \epsilon_{\text{ele}} + \epsilon_{\text{rad}} . \quad (17.20)$$

Here the subscripts “ion,” “ele,” and “rad” represent the contributions from nuclei, electrons, and radiation, respectively. The radiation portion assumes a black body in local thermodynamic equilibrium, the ion portion (nuclei) is treated as an ideal gas with  $\gamma = 5/3$ , and the electrons are treated as a classical ideal gas as well.

As for the 1T Helmholtz implementation, the black body pressure and energy relate to the radiation “temperature” by

$$P_{\text{rad}} = \frac{aT_{\text{rad}}^4}{3} \quad (17.21)$$

$$\epsilon_{\text{rad}} = \frac{3P_{\text{rad}}}{\rho} \quad (17.22)$$

where  $a$  is related to the Stephan-Boltzmann constant  $\sigma_B = ac/4$ , and  $c$  is the speed of light.

Like 1T implementations, all multitemperature implementations of `Eos` must return  $\gamma = \gamma_1 = \frac{\rho}{P} \frac{\partial P}{\partial \rho}$ , and  $\gamma_4$  is calculated from (17.2). These two generalizations of adiabatic indices are usually stored as the mesh-based variables `GAMC_VAR` and `GAME_VAR` as a result of calling `Eos_wrapped`.

#### 17.4.1.1 Gamma/Ye

The `Ye` variant of `Gamma` implementation is like the `Gamma` implementation, except that  $\bar{A}$  and  $\bar{Z}$  need not be constant throughout the domain but can vary in different parts of the fluid.

Typically, a simulation would implement a `Simulation_initBlock` that initializes mass scalar variables `sumy` and `ye` as desired. The initial values then get advected with the motion of the fluid. `Eos` interprets them as

$$\text{sumy} = \sum Y_i \quad (17.23)$$

and

$$\text{ye} = Y_e \quad (17.24)$$

and computes

$$\text{abar} = \bar{A} = \frac{1}{\sum Y_i} \quad (17.25)$$

and

$$\text{zbar} = \bar{Z} = Y_e \bar{A} \quad (17.26)$$

from them.

#### 17.4.2 Multigamma

First a clarification, to avoid confusion: We will call ions, electrons, and radiation the three **components** of the fluid; and we designate different kinds of materials as different **species**. A fluid with several materials contributes its ions to the common ion component and its electrons to the common electron component. Currently, all adiabatic coefficients  $\gamma_i$  describing the electrons and the various kinds of ions should be 5/3 as for a monatomic ideal gas (despite the implementation’s name!). Mixing of species in a cell results in a fluid whose matter components are described by average values  $\bar{A}$  and  $\bar{Z}$ , which depends on the local composition as well as on the  $A_i$  and  $Z_i$  values of the various species.

In our multitemperature approximation, the three components may have different temperatures, but there is no notion of a per-species temperature.

The multitemperature `Multigamma` implementation is another variation on the multitemperature `Gamma` theme. Except for the different way of getting  $\bar{A}$  and  $\bar{Z}$ , all else goes as described for `Gamma`, above.

### 17.4.3 Tabulated

The Tabulated Eos implementation is currently only available for simulations configured in 3T mode. The currently supported table formats are IONMIX1, IONMIX4, and SESAME. See the Opacity section [Section 23.5.6](#) for a description of the IONMIX1 and IONMIX4 table file format. SESAME’s file format will be described here. At the moment, the SESAME tables read in the code are the 303 and 304 tables for the ion and electron EOS respectively. Opacities (if required) for simulations with SESAME need to be provided by some other source.

Tables are read into memory at initialization. For both IONMIX4 and SESAME, tables for  $\bar{z}(T, N_{ion})$ ,  $E_i(T, N_{ion})$ ,  $E_e(T, N_{ion})$ ,  $P_i(T, N_{ion})$ , and  $P_e(T, N_{ion})$  are used.

Eos modes like `MODE_DENS_TEMP`, `MODE_DENS_TEMP_component`, `MODE_DENS_TEMP_EQUI` compute  $N_{ion}$  from  $\text{dens}$  and directly look up the output quantities in the appropriate tables. For other modes, root searches would be required.

However, the Tabulated Eos implementation is currently not being used on its own but only in connection with the Multitype implementation.

#### 17.4.3.1 SESAME TEOS

The SESAME database is a TEOS although rather than solely relying on theoretical models, it is physically motivated; see LA-UR-92-407, and Crockett (1999). Experimental data check in different regions the consistency of the thermodynamic interpolations.

SESAME involves tabulated equilibrium data Pimentel & Sheppard (2018) in logically-Cartesian tables (see LA-UR-21-23834), using two state functions (e.g., temperature and density) to populate the phase-space with other macroscopic properties like pressure, internal energy, for 150+ materials.

Therefore SESAME, being publicly available, contains the necessary physics for simulating astrophysical and HEDP systems using FLASH4. Nevertheless, the database is not fully compatible with the code outright. Data entries may fall into solid-state regimes which have negative pressures and/or energies, for which the current FLASH4 will predict non-physical wave speeds.

For this reason, FLASH4 typically initializes a given material at a higher temperature to avoid pathologies (see [Figure 17.2](#)).

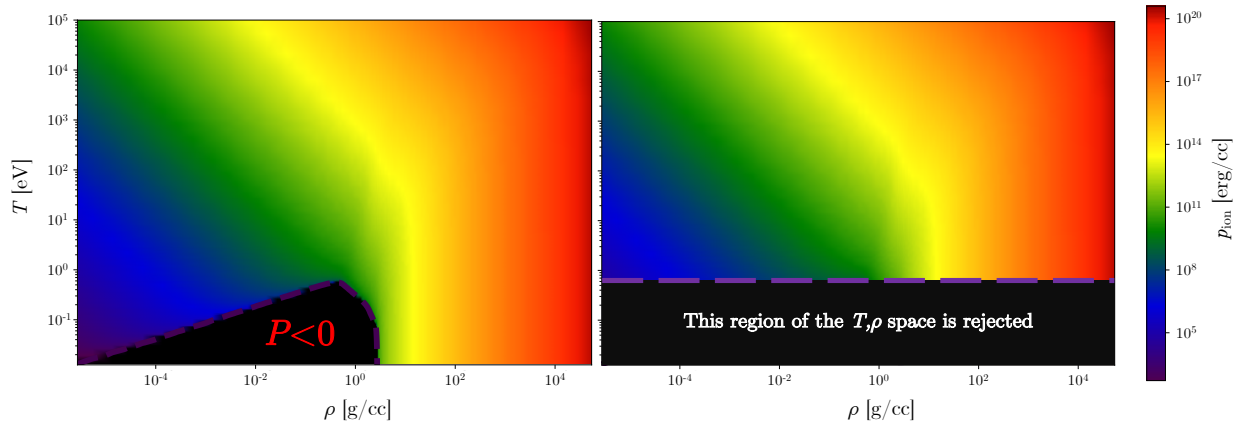


Figure 17.2: Schematic showing the temperature initialization in `eos_tabReadSesameTables.F90` for the  $p_{ion}$  from table 303 for 3719 (Aluminum). Solid-state values ( $P < 0$ ) on the left, are excluded using a  $T$ -point above the “solid” region (right).

#### 17.4.3.2 Interpolation strategy for SESAME

The SESAME tables of interest are selected by reading through the material property file to isolate the data associated with specific EOS headers. SESAME provides several TEOS options. The framework of the

current FLASH4 code requires tables 303 and 304, the ion and electron TEOS entries, respectively.

The reader, found in `eos_tabReadSesameTables.F90` truncates the range in the  $(\rho, T)$  space, eliminating data of problematic regions, i.e., negative entries.

With the working range defined, FLASH4 passed the data to `eos_tabEditSesameTables.F90`, which scans the ranges of the two truncated 303 and 304 matrices to determine their relative orientation, as shown in Figure 17.3.

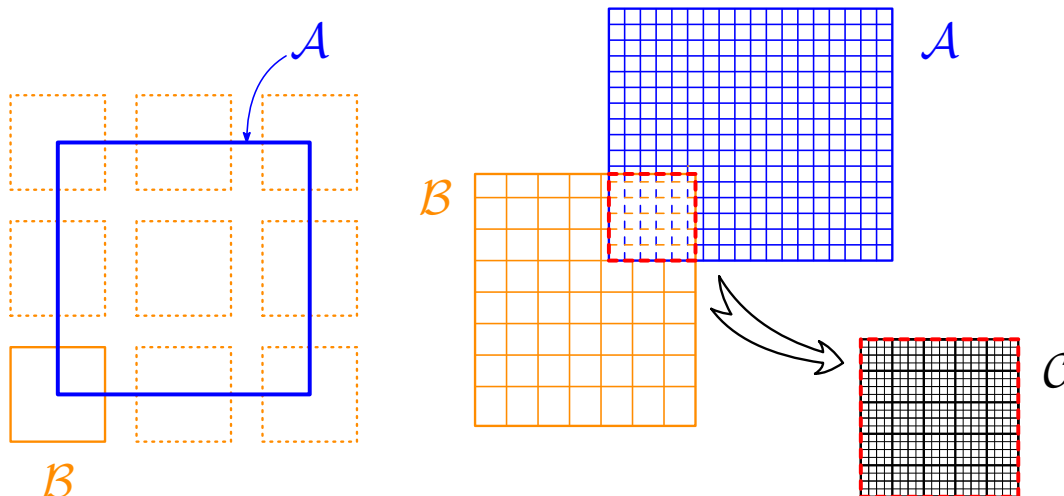


Figure 17.3: Schematic showing the overlap of the interpolated grids  $\mathcal{A}$ , and  $\mathcal{B}$ , and the common interpolation grid  $\mathcal{C}$  inside `eos_tabEditSesameTables.F90`.

Once FLASH4 determines their relative orientation, it can pinpoint the overlapping region in the  $(\rho, T)$  space of the concatenated 303 and 304 tables; there are represented by  $\mathcal{A}$ , and  $\mathcal{B}$ , in Figure 17.3.

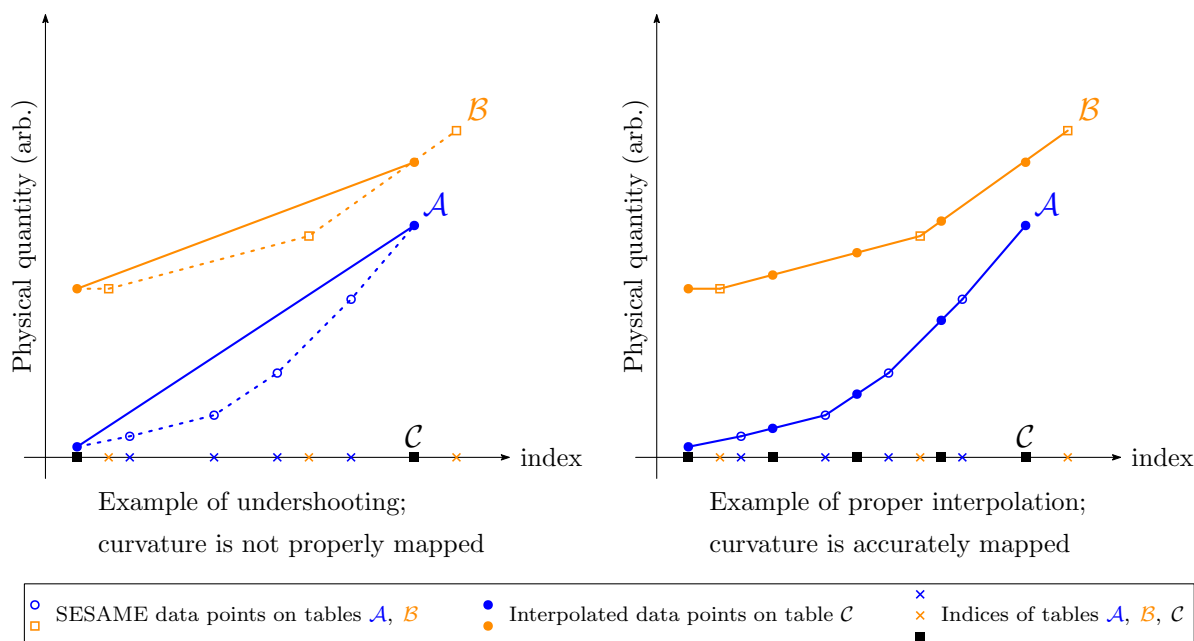


Figure 17.4: Schematic showing the utility of a highly-resolved  $\mathcal{C}$  to capture the slopes in SESAME.

By implication, this allows the code to establish a common temperature and density envelope for the simulation. The new bounds direct the creation of a new, very fine grid (matrix  $\mathcal{C}$ , see Figure 17.3), upon which the physical values of the 303 and 304 EOS tables are interpolated; its high resolution,  $100 \times 100$  points across  $(\rho, T)$ , ensures that the derived table accurately captures the slopes of constitutive quantities (see Figure 17.4).

Bilinear interpolation is used to populate matrix  $\mathcal{C}$ . The user is advised not to initialize the temperature and/or density far outside the upper and lower bounds set by the  $\mathcal{C}$  grid. The slopes will be artificially fixed, and there can be no guarantee on the physicality of the extrapolated state.

#### 17.4.3.3 Use of the SESAME database

The FLASH4 code requires ionization information on top of the thermodynamic quantities provided by SESAME. Thus, interpolating the 303 and 304 matrices using the uniform grid, shown in the previous figures as table  $\mathcal{C}$ , allows the generation of an additional table mapping the ionization of the material across the interpolated  $(\rho, T)$  range.

Calculation of the average ionization of the material follows the formulary description found in Atzeni & Meyer-ter-Vehn (2004).

#### 17.4.3.4 Types and structures thereof of SESAME data records

Below, the SESAME format is defined. Certain information is ignored by FLASH, and is presently not used for any calculations. The format is extracted by LA-UR-92-407, and focuses on information which is specifically relevant to FLASH4. The reader is encouraged to read more about the SESAME TEOS database from this and related documents.

It should be noted that table 306 and table series 400, 500, and 600 are ignored by FLASH4.

1. Header structure (general):
  - (a) Material identification number (e.g. 9999)
  - (b) Table identification number (e.g. 101)
  - (c) Number of words per table (each line is 80 characters)
  - (d) Date created and last modified
  - (e) Version number
2. Tables 101 and 102 contain comments
3. Table 201:
  - (a) Mean atomic number  $\bar{Z}$
  - (b) Mean atomic mass  $\bar{A}$
  - (c) Normal solid density  $\rho_o$
  - (d) Solid bulk modulus  $B_o$  (ignored by FLASH)
  - (e) Exchange coefficient  $C_{ex}$  (ignored by FLASH)
4. Tables 300-series: 301-305 (FLASH uses only 303 and 304)
  - (a) Number of specie density points (ndens is word 1)
  - (b) Number of temperature points (ntemp is word 2)
  - (c) Density array (`densnn(id), id=1,ndens`) [ $g/cm^3$ ] (starts at word 3)
  - (d) Temperature array (`tplsma(it), it=1,ntemp`) [K] (starts at `ndens+3` word)
  - (e) Pressure array (`(press(id,it), id=1,ndens),it=1,ntemp`) [GPa] (starts at `ndens+ntemp+3` word)
  - (f) Energy array (`(enrgy(id,it), id=1,ndens),it=1,ntemp`) [MJ/kg] (starts at `ndens+ntemp+3+ndens*ntemp` word)
  - (g) Helmholtz free energy array [MJ/kg] (ignored by FLASH)



### 17.4.3.5 Example format of a SESAME table

Below, the exact file format for SESAME is included. It shows how the file should look like to be usable in FLASH. The 300-series have identical mock data in a  $(\rho, T)$  grid  $38 \times 27$  wide, including solid-state entries.

```

0 9999 101 240 r 22481 102082 4 0
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxx
1 9999 101 232 r 22481 102082 4 1
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
1 9999 201 5 r 22481 102082 4 1
2.000000000000000E+00 4.002600000000000E+00 4.000000000000000E-01 0.000000000000000E+00 0.000000000000000E+00111111
1 9999 301 2119 r 22481 102082 4 1
3.800000000000000E+01 2.700000000000000E+01 0.000000000000000E+00 1.000000000000000E-06 1.000000000000000E-05111111
1.000000000000000E-04 2.154434690000000E-04 4.641588830000000E-04 1.000000000000000E-03 2.154434690000000E-03111111
4.641588830000000E-03 2.000000000000000E-02-1.465243200000000E-02 2.146937640000000E-02 3.145785770000000E-02111111
4.609341210000000E-02 6.753805870000000E-02 9.895968120000000E-02-1.450000000000000E-01-1.910612320000000E-01111111
. . .
1 9999 303 2119 r 22481 102082 4 1
3.800000000000000E+01 2.700000000000000E+01 0.000000000000000E+00 1.000000000000000E-06 1.000000000000000E-05111111
1.000000000000000E-04 2.154434690000000E-04 4.641588830000000E-04 1.000000000000000E-03 2.154434690000000E-03111111
4.641588830000000E-03 2.000000000000000E-02-1.465243200000000E-02 2.146937640000000E-02 3.145785770000000E-02111111
4.609341210000000E-02 6.753805870000000E-02 9.895968120000000E-02-1.450000000000000E-01-1.910612320000000E-01111111
. . .
1 9999 304 2119 r 22481 102082 4 1
3.800000000000000E+01 2.700000000000000E+01 0.000000000000000E+00 1.000000000000000E-06 1.000000000000000E-05111111
1.000000000000000E-04 2.154434690000000E-04 4.641588830000000E-04 1.000000000000000E-03 2.154434690000000E-03111111
4.641588830000000E-03 2.000000000000000E-02-1.465243200000000E-02 2.146937640000000E-02 3.145785770000000E-02111111
4.609341210000000E-02 6.753805870000000E-02 9.895968120000000E-02-1.450000000000000E-01-1.910612320000000E-01111111
. . .

```

Users might find that their SESAME table uses a different data format from the expected default E22.15. If that is the case the desired number format may be specified on a per species basis using the runtime parameter `eos_specSesDblFmt`.

### 17.4.4 Multitype

Multitype implementation does Eos computations by combining Eos calls for different materials. A material is a species in the sense of the Multispecies unit. Properties of materials relevant to the Multitype Eos are thus kept in the Multispecies database.

The Multitype implementation can combine materials of different eos types. It can be used for 3T Eos modes. When used in this generally, we speak of MTMMMT Eos - multitemperature multimaterial multitype.

Currently, the Eos types that can be combined are:

- Gamma,
- Tabulated.

The Multitype implementation calls `eos_idealGamma` and `eos_tabIonmix` only in `MODE_DENS_TEMP_component` modes.

The prescription for combining results of such calls is currently simple: Make Eos calls for different materials separately, passing the material's partial density into the Eos as the density. The per-material calls are thus not aware of other materials in the same cell. Results are combined as appropriate: pressures are taken as partial pressures and are added up to give the total multimaterial pressure; specific energies are multiplied with partial densities to give energy densities, which are added up.

For 3T modes, the above is applied separately for ion and electron components, resulting in separate total energies, pressures, *etc.*, for ions and electrons. Newton-Raphson root search is done to implement Eos modes that take energies or pressures as inputs (like `MODE_DENS_EI*`, `MODE_DENS_PRE` [not completely implemented]), separately for ions and electrons if necessary. The blackbody radiation component is added.



To use MTMMMT, use the `+mtmmmt` setup shortcut. See the `LaserSlab` simulation for an example. See the `Multispecies` unit on initializing material properties.

## 17.5 MultiFluid

The `multiFluid` implementation of the equation of state is meant to complement the volume-of-fluid (VOF) hydrodynamics solver for capturing material interfaces described in [Section 15.1.4](#). The EoS operates as previously described for any cells that are purely a single fluid. In mixed cells the EoS is called separately on each of the components, here labelled by  $\alpha$ , for the state  $(f^\alpha \rho^\alpha, f^\alpha E^\alpha, P^\alpha)$ . After a call to the EoS it is possible that  $P^\alpha \neq P^\beta$ . In order to maintain the mechanical equilibrium assumed in VOF the volume fractions,  $f^\alpha$  need to be relaxed until the two pressures equilibrate. First an average pressure is computed,

$$P = \frac{\sum_\alpha f^\alpha / \gamma^\alpha}{\sum_\alpha \frac{f^\alpha}{P^\alpha \gamma^\alpha}}, \quad (17.27)$$

and the component fluid states are adjusted as

$$\Delta f^\alpha = \frac{f^\alpha}{\gamma^\alpha} \left(1 - \frac{P}{P^\alpha}\right) \quad (17.28)$$

$$f^\alpha \leftarrow f^\alpha + \Delta f^\alpha \quad (17.29)$$

$$\rho^\alpha \leftarrow \frac{f^\alpha \rho^\alpha}{f^\alpha + \Delta f^\alpha} \quad (17.30)$$

$$f^\alpha \rho^\alpha E^\alpha \leftarrow f^\alpha \rho^\alpha E^\alpha - P \Delta f^\alpha. \quad (17.31)$$

In the above  $\gamma^\alpha$  is the adiabatic index for fluid  $\alpha$  as in other EoS implementations,

$$\gamma^\alpha \equiv \frac{\rho^\alpha c_\alpha^2}{P^\alpha}. \quad (17.32)$$

An equivalent single fluid state may also be parameterized in mixed cells as

$$\rho = \sum f^\alpha \rho^\alpha \quad (17.33)$$

$$E = \sum f^\alpha \rho^\alpha E^\alpha \quad (17.34)$$

$$\gamma = \left( \sum \frac{f^\alpha}{\gamma^\alpha} \right)^{-1}. \quad (17.35)$$

## 17.6 Usage

### 17.6.1 Initialization

The initialization function of the Eos unit `Eos_init` is fairly simple for the two ideal gas gamma law implementations included. It gathers the runtime parameters and the physical constants needed by the equation of state and stores them in the data module. The Helmholtz EOS `Eos_init` routine is a little more complex. The `Helmholtz` EOS requires an input file `helm_table.dat` that contains the lookup table for the electron contributions. This table is currently stored in ASCII for portability purposes. When the table is first read in, a binary version called `helm_table.bdat` is created. This binary format can be used for faster subsequent restarts on the same machine but may not be portable across platforms. The `Eos_init` routine reads in the table data on processor 0 and broadcasts it to all other processors.

### 17.6.2 Runtime Parameters

Runtime parameters for the `Gamma` unit require the user to set the thermodynamic properties for the single gas. `gamma`, `eos_singleSpeciesA`, `eos_singleSpeciesZ` set the ratio of specific heats and the nucleon and

proton numbers for the gas. In contrast, the **Multigamma** implementation does not set runtime parameters to define properties of the multiple species. Instead, the simulation **Config** file indicates the requested species, for example helium and oxygen can be defined as

```
SPECIES HE4
SPECIES O16
```

The properties of the gases are initialized in the file **Simulation\_initSpecies.F90**, for example

```
subroutine Simulation_initSpecies()
  use Multispecies_interface, ONLY : Multispecies_setProperty
  implicit none
#include "Flash.h"
#include "Multispecies.h"
  call Multispecies_setProperty(HE4_SPEC, A, 4.)
  call Multispecies_setProperty(HE4_SPEC, Z, 2.)
  call Multispecies_setProperty(HE4_SPEC, GAMMA, 1.66666666667e0)
  call Multispecies_setProperty(O16_SPEC, A, 16.0)
  call Multispecies_setProperty(O16_SPEC, Z, 8.0)
  call Multispecies_setProperty(O16_SPEC, GAMMA, 1.4)
end subroutine Simulation_initSpecies
```

For the Helmholtz equation of state, the table-lookup algorithm requires a given temperature and density. When temperature or internal energy are supplied as the input parameter, an iterative solution is found. Therefore, no matter what mode is selected for **Helmholtz** input, the best initial value of temperature should be provided to speed convergence of the iterations. The iterative solver is controlled by two runtime parameters **eos\_maxNewton** and **eos\_tolerance** which define the maximum number of iterations and convergence tolerance. An additional runtime parameter for **Helmholtz**, **eos\_coulombMult**, indicates whether or not to apply Coulomb corrections. In some regions of the  $\rho$ - $T$  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.

### 17.6.3 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.

## 17.7 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_TEMP` 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 18

# Local Source Terms

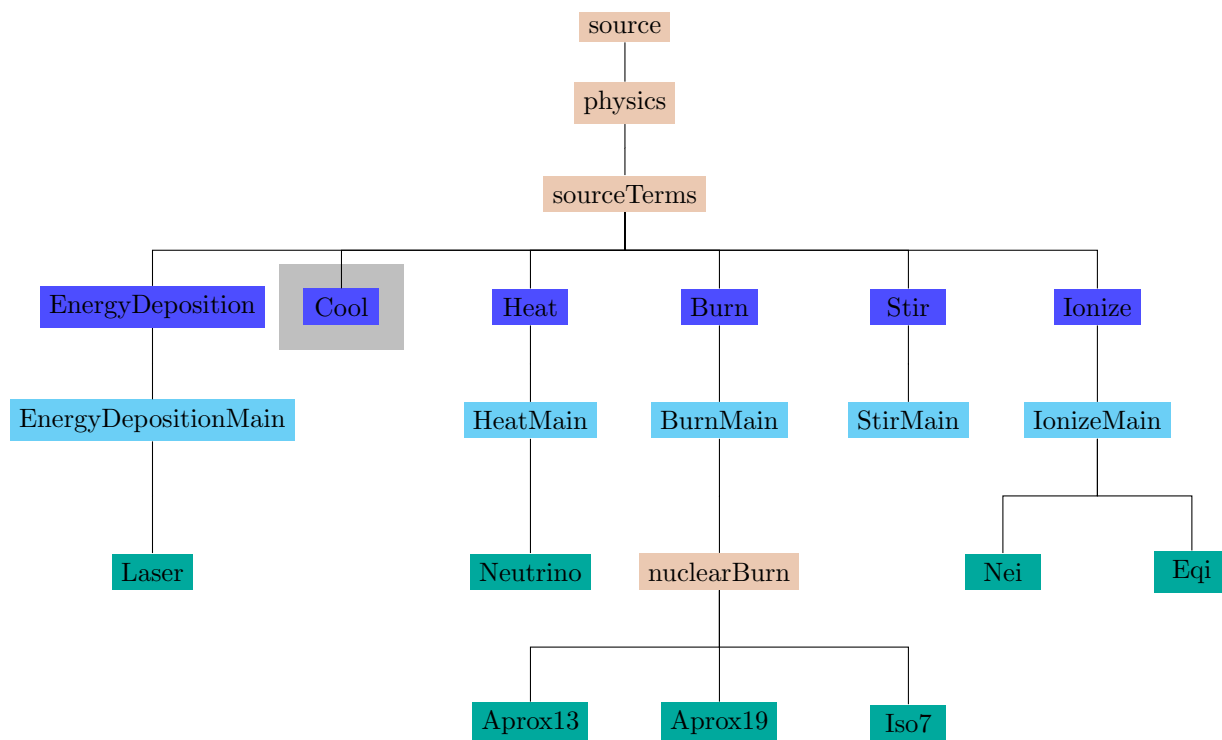


Figure 18.1: The organizational structure of physics source terms, which include units such as `Burn` and `Stir`. Shaded units include only stub implementations. Only a subset of units is shown.

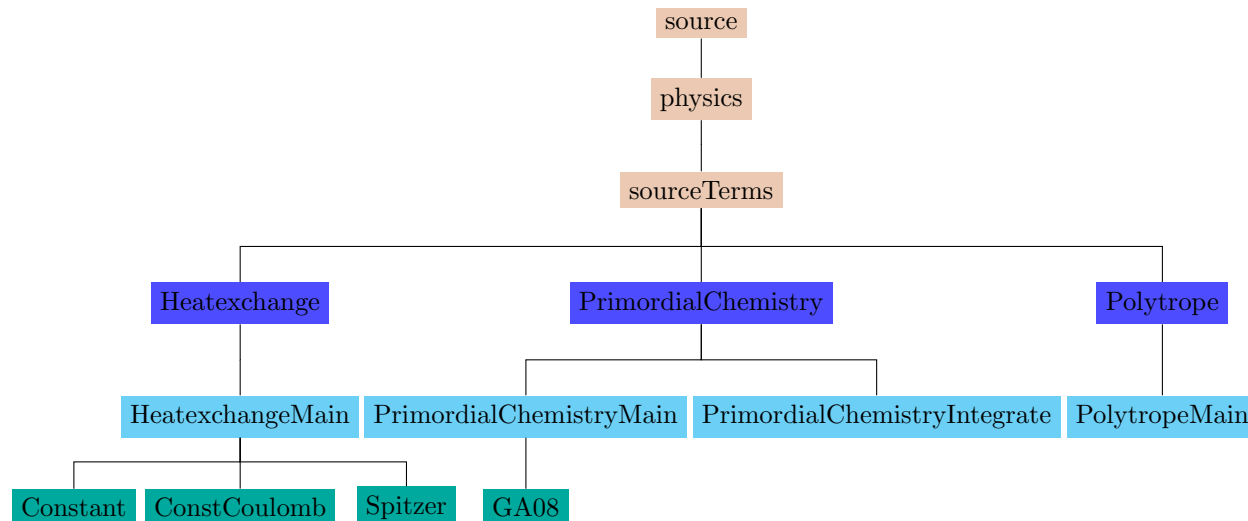


Figure 18.2: The organizational structure of physics source terms, with additional units, including `Heatexchange`.

The `physics/sourceTerms` organizational directory contains several units that implement forcing terms. The `Burn`, `Stir`, `Ionize`, and `Diffuse` units contain implementations in FLASH4. Two other units, `Cool` and `Heat`, contain only stub level routines in their API.

## 18.1 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.

### 18.1.1 Algorithms

Modeling thermonuclear flashes typically requires the energy generation rate due to nuclear burning over a large range of temperatures, densities and compositions. The average energy generated or lost over a period of time is found by integrating a system of ordinary differential equations (the nuclear reaction network) for the abundances of important nuclei and the total energy release. In some contexts, such as supernova models, the abundances themselves are also of interest. In either case, the coefficients that appear in the equations are typically extremely sensitive to temperature. The resulting stiffness of the system of equations requires the use of an implicit time integration scheme.

A user can choose between two implicit integration methods and two linear algebra packages in FLASH. The runtime parameter `odeStepper` controls which integration method is used in the simulation. The choice `odeStepper = 1` is the default and invokes a Bader-Deuffhard scheme. The choice `odeStepper = 2` invokes a Kaps-Rentrop or Rosenbrock scheme. The runtime parameter `algebra` controls which linear algebra package is used in the simulation. The choice `algebra = 1` is the default and invokes the sparse matrix MA28 package. The choice `algebra = 2` invokes the GIFT linear algebra routines. While any combination of the integration methods and linear algebra packages will produce correct answers, some combinations may execute more efficiently than others for certain types of simulations. No general rules have been found for best

combination for a given simulation. The most efficient combination depends on the timestep being taken, the spatial resolution of the model, the values of the local thermodynamic variables, and the composition. Users are advised to experiment with the various combinations to determine the best one for their simulation. However, an extensive analysis was performed in the Timmes paper cited below.

Timmes (1999) reviewed several methods for solving stiff nuclear reaction networks, providing the basis for the reaction network solvers included with FLASH. The scaling properties and behavior of three semi-implicit time integration algorithms (a traditional first-order accurate Euler method, a fourth-order accurate Kaps-Rentrop / Rosenbrock method, and a variable order Bader-Deuflhard method) and eight linear algebra packages (LAPACK, LUDCMP, LEQS, GIFT, MA28, UMFPACK, and Y12M) were investigated by running each of these 24 combinations on seven different nuclear reaction networks (hard-wired 13- and 19-isotope networks and soft-wired networks of 47, 76, 127, 200, and 489 isotopes). Timmes' analysis suggested that the best balance of accuracy, overall efficiency, memory footprint, and ease-of-use was provided by the two integration methods (Bader-Deuflhard and Kaps-Rentrop) and the two linear algebra packages (MA28 and GIFT) that are provided with the FLASH code.

### 18.1.2 Reaction networks

We begin by describing the equations solved by the nuclear burning unit. We consider material that may be described by a density  $\rho$  and a single temperature  $T$  and contains a number of isotopes  $i$ , each of which has  $Z_i$  protons and  $A_i$  nucleons (protons + neutrons). Let  $n_i$  and  $\rho_i$  denote the number and mass density, respectively, of the  $i$ th isotope, and let  $X_i$  denote its mass fraction, so that

$$X_i = \rho_i / \rho = n_i A_i / (\rho N_A) , \quad (18.1)$$

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

$$Y_i = X_i / A_i = n_i / (\rho N_A) . \quad (18.2)$$

Mass conservation is then expressed by

$$\sum_{i=1}^N X_i = 1 . \quad (18.3)$$

At the end of each timestep, FLASH checks that the stored abundances satisfy (18.3) 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

$$\frac{dY_i}{dt} + \nabla \cdot (Y_i \mathbf{V}_i) = \dot{R}_i . \quad (18.4)$$

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

$$\dot{R}_i = \sum_{j,k} Y_j Y_k \lambda_{kj}(l) - Y_i Y_j \lambda_{jk}(i) , \quad (18.5)$$

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 (18.5). The mass diffusion velocities  $\mathbf{V}_i$  in (18.4) are obtained from the solution of a multicomponent diffusion equation (Chapman & Cowling 1970; Burgers 1969; Williams 1988) and reflect the fact that mass diffusion processes arise from pressure, temperature, and/or abundance gradients as well as from external gravitational or electrical forces.

The case  $\mathbf{V}_i \equiv 0$  is important for two reasons. First, mass diffusion is often unimportant when compared to other transport processes, such as thermal or viscous diffusion (*i.e.*, large Lewis numbers and/or small Prandtl numbers). Such a situation obtains, for example, in the study of laminar flame fronts propagating through the quiescent interior of a white dwarf. Second, this case permits the decoupling of the reaction

network solver from the hydrodynamical solver through the use of operator splitting, greatly simplifying the algorithm. This is the method used by the default FLASH distribution. Setting  $\mathbf{V}_i \equiv 0$  transforms (18.4) into

$$\frac{dY_i}{dt} = \dot{R}_i, \quad (18.6)$$

which may be written in the more compact, standard form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}). \quad (18.7)$$

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 (18.6) and (18.7) 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  $\tilde{\mathbf{J}} \equiv \partial \mathbf{f} / \partial \mathbf{y}$  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 (18.6) 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 (18.7) through the terms

$$\begin{aligned} \dot{Y}({}^4\text{He}) &= -Y({}^4\text{He}) Y({}^{12}\text{C}) R + \dots \\ \dot{Y}({}^{12}\text{C}) &= -Y({}^4\text{He}) Y({}^{12}\text{C}) R + \dots \\ \dot{Y}({}^{16}\text{O}) &= +Y({}^4\text{He}) Y({}^{12}\text{C}) R + \dots, \end{aligned} \quad (18.8)$$

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

$$\begin{aligned} J({}^4\text{He}, {}^4\text{He}) &= -Y({}^{12}\text{C}) R + \dots & J({}^4\text{He}, {}^{12}\text{C}) &= -Y({}^4\text{He}) R + \dots \\ J({}^{12}\text{C}, {}^4\text{He}) &= -Y({}^{12}\text{C}) R + \dots & J({}^{12}\text{C}, {}^{12}\text{C}) &= -Y({}^4\text{He}) R + \dots \\ J({}^{16}\text{O}, {}^4\text{He}) &= +Y({}^{12}\text{C}) R + \dots & J({}^{16}\text{O}, {}^{12}\text{C}) &= +Y({}^4\text{He}) R + \dots \end{aligned} \quad (18.9)$$

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

This release of FLASH4 contains three reaction networks. A seven-isotope alpha-chain (`Iso7`) is useful for problems that do not have enough memory to carry a larger set of isotopes. The 13-isotope  $\alpha$ -chain plus heavy-ion reaction network (`Aprox13`) is suitable for most multi-dimensional simulations of stellar phenomena, where having a reasonably accurate energy generation rate is of primary concern. The 19-isotope reaction network (`Aprox19`) has the same  $\alpha$ -chain and heavy-ion reactions as the 13-isotope network, but it includes additional isotopes to accommodate some types of hydrogen burning (PP chains and steady-state CNO cycles), along with some aspects of photo-disintegration into  ${}^{54}\text{Fe}$ . This 19 isotope reaction network is described in Weaver, Zimmerman, & Woosley (1978).

The networks supplied with FLASH are examples of a “hard-wired” reaction network, where each of the reaction sequences are carefully entered by hand. This approach is suitable for small networks, when minimizing the CPU time required to run the reaction network is a primary concern, although it suffers the disadvantage of inflexibility.



### 18.1.2.1 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  $\mathbf{\tilde{A}} \cdot \mathbf{x} = \mathbf{b}$  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  $\mathbf{\tilde{A}}$  is reduced to upper triangular form, and backsubstitution with the right-hand side  $\mathbf{b}$  yields the solution to  $\mathbf{\tilde{A}} \cdot \mathbf{x} = \mathbf{b}$ . GIFT generated routines skip all calculations with matrix elements that are zero; in this restricted sense, GIFT generated routines are sparse, but the storage of a full matrix is still required. It is assumed that the pivot element is located on the diagonal and no row or column interchanges are performed, so GIFT generated routines may become unstable if the matrices are not diagonally dominant. These routines must decompose the matrix for each right-hand side in a set of linear equations. GIFT writes out (in Fortran code) the sequence of Gaussian elimination and backsubstitution steps without any do loop constructions on the matrix  $A(i, j)$ . As a result, the routines generated by GIFT can be quite large. For the 489 isotope network discussed by Timmes (1999), GIFT generated  $\sim 5.0 \times 10^7$  lines of code! Fortunately, for small reaction networks (less than about 30 isotopes), GIFT generated routines are much smaller and generally faster than other linear algebra packages.

The FLASH runtime parameter `algebra` controls which linear algebra package is used in the simulation. `algebra = 1` is the default choice and invokes the sparse matrix MA28 package. `algebra = 2` invokes the GIFT linear algebra routines.

### 18.1.2.2 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

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \sum_{i=1}^4 b_i \Delta_i, \quad (18.10)$$

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

$$(\tilde{\mathbf{I}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_1 = \mathbf{f}(\mathbf{y}^n) \quad (18.11)$$

$$(\tilde{\mathbf{I}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_2 = \mathbf{f}(\mathbf{y}^n + a_{21}\Delta_1) + c_{21}\Delta_1/h \quad (18.12)$$

$$(\tilde{\mathbf{I}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_3 = \mathbf{f}(\mathbf{y}^n + a_{31}\Delta_1 + a_{32}\Delta_2) + (c_{31}\Delta_1 + c_{32}\Delta_2)/h \quad (18.13)$$

$$(\tilde{\mathbf{I}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_4 = \mathbf{f}(\mathbf{y}^n + a_{41}\Delta_1 + a_{42}\Delta_2) + (c_{41}\Delta_1 + c_{42}\Delta_2 + c_{43}\Delta_3)/h. \quad (18.14)$$

$b_i$ ,  $\gamma$ ,  $a_{ij}$ , 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  $\Delta_3 \dots$  depends on  $\Delta_1$ ). Not all of the right-hand sides are known in advance. This general feature of higher-order integration methods impacts the optimal choice of a linear algebra package. The fourth-order Kaps-Rentrop routine in FLASH makes use of the routine GRK4T given by Kaps & Rentrop (1979).

Another time integration method used by FLASH for evolving the reaction networks is the variable order Bader-Deuflhard method (*e.g.*, Bader & Deuflhard 1983). The reaction network is advanced over a large timestep  $H$  from  $\mathbf{y}^n$  to  $\mathbf{y}^{n+1}$  by the following sequence of matrix equations. First,

$$\begin{aligned} h &= H/m \\ (\tilde{\mathbf{I}} - \tilde{\mathbf{J}}) \cdot \Delta_0 &= h\mathbf{f}(\mathbf{y}^n) \\ \mathbf{y}_1 &= \mathbf{y}^n + \Delta_0. \end{aligned} \quad (18.15)$$

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

$$\begin{aligned} (\tilde{\mathbf{I}} - \tilde{\mathbf{J}}) \cdot \mathbf{x} &= h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1} \\ \Delta_k &= \Delta_{k-1} + 2\mathbf{x} \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k, \end{aligned} \quad (18.16)$$

and closure is obtained by the last stage

$$\begin{aligned} (\tilde{\mathbf{I}} - \tilde{\mathbf{J}}) \cdot \Delta_m &= h[\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}] \\ \mathbf{y}^{n+1} &= \mathbf{y}_m + \Delta_m. \end{aligned} \quad (18.17)$$

This staged sequence of matrix equations is executed at least twice with  $m = 2$  and  $m = 6$ , yielding a fifth-order method. The sequence may be executed a maximum of seven times, which yields a fifteenth-order method. The exact number of times the staged sequence is executed depends on the accuracy requirements (set to one part in  $10^6$  in FLASH) and the smoothness of the solution. Estimates of the accuracy of an integration step are made by comparing the solutions derived from different orders. The minimum cost of this method — which applies for a single timestep that met or exceeded the specified integration accuracy — is one Jacobian evaluation, eight evaluations of the right-hand side, two matrix decompositions, and ten backsubstitutions. This minimum cost can be increased at a rate of one decomposition (the expensive part) and  $m$  backsubstitutions (the inexpensive part) for every increase in the order  $2k+1$ . The cost of increasing the order is compensated for, hopefully, by being able to take correspondingly larger (but accurate) timestep. The controls for order versus step size are a built-in part of the Bader-Deuflhard method. The cost per step of this integration method is at least twice as large as the cost per step of either a traditional first-order

accurate Euler method or the fourth-order accurate Kaps-Rentrop discussed above. However, if the Bader-Deuffhard 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 18.15 – 18.17, not all of the right-hand sides are known in advance, since the sequence of linear equations is staged. This staging feature of the integration method may make some matrix packages, such as MA28, a more efficient choice.

The FLASH runtime parameter `odeStepper` controls which integration method is used in the simulation. The choice `odeStepper = 1` is the default and invokes the variable order Bader-Deuffhard scheme. The choice `odeStepper = 2` invokes the fourth order Kaps-Rentrop / Rosenbrock scheme.

### 18.1.3 Detecting shocks

For most astrophysical detonations, the shock structure is so thin that there is insufficient time for burning to take place within the shock. However, since numerical shock structures tend to be much wider than their physical counterparts, it is possible for a significant amount of burning to occur within the shock. Allowing this to happen can lead to unphysical results. The burner unit includes a multidimensional shock detection algorithm that can be used to prevent burning in shocks. If the `useShockBurn` parameter is set to `.false.`, this algorithm is used to detect shocks in the Burn unit and to switch off the burning in shocked cells.

Currently, the shock detection algorithm supports Cartesian and 2-dimensional cylindrical coordinates. The basic algorithm is to compare the jump in pressure in the direction of compression (determined by looking at the velocity field) with a shock parameter (typically 1/3). If the total velocity divergence is negative and the relative pressure jump across the compression front is larger than the shock parameter, then a cell is considered to be within a shock.

This computation is done on a block by block basis. It is important that the velocity and pressure variables have up-to-date guard cells, so a guard cell call is done for the burners only if we are detecting shocks (*i.e.* `useShockBurning = .false.`).

### 18.1.4 Energy generation rates and reaction rates

The instantaneous energy generation rate is given by the sum

$$\dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{dY_i}{dt} . \quad (18.18)$$

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

$$\dot{\epsilon}_{\text{nuc}} = N_A \sum_i \frac{\Delta Y_i}{\Delta t} . \quad (18.19)$$

In this case, the nuclear reaction network does need to be evolved. The energy generation rate, after subtraction of any neutrino losses, is returned to the FLASH program for use with the operator splitting technique.

The tabulation of Caughlan & Fowler (1988) is used in FLASH for most of the key nuclear reaction rates. Modern values for some of the reaction rates were taken from the reaction rate library of Hoffman (2001, priv. comm.). A user can choose between two reaction rate evaluations in FLASH. The runtime parameter `useBurnTable` controls which reaction rate evaluation method is used in the simulation. The choice `useBurnTable = 0` is the default and evaluates the reaction rates from analytical expressions. The choice `useBurnTable = 1` evaluates the 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.

### 18.1.5 Temperature-based timestep limiting

When using explicit hydrodynamics methods, a timestep limiter must be used to ensure the stability of the numerical solution. The standard CFL limiter is always used when an explicit hydrodynamics unit is included in FLASH. This constraint does not allow any information to travel more than one computational cell per timestep. When coupling burning with the hydrodynamics, the CFL timestep may be so large compared to the burning timescales that the nuclear energy release in a cell may exceed the existing internal energy in that cell. When this happens, the two operations (hydrodynamics and nuclear burning) become decoupled.

To limit the timestep when burning is performed, an additional constraint is imposed. The limiter tries to force the energy generation from burning to be smaller than the internal energy in a cell. The runtime parameter `enucDtFactor` controls this ratio. The timestep limiter is calculated as

$$\Delta t_{\text{burn}} = \text{enucDtFactor} \cdot \frac{E_{\text{int}}}{E_{\text{nuc}}} \quad (18.20)$$

where  $E_{\text{nuc}}$  is the nuclear energy, expressed as energy per volume per time, and  $E_{\text{int}}$  is the internal energy per volume. For good coupling between the hydrodynamics and burning, `enucDtFactor` should be  $< 1$ . The default value is kept artificially high so that in most simulations the time limiting due to burning is turned off. Care must be exercised in the use of this routine.

## 18.2 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 `Ionize`, in particular the implementation `physics/sourceTerms/Ionize/IonizeMain/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

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \quad (18.21)$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla P = \rho \mathbf{g} \quad (18.22)$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P) \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} \quad [+ S] \quad (18.23)$$

$$\frac{\partial n_i^Z}{\partial t} + \nabla \cdot (n_i^Z \mathbf{v}) = R_i^Z \quad (i = 0, \dots, Z), \quad (18.24)$$

where  $\rho$  is the fluid density,  $t$  is the time,  $\mathbf{v}$  is the fluid velocity,  $P$  is the pressure,  $E$  is the sum of the internal energy and kinetic energy per unit mass,  $\mathbf{g}$  is the acceleration due to gravity,  $n_i^Z$  is the number density of ions of ionization level  $i$  of the element  $Z$ , and

$$R_i^Z = N_e [n_{i+1}^Z \alpha_{i+1}^Z + n_{i-1}^Z S_{i-1}^Z - n_i^Z (\alpha_i^Z + S_i^Z)], \quad (18.25)$$

Table 18.1: Runtime parameters used with the `Ionize` unit.

Variable	Type	Default	Description
<code>tneimin</code>	real	$1.0 \times 10^4$	Min nei temperature
<code>tneimax</code>	real	$1.0 \times 10^7$	Max nei temperature
<code>dneimin</code>	real	1.0	Min nei electron number density
<code>dneimax</code>	real	$1.0 \times 10^{12}$	Max nei electron number density

where  $N_e$  is the electron number density,  $\alpha_i^Z \equiv \alpha(N_e, T)$  are the coefficients of collisional and dielectronic recombination, and  $S_i^Z \equiv S(N_e, T)$  are the collisional ionization coefficients of Summers(1974). Note that `NSPECIES`, the total number of FLASH species, will be given by

$$N_{\text{spec}} = 2 + \sum_Z (Z + 1);$$

the sum ranges over all the elements from the list above that are included in the problem, and the additional 2 comes from the hydrogen and electron mass fractions which are automatically included by the `IonizeMain` subunit.

### 18.2.1 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 (18.21) are solved using the FLASH hydrodynamics solver with  $R_i^Z = 0$ . After each transport step, the “stiff” systems of ordinary differential equations (one system per element included in the simulation) for the NEI problem

$$\frac{\partial n_i^Z}{\partial t} = R_i^Z \quad (i = 0, \dots, Z) \quad (18.26)$$

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-Deuffhard 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 for the similar problem of nuclear burning, see Section 18.1.

Note that in the present version, the contribution of the ionization and recombination to the energy equation (the bracketed term in (18.23)) 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.

### 18.2.2 Usage

In order to run a FLASH executable that uses the ionization unit, the ionization coefficients of Summers (1974) must be contained in a file named `summers.den_1e8.rates` in the same directory as the executable when the simulation is run. This file is copied into the `object/` directory with the `Config` keyword `DATAFILES` in the `physics/sourceTerms/Ionize/IonizeMain` implementation.

The `Ionize` unit supplies the runtime parameters described in Table 18.1. There are two implementations of `physics/sourceTerms/Ionize/IonizeMain`: the default implementation, `Nei` (tested using `Neitest` (see Section 35.7.1)), and `Eqi` (untested in FLASH4). The former computes ion species for non-equilibrium ionization, and the latter computes ion species in the approximation of ionization equilibrium.

The `Ionize` unit requires that the subunit implementation `Simulation/SimulationComposition/-Ionize` be used to set up the ion species of the fluid. The ions are defined in a file `Simulation/-SimulationComposition/Ionize/SpeciesList.txt`, however, the `Config` file in the simulation directory (e.g. `Simulation/SimulationMain/Neitest/Config`) defines which subset of these elements are to be used.

## 18.3 Stir Unit

The addition of driving terms in a hydrodynamical simulation can be a useful feature, for example, in generating turbulent flows or for simulating the addition of power on larger scales (*e.g.*, supernova feedback into the interstellar medium). The **Stir** unit comes in two implementations: 1) the **Generate** implementation, in which a divergence-free, random time-correlated ‘stirring’ velocity is directly added at selected modes in the simulation and 2) the **FromFile** implementation, in which a stirring field is set up from data residing on a file. The **FromFile** implementation allows to set up identical stirring fields on different platforms, and thus comparisons can be made between different codes.

Before FLASH 4.2, the implementation now called **Generate** was the only one provided. It is still the default that is being used if one specifies just `REQUIRES physics/sourceTerms/Stir` in a **Config** file or `-unit=physics/sourceTerms/Stir` on the **setup** command line.

### 18.3.1 Stir Unit: Generate Implementation

In the generate implementation, the Stir unit directly adds a divergence-free, time-correlated ‘stirring’ velocity at selected modes in the simulation.

The time-correlation is important for modeling realistic driving forces. Most large-scale driving forces are time-correlated, rather than white-noise; for instance, turbulent stirring from larger scales will be correlated on timescales related to the lifetime of an eddy on the scale of the simulation domain. This time correlation will lead to coherent structures in the simulation that will be absent with white-noise driving.

For each mode at each timestep, six separate phases (real and imaginary in each of the three spatial dimensions) are evolved by an Ornstein-Uhlenbeck (OU) random process (Uhlenbeck 1930). The OU process is a zero-mean, constant-rms process, which at each step ‘decays’ the previous value by an exponential  $f = e^{(\frac{\Delta t}{\tau})}$  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_energy`) divided by the decay time  $\tau$  (`st_decay`). In the limit that the timestep  $\Delta t \rightarrow 0$ , it is easily seen that the algorithm represents a linearly-weighted summation of the old state with the new Gaussian random variable.

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

The velocities are then converted to physical space by a direct Fourier transform – *i.e.*, adding the sin and cos terms explicitly. Since most drivings involve a fairly small number of modes, this is more efficient than an FFT, since the FFT would need large numbers of modes (equal to six times the number of cells in the domain), the vast majority of which would have zero amplitude.

### 18.3.2 Stir Unit: FromFile Implementation

In the from file implementation, the Stir unit sets up a stirring field from data residing on a file. Here we summarize the method for driving turbulence used in Federrath et al. (2010, A&A, 512, A81). Please refer to that paper for further details.

Turbulence decays in about a crossing time, because the kinetic energy carried by the turbulence dissipates on small scales and turns into heat. In order to study the statistics of turbulence (*e.g.*, the PDF, power spectrum, structure functions, etc.) over a significant time period thus requires continuous stirring (also called driving or forcing) with a turbulent acceleration field, which we call  $\vec{f}(\vec{x}, t)$  in the following.

The stirring field  $\vec{f}$  is often modeled with a spatially static pattern for which the amplitude is adjusted in time. This results in a roughly constant energy input on large scales, but has the disadvantage that the turbulence is not really random, because the large-scale pattern is fixed, which may introduce undesirable systematics. Other studies model  $\vec{f}$  such that it can vary in time *and* space to achieve a smoothly varying pattern that resembles the flow of kinetic energy from scales larger than the simulation box scale. The most widely used method to achieve this is the Ornstein-Uhlenbeck (OU) process. The OU process is a well-defined stochastic process with a finite autocorrelation timescale. It can be used to excite turbulent motions in 3D, 2D, or 1D simulations as explained in Eswaran & Pope (1988, Computers & Fluids, 16, 257).

The OU process is a stochastic differential equation describing the evolution of  $\widehat{\vec{f}}$  in Fourier space ( $k$ -space):

$$d\widehat{\vec{f}}(\vec{k}, t) = f_0(\vec{k}) \underline{\mathcal{P}}^\zeta(\vec{k}) d\vec{W}(t) - \widehat{\vec{f}}(\vec{k}, t) \frac{dt}{T}. \quad (18.27)$$

The first term on the right hand side is a diffusion term. This term is modeled by a Wiener process  $\vec{W}(t)$ , which adds a Gaussian random increment to the vector field given in the previous time step  $dt$ . Wiener processes are random processes, such that

$$\vec{W}(t) - \vec{W}(t - dt) = \vec{\mathcal{N}}(0, dt), \quad (18.28)$$

where  $\vec{\mathcal{N}}(0, dt)$  denotes the 3D, 2D, or 1D version of a Gaussian distribution with zero mean and standard deviation  $dt$ . This is combined with a projection using the projection tensor  $\underline{\mathcal{P}}^\zeta(\vec{k})$  in Fourier space. In index notation, the projection operator reads

$$\mathcal{P}_{ij}^\zeta(\vec{k}) = \zeta \mathcal{P}_{ij}^\perp(\vec{k}) + (1 - \zeta) \mathcal{P}_{ij}^\parallel(\vec{k}) = \zeta \delta_{ij} + (1 - 2\zeta) \frac{k_i k_j}{|\vec{k}|^2}, \quad (18.29)$$

where  $\delta_{ij}$  is the Kronecker symbol, and  $\mathcal{P}_{ij}^\perp = \delta_{ij} - k_i k_j / k^2$  and  $\mathcal{P}_{ij}^\parallel = k_i k_j / k^2$  are the solenoidal (divergence-free) and the compressive (curl-free) projection operators, respectively. The projection operator serves to construct a purely solenoidal stirring field by setting  $\zeta = 1$ . For  $\zeta = 0$ , a purely compressive stirring field is obtained. Any combination of solenoidal and compressive modes can be constructed by choosing  $\zeta \in [0, 1]$ . By changing the parameter  $\zeta$ , we can thus set the power of compressive modes with respect to the total power in the driving field. The analytical ratio of compressive power to total power can be derived from equation (18.29) by evaluating the norm of the compressive component of the projection tensor,

$$\left| (1 - \zeta) \mathcal{P}_{ij}^\parallel \right|^2 = (1 - \zeta)^2, \quad (18.30)$$

and by evaluating the norm of the full projection tensor

$$\left| \mathcal{P}_{ij}^\zeta \right|^2 = 1 - 2\zeta + D\zeta^2. \quad (18.31)$$

The result of the last equation depends on the dimensionality  $D = 1, 2, 3$  of the simulation, because the norm of the Kronecker symbol  $|\delta_{ij}| = 1, 2$  or  $3$  in one, two or three dimensions, respectively. The ratio of equations (18.30) and (18.31) gives the relative power in compressive modes,  $F_{\text{long}}/F_{\text{tot}}$ , as a function of  $\zeta$ :

$$\frac{F_{\text{long}}}{F_{\text{tot}}} = \frac{(1 - \zeta)^2}{1 - 2\zeta + D\zeta^2}. \quad (18.32)$$

Figure 18.3 provides a graphical representation of this ratio for the 1D, 2D, and 3D case. For comparison, we plot numerical values of the forcing ratio obtained in eleven 3D and 2D hydrodynamical simulations by Federrath et al. (2010, A&A, 512, A81), in which we varied the forcing parameter  $\zeta$  from purely compressive stirring ( $\zeta = 0$ ) to purely solenoidal stirring ( $\zeta = 1$ ) in the range  $\zeta = [0, 1]$ , separated by  $\Delta\zeta = 0.1$ . Note that a natural mixture of stirring modes is obtained for  $\zeta = 0.5$ , which leads to  $F_{\text{long}}/F_{\text{tot}} = 1/3$  for 3D turbulence, and  $F_{\text{long}}/F_{\text{tot}} = 1/2$  for 2D turbulence. A simple way to understand this natural ratio is to consider longitudinal and transverse waves. In 3D, the longitudinal waves occupy one of the three spatial dimensions, while the transverse waves occupy two of the three on average. Thus, the longitudinal (compressive) part has a relative power of  $1/3$ , while the transverse (solenoidal) part has a relative power of  $2/3$  in 3D. In 2D, the natural ratio is  $1/2$ , because longitudinal and transverse waves are evenly distributed in two dimensions.

The second term on the right-hand side of equation (18.27) is a drift term describing the exponential decay of the autocorrelation of  $\vec{f}$ . The usual procedure is to set the autocorrelation timescale equal to the turbulent crossing time,  $T = L_{\text{peak}}/V$ , on the scale of energy injection,  $L_{\text{peak}}$ . This type of stirring models the kinetic energy input from large-scale turbulent fluctuations breaking up into smaller and smaller structures.

The runtime parameters associated with the `StirFromFile` unit are described in the 18.3.3 section.



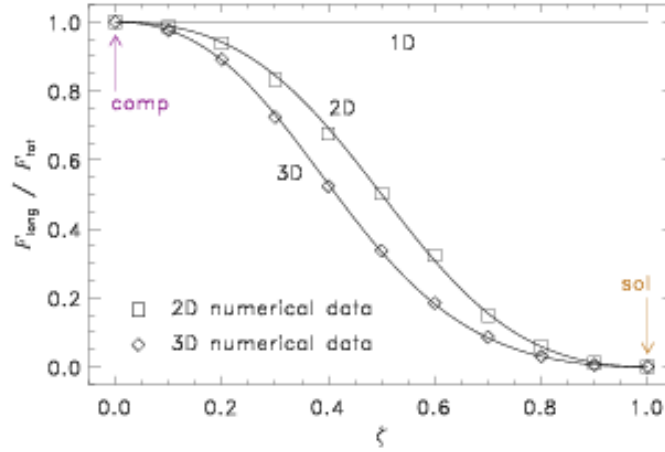


Figure 18.3: Ratio of compressive to total power of the turbulent stirring field, reprinted from Federrath et al. (2010, A&A, 512, A81) with permission by Astronomy & Astrophysics. The solid lines labelled with 1D, 2D, and 3D show the analytical expectation for this ratio, equation (18.32), as a function of the stirring parameter  $\zeta$  for one-, two- and three-dimensional driving, respectively. The diamonds and squares show results of numerical simulations in 3D and 2D with  $\zeta = [0, 1]$ , separated by  $\Delta\zeta = 0.1$ . The two limiting cases of purely solenoidal stirring ( $\zeta = 1$ ) and purely compressive stirring ( $\zeta = 0$ ) are indicated as "sol" and "comp", respectively. Note that in any 1D model, all power is in the compressive component, and thus  $F_{\text{long}}/F_{\text{tot}} = 1$ , independent of  $\zeta$ .

### 18.3.3 Using the StirFromFile Unit

#### 18.3.3.1 Runtime Parameters

Table 18.2: Runtime parameters for the stirring module.

Variable	Type	Default	Description
<code>useStir</code>	boolean	<code>.true.</code>	switch stirring on/off
<code>st_computeDt</code>	boolean	<code>.false.</code>	restrict timestep based on stirring
<code>st_infilename</code>	string	<code>"forcingfile.dat"</code>	file containing the stirring time sequence

Table 18.2 lists the runtime parameters for the StirFromFile unit. This includes a switch for turning the stirring module on/off and a switch to restrict the timestep based on the acceleration field used for stirring (`st_computeDt` is switched off by default, because it is normally sufficient to restrict the timestep based on the gas velocity). Finally, `st_infilename` is the name of the input file containing the time and mode sequence used for stirring. This file must be prepared in advance with a separate Fortran program located in `SimulationMain/StirFromFile/forcing_generator/`. The reason for this structural splitting is to predetermine what the code is going to do. For instance, by preparing the time sequence of the stirring in advance, one can always reproduce exactly the full evolution of all driving patterns applied during a simulation. It also has the advantage that exactly identical stirring patterns can be applied in completely different codes, because they read the time and mode sequence from the same stirring file (Price & Federrath, 2010, MNRAS, 406, 1659).

The stirring module is compatible with any hydro or MHD solver and any grid implementation (uniform or AMR). Upon inclusion in a FLASH setup or module, the `StirFromFile` module defines three additional grid scalar fields, `accx`, `accy`, and `accz`, holding the three vector components of the stirring field  $\vec{f}$ .



18.3.3.2 Preparing the Stirring Sequence (`st_infilename`)Table 18.3: Parameters in `forcing_generator.F90` to prepare a stirring sequence.

Variable	Type	Default	Description
<code>ndim</code>	integer	3	The dimensionality of the simulation (1, 2, or 3)
<code>xmin, xmax</code>	real	-0.5, 0.5	Domain boundary coordinates in $x$ direction
<code>ymin, ymax</code>	real	-0.5, 0.5	Domain boundary coordinates in $y$ direction
<code>zmin, zmax</code>	real	-0.5, 0.5	Domain boundary coordinates in $z$ direction
<code>st_spectform</code>	integer	1	Spectral shape (0: band, 1: paraboloid)
<code>st_decay</code>	real	0.5	Autocorrelation time of the OU process, $T = L_{\text{peak}}/V$
<code>st_energy</code>	real	2e-3	Determines the driving amplitude
<code>st_stirmin</code>	real	6.283	Minimum wavenumber stirred (e.g., $k_{\text{min}} \lesssim 2\pi/L_{\text{box}}$ )
<code>st_stirmax</code>	real	18.95	Maximum wavenumber stirred (e.g., $k_{\text{max}} \gtrsim 6\pi/L_{\text{box}}$ )
<code>st_solweight</code>	real	1.0	Mode mixture $\zeta = [0, 1]$ in Eq. (18.32). Typical values are 1.0: solenoidal; 0.0: compressive; 0.5: natural mixture.
<code>st_seed</code>	integer	140281	Random seed for stirring sequence
<code>end_time</code>	real	5.0	Final time in stirring sequence
<code>nsteps</code>	integer	100	Number of realizations between $t = 0$ and <code>end_time</code>
<code>outfilename</code>	string	"forcingfile.dat"	Output name (input file <code>st_infilename</code> for FLASH)

The code requires a time sequence of stirring modes at runtime, which have to be prepared with the stand-alone Fortran program `forcing_generator.F90` in `SimulationMain/StirFromFile/forcing_generator/`. A Makefile is provided in the same directory. This program prepares the time sequence of Fourier modes, which is then read by FLASH during runtime, to construct the physical acceleration fields used for stirring. It controls the spatial structure and the temporal correlation of the driving, its amplitude, the mode mixture, and the time separation between successive driving patterns. The user has to modify `forcing_generator.F90` to construct a requested driving sequence and to tailor it to the desired physical situation to be modeled.

Table 18.3 lists all the parameters that can be adjusted in the main routine of `forcing_generator.F90`. Most of them are straightforward to set (`ndim, xmin, xmax, ymin, ...`<sup>1</sup>), but others may require some explanation. For example, `st_spectform` determines the shape of the driving amplitude in Fourier space. Many colleagues drive a band (`st_spectform=0`), i.e., equal power injected between wavenumber modes  $k_{\text{min}} = \text{st\_stirmin}$  and  $k_{\text{max}} = \text{st\_stirmax}$ . This produces a sharp transition between stirred modes and modes that are not stirred. Here we set the default to a smooth function, a paraboloid (`st_spectform=1`), such that most power is injected on wavenumber  $k_{\text{peak}} = (k_{\text{min}} + k_{\text{max}})/2$  and falls off quadratically towards both wavenumber ends, normalized such that the injected power at  $k_{\text{min}}$  and  $k_{\text{max}}$  vanishes. This has the advantage of defining a characteristic peak injection scale  $k_{\text{peak}}$  and achieves a smooth transition between stirred and non-stirred wavenumbers.

`st_decay` and `st_energy` determine the autocorrelation time of the OU process and the total injected energy, which is simply a measure for the normalization of the acceleration field. These parameters must be adjusted according to the physical setup. For instance, for a given target velocity dispersion  $V$  on the injection scale  $L_{\text{peak}} = 2\pi/k_{\text{peak}}$ , the autocorrelation time should be set equal to the turbulent crossing time,  $T = L_{\text{peak}}/V$ . In contrast, setting `st_decay` to a very small or a very large number results in white noise driving or in a static driving pattern, respectively.

The parameter `st_solweight` determines whether the acceleration field will be solenoidal (divergence-free) or compressive (curl-free) or any mixture, according to Equation (18.32). Incompressible gases should naturally be driven with a purely solenoidal field ( $\zeta = 1$ ), while compressible turbulence in the interstellar medium may be driven primarily by a mixture of solenoidal and compressive modes. A detailed study of the influence of  $\zeta$  is presented in Federrath et al. (2010, A&A, 512, A81).

<sup>1</sup>Note that we typically assume a cubic box with side length  $L_{\text{box}} = \text{xmax} - \text{xmin} = \text{ymax} - \text{ymin} = \text{zmax} - \text{zmin}$

`st_seed` is the random seed for the OU sequence and determines the pseudo random number sequence for the integrated Box-Muller random number generator.

Finally, `end_time` and `nsteps` determine the final physical time for stirring and the number of driving patterns to be prepared within the time period from  $t = 0$  to  $t = \text{end\_time}$ . This sets the number of equally-spaced times at which FLASH is going to read a new stirring pattern from the file. This allows the user to control how frequently a new driving pattern is constructed. A useful time separation of successive driving patterns is about 10% of a crossing time (or autocorrelation time), i.e., setting `nsteps` =  $10 \times \text{end\_time}/\text{st\_decay}$ . This will sample the smooth changes in the OU driving sequence sufficiently well for most applications.

### 18.3.4 Stirring Unit Test

An example setup using the `StirFromFile` unit is located in `SimulationMain/StirFromFile/`. The unit test can be invoked by

```
./setup StirFromFile -auto -3d -nxb=16 -nyb=16 -nzb=16 +ug -with-unit=physics/Hydro.
```

The FLASH executable must be copied into the run directory together with the standard `flash.par` for this setup, and together with the default forcing file (to be constructed using the standard parameters; see section 18.3.3.2). During runtime the code writes a file with the time evolution of spatially integrated quantities, amongst others, the rms Mach number and vorticity, which can be used as basic code checks.

## 18.4 Energy Deposition Unit

The `Energy Deposition` unit calculates the energy deposited by one or more laser beams incident upon the simulation domain. The function `EnergyDeposition` is the main control driver of the `Energy Deposition` unit. The current implementation treats the laser beams in the geometric optics approximation. Beams are made of a number of rays whose paths are traced through the domain based on the local refractive index of each cell. The laser power deposited in a cell is calculated based on the inverse Bremsstrahlung power in the cell and depends on the local electron number density gradient and local electron temperature gradient. Currently there are two schemes implemented into FLASH: 1) a ray tracing algorithm based on cell average quantities and 2) a more refined ray tracing method based on cubic interpolation. Both schemes are discussed in the algorithmic implementation section 18.4.4 below.

### 18.4.1 Ray Tracing in the Geometric Optics Limit

In the geometric optics approach, the path of a laser wave can be described as the motion of a ray of unit-mass through the potential field

$$V(\mathbf{r}) = \frac{c^2}{2}\eta(\mathbf{r})^2, \quad (18.33)$$

where  $c$  is the speed of light in vacuum and  $\eta$  is the index of refraction of the medium, assumed to vary on a much longer spatial scale than the wavelength of the laser wave. Also  $\eta(\mathbf{r})$  is considered constant during the time it takes for the ray to cross the domain (frozen medium). However  $\eta(\mathbf{r})$  is allowed to vary from one time step to the next. For a non-relativistic unmagnetized plasma, the refractive index is given by

$$\eta(\mathbf{r})^2 = 1 - \frac{\omega_p^2(\mathbf{r})}{\omega^2} = 1 - \frac{n_e(\mathbf{r})}{n_c}, \quad (18.34)$$

where  $\omega_p(\mathbf{r})$  is the plasma frequency,  $\omega$  the laser frequency,  $n_e(\mathbf{r})$  is the electron number density at location  $\mathbf{r}$  and

$$n_c = \frac{m_e \pi \omega^2}{e^2} = \frac{m_e \pi c^2}{\lambda^2 e^2} \quad (18.35)$$

is the critical density at which the ray frequency and the plasma frequency are equal ( $m_e$  and  $e$  are the electron mass and charge,  $\lambda$  is the laser wavelength). The ray equation of motion is then

$$\frac{d^2 \mathbf{r}}{dt^2} = \nabla \left( -\frac{c^2}{2} \frac{n_e(\mathbf{r})}{n_c} \right), \quad (18.36)$$

which constitutes the basic ray tracing 2nd order ODE equation. Splitting into two 1st order ODEs leads to

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} \quad (18.37)$$

$$\frac{d\mathbf{v}}{dt} = -\frac{c^2 \nabla n_e(\mathbf{r})}{2n_c}. \quad (18.38)$$

For short distances we can assume a first order Taylor expansion of the electron number density around a specific location  $\mathbf{r}_0$ :

$$n_e(\mathbf{r}) \approx n_e(\mathbf{r}_0) + \nabla n_e(\mathbf{r}_0) \cdot (\mathbf{r} - \mathbf{r}_0), \quad (18.39)$$

where  $\nabla n_e(\mathbf{r}_0)$  is the electron number density gradient vector at  $\mathbf{r}_0$ . A consequence of this short distance approximation is that the electron number density gradient vectors at  $\mathbf{r}$  and  $\mathbf{r}_0$  are equal. For separable 1st order ODEs (cartesian) we can write the following equations for the ray velocity and position as a function of time

$$\mathbf{v}(t) = \mathbf{v}_0 - \frac{c^2}{2n_c} \nabla n_e(\mathbf{r}_0) t, \quad (18.40)$$

$$\mathbf{r}(t) = \mathbf{r}_0 + \mathbf{v}_0 t - \frac{c^2}{4n_c} \nabla n_e(\mathbf{r}_0) t^2, \quad (18.41)$$

where  $\mathbf{r}_0$  and  $\mathbf{v}_0$  are the initial ray position and velocity. For non-separable 1st order ODEs, like for example cylindrical or spherical coordinates (see below), we cannot state analytical equations for the ray velocity and position as a function of time, and we must solve the set of 1st order ODEs using specific numerical methods like for example Runge-Kutta (RK).

### 18.4.2 Laser Power Deposition

The power  $P$  of an electromagnetic wave is depleted by the inverse Bremsstrahlung (ib) process. The rate of power loss is governed by a 1st order ordinary differential equation (ODE):

$$\frac{dP}{dt} = -\nu_{ib}(t)P. \quad (18.42)$$

The inverse Bremsstrahlung frequency factor is given by the formula

$$\nu_{ib} = \frac{n_e}{n_c} \nu_{ei}, \quad (18.43)$$

where  $\nu_{ei}$  is the electron-ion collision frequency

$$\nu_{ei} = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_e Z e^4 \ln \Lambda}{(k_B T_e)^{3/2}}. \quad (18.44)$$

Here  $m_e$  is the electron mass,  $Z$  is the average ionization number of the plasma,  $e$  is the electron charge,  $\ln \Lambda$  is the Coulomb logarithm,  $k_B$  is the Boltzmann constant and  $T_e$  is the electron temperature. The Coulomb logarithm is the natural logarithm of the Debye number  $\Lambda$  and is taken here as

$$\ln \Lambda = \ln \left[ \frac{3}{2Ze^3} \left( \frac{k_B^3 T_e^3}{\pi n_e} \right)^{1/2} \right]. \quad (18.45)$$

The inverse Bremsstrahlung frequency depends thus on the electron number density and the electron temperature, both of which are functions of the position, and, since the position changes with time, it ultimately is also a function of time

$$\nu_{ib}(\mathbf{r}) = \nu_{ib}(t) = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{Ze^4}{n_e k_B^{3/2}} \frac{n_e[\mathbf{r}(t)]^2 \ln \Lambda[\mathbf{r}(t)]}{T_e[\mathbf{r}(t)]^{3/2}}. \quad (18.46)$$

Solution of the above ODE in Eq.(18.42) gives the attenuation of the ray's power from time zero to time  $t$ :

$$P_t = P_0 \exp \left\{ - \int_0^t \nu_{ib}(t') dt' \right\}. \quad (18.47)$$

For a given small time step, the integral in Eq.(18.47) can be approximated in the following way. In order to remove the intermediate function  $\mathbf{r}(t)$  from the expression in Eq.(18.46), we first assume that the Coulomb logarithm remains constant during the incremental time step

$$\ln \Lambda[\mathbf{r}(t)] \approx \ln \Lambda[\mathbf{r}_0] = \ln \left[ \frac{3}{2Ze^3} \left( \frac{k^3 T_e(\mathbf{r}_0)^3}{\pi n_e(\mathbf{r}_0)} \right)^{1/2} \right]. \quad (18.48)$$

Using first order Taylor expansions on both  $n_e$  and  $T_e$  similar to equation 18.39

$$n_e[\mathbf{r}(t)] = n_e(\mathbf{r}_0) + \nabla n_e(\mathbf{r}_0) \cdot (\mathbf{r}(t) - \mathbf{r}_0) \quad (18.49)$$

$$T_e[\mathbf{r}(t)] = T_e(\mathbf{r}_0) + \nabla T_e(\mathbf{r}_0) \cdot (\mathbf{r}(t) - \mathbf{r}_0) \quad (18.50)$$

and inserting the ray position equation 18.41, we get

$$n_e[\mathbf{r}(t)] = n_e(\mathbf{r}_0) \left[ 1 + \frac{\nabla n_e(\mathbf{r}_0) \cdot \mathbf{v}_0}{n_e(\mathbf{r}_0)} t - \frac{c^2 \nabla n_e(\mathbf{r}_0) \cdot \nabla n_e(\mathbf{r}_0)}{4n_e n_e(\mathbf{r}_0)} t^2 \right] \quad (18.51)$$

$$T_e[\mathbf{r}(t)] = T_e(\mathbf{r}_0) \left[ 1 + \frac{\nabla T_e(\mathbf{r}_0) \cdot \mathbf{v}_0}{T_e(\mathbf{r}_0)} t - \frac{c^2 \nabla T_e(\mathbf{r}_0) \cdot \nabla n_e(\mathbf{r}_0)}{4n_e T_e(\mathbf{r}_0)} t^2 \right]. \quad (18.52)$$

The inverse-Bremsstrahlung rate can thus be written as a rational polynomial expression

$$\nu_{ib}(t) = \nu_{ib}(0) \frac{(1 + Ut + Rt^2)^2}{(1 + Wt + St^2)^{3/2}}, \quad (18.53)$$

where  $\nu_{ib,0}$  is the inverse-Bremsstrahlung rate at the initial point (zero time)

$$\nu_{ib}(0) = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{Ze^4 \ln \Lambda[\mathbf{r}_0]}{n_e k_B^{3/2}} \frac{n_e(\mathbf{r}_0)^2}{T_e(\mathbf{r}_0)^{3/2}} \quad (18.54)$$

and

$$U = \frac{\nabla n_e(\mathbf{r}_0) \cdot \mathbf{v}_0}{n_e(\mathbf{r}_0)} \quad (18.55)$$

$$W = \frac{\nabla T_e(\mathbf{r}_0) \cdot \mathbf{v}_0}{T_e(\mathbf{r}_0)} \quad (18.56)$$

$$R = - \frac{c^2 \nabla n_e(\mathbf{r}_0) \cdot \nabla n_e(\mathbf{r}_0)}{4n_e n_e(\mathbf{r}_0)} \quad (18.57)$$

$$S = - \frac{c^2 \nabla T_e(\mathbf{r}_0) \cdot \nabla n_e(\mathbf{r}_0)}{4n_e T_e(\mathbf{r}_0)}. \quad (18.58)$$

The integral in Eq.(18.47) can be solved by using a second order Gaussian quadrature

$$\int_0^t \nu_{ib}(t') dt' = \nu_{ib}(0) \frac{t}{2} \sum_{i=1}^2 w_i \frac{(1 + Ut_i + Rt_i^2)^2}{(1 + Wt_i + St_i^2)^{3/2}}, \quad (18.59)$$

where  $t_{1,2} = (1 \pm 1/\sqrt{3})t/2$  and both weights are equal to 1. The rate of energy deposition (energy deposited per unit time) during this time is then  $P_0 - P_t$ .

### 18.4.3 Laser Energy Density

*Implemented only for 3D Cartesian geometry and 2D cylindrical geometry with `ed_laser3Din2D` ray tracing in FLASH 4.5.*

Light carries energy through space and therefore can be attributed an energy density (energy per volume). For example, light propagating at velocity  $c$  with power  $P$  through an area  $A$  has an energy density of  $U_{light} = \frac{P}{Ac}$ .

Splitting the laser's power across a finite number of rays is an abstraction with an interesting consequence. Because each ray's power is artificially confined to a 1D curvilinear axis, light rays in FLASH are naturally described by linear energy densities (energy per distance) rather than volumetric energy densities (energy per volume). Volumetric energy density is more physically relevant. Therefore, the contribution of each ray to volumetric laser energy density (a cell quantity, "lase", available as an output when using certain ray-tracing modules – see 18.4.11.5) is calculated by taking the linear energy density of each ray, integrating over the ray path within the cell, and dividing by the cell volume.

The details of the laser energy density calculation are as follows. Trading path length for parametrized time, ray power  $P_t$  changes along the ray path by Eq.(18.47). For light traveling at speed  $c$  (determined by frequency and medium), ray linear energy density  $E_t$  similarly varies along ray path:

$$E_t = \frac{P_t}{c} \quad (18.60)$$

Volumetric energy density  $U_{cell}$  is found by integrating linear energy density  $E_t$  over the ray's path within the cell ( $t_f$  is the parametrized time value for which the ray reaches edge of cell), then dividing by cell volume  $V_{cell}$ :

$$U_{cell} = \frac{\int_0^{t_f} E_t dt}{V_{cell}} = \frac{1}{cV_{cell}} \int_0^{t_f} P_t dt \quad (18.61)$$

For each cell, all contributions to volumetric laser energy density (all rays'  $U_{cell}$ ) are summed to give a total laser energy density for that cell.

### 18.4.4 Algorithmic Implementations of the Ray Tracing

The current implementation of the laser energy deposition assumes that rays transit the entire domain in a single time step. This is equivalent of saying that the domain's reaction time (changes of its state variables) is much longer than the time it takes for each ray to either cross the domain or get absorbed by it. For each time step it is first checked, if at least one of the laser beams is still active. If this is the case, then for each active beam a collection of rays is generated possessing the right specifications according to the beam's direction of firing, its frequency, its pulse shape and its power. The Energy Deposition unit moves all rays across all blocks until every ray has either exited the domain or has been absorbed by the domain. The rays are moved on a block by block basis and for each block the rays are traced through the interior cells. The Energy Deposition unit utilizes the infrastructure from the Grid Particles unit 8.9 to move rays between blocks and processors. The code structure of the Energy Deposition unit during one time step is as follows:

```

Loop over all blocks containing rays
  Calculate needed block data for all cells.
  ———— Start threading ————
    Loop over all rays in block
      Trace each ray through all cells in block
    End loop
  ———— End threading ————

```

End loop

Reassign rays to blocks/processors and repeat (exit, if no more rays)

The inner loop over all rays in a block is conveniently handled in one subroutine to allow for compact (optional) threading. Currently there are three algorithmic options on how to trace the rays in FLASH: 1) Cell average algorithm, 2) Cubic interpolation with piecewise parabolic ray tracing and 3) Cubic interpolation using Runge Kutta integration schemes.

#### 18.4.4.1 Cell Average (AVG) Algorithm

The AVG algorithm (Kaiser 2000) is based on tracing the rays on a cell-by-cell basis. Each cell has its own average center electron number density  $\langle n_e \rangle$  and electron temperature  $\langle T_e \rangle$  value at the center of the cell. The electron number density gradient vector  $\langle \nabla n_e \rangle$  as well as the electron temperature gradient vector  $\langle \nabla T_e \rangle$  are assumed to be constant within each cell. Rays will be transported through each cell between cell faces in one step. Electron number densities  $n_{e0}$  and electron temperatures  $T_{e0}$  on the entry cell face are calculated from the cell center values in accordance with equation 18.39 using first order Taylor expansions

$$n_{e0} = \langle n_e \rangle + \langle \nabla n_e \rangle \cdot (\mathbf{r}_0 - \langle \mathbf{r} \rangle) \quad (18.62)$$

$$T_{e0} = \langle T_e \rangle + \langle \nabla T_e \rangle \cdot (\mathbf{r}_0 - \langle \mathbf{r} \rangle), \quad (18.63)$$

where  $\mathbf{r}_0$  are the position coordinates of the ray at the cell's face and  $\langle \mathbf{r} \rangle$  are the coordinates of the cell's center. The ray's cell crossing time  $t_{cross}$  is determined from the quadratic time equation 18.41 by inserting planar equations  $A_x x + A_y y + A_z z + D = \mathbf{A} \cdot \mathbf{r}(t) + D = 0$  for all cell faces (in cartesian coordinates), leading to

$$-\frac{c^2 \mathbf{A} \cdot \langle \nabla n_e \rangle}{4n_c} t^2 + \mathbf{A} \cdot \mathbf{v}_0 t + \mathbf{A} \cdot \mathbf{r}_0 + D = 0, \quad (18.64)$$

and selecting the shortest time (excluding the zero time corresponding to the point of entry). In FLASH, the cartesian cell faces are coplanar with the xy-, xz- and the yz-plane, simplifying the quadratic time equation. A cell face located at  $x_{cell}$  and coplanar with the yz-plane has the plane equation  $x = x_{cell}$  and thus  $A_x = 1$ ,  $A_y, A_z = 0$  and  $D = -x_{cell}$ . For this cell face, the quadratic time equation becomes

$$-\frac{c^2 \langle \nabla n_e \rangle_x}{4n_c} t^2 + v_{0x} t + r_{0x} - x_{cell} = 0 \quad (18.65)$$

and similar for the other cartesian components. In order to achieve a stable ray tracing algorithm and uniquely assign rays to cells and blocks while crossing through cell walls, a cell wall thickness is defined, which is shared between two adjacent cells. The code is set up such that rays are never to be found inside this wall. When crossing between two cells occurs (same block or different blocks), the rays positions are always adjusted (nudged) such that they will sit on this wall with finite thickness. The cell wall thickness is defined in the code as being a very tiny fraction (adjustable, with default value of 1:1,000,000) of the smallest cell dimension during a simulation.

The ray's rate of energy deposition between the cell entry face and the cell exit face is calculated using equations 18.47 to 18.59 with  $\nabla n_e(\mathbf{r}_0)$  and  $\nabla T_e(\mathbf{r}_0)$  replaced by their cell average values  $\langle \nabla n_e \rangle$  and  $\langle \nabla T_e \rangle$  and  $n_e(\mathbf{r}_0)$  and  $T_e(\mathbf{r}_0)$  replaced by their corresponding cell entry values  $n_{e0}$  and  $T_{e0}$ . The upper integration time limit is  $t_{cross}$ , the ray's cell crossing time.

The AVG algorithm, while conceptually simple, leads to discontinuous change in  $n_e$  at the cell interfaces. To account for this, the ray undergoes refraction according to Snell's law. Refraction at cell interfaces causes a change in the ray's velocity normal to the interface surface while preserving the ray velocity component transverse to the interface normal. To derive the change in the ray normal velocity component, imagine the cell interface to have a small infinitesimal thickness  $s$  and the ray moving from left to right through the interface. On the left and right the normal velocity components are  $v_\perp(\ell)$  and  $v_\perp(r)$ , respectively, while the

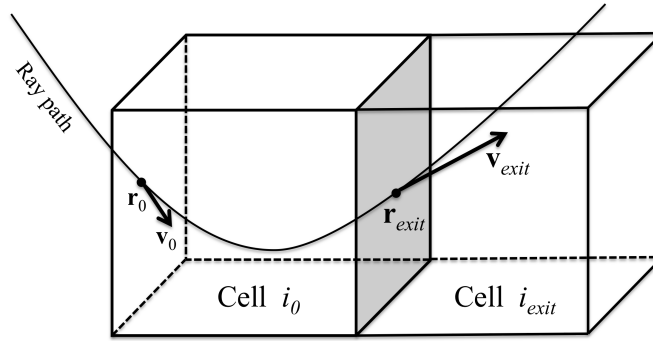


Figure 18.4: A single ray crossing a cell.

corresponding  $n_e$  are  $n_e(\ell)$  and  $n_e(r)$ . Since we are dealing with an interface of infinitesimal thickness, we can use the first order equation 18.40 to get

$$v_{\perp}(r) = v_{\perp}(\ell) - \frac{c^2}{2n_c} \frac{\Delta n_e}{s} t, \quad (18.66)$$

where  $\Delta n_e = n_e(r) - n_e(\ell)$ , and the electron number density gradient is  $\nabla n_e = \Delta n_e/s$ . But  $s/t$  is the average velocity of the ray passing through the interface and since we have constant acceleration we have  $s/t = [v_{\perp}(r) + v_{\perp}(\ell)]/2$ , which, when inserted into the last equation, gives

$$\begin{aligned} v_{\perp}(r) - v_{\perp}(\ell) &= -\frac{c^2}{n_c} \frac{\Delta n_e}{[v_{\perp}(r) + v_{\perp}(\ell)]} \\ v_{\perp}^2(r) - v_{\perp}^2(\ell) &= -\frac{c^2 \Delta n_e}{n_c} \\ \Delta v_{\perp}^2 &= -\frac{c^2 \Delta n_e}{n_c}. \end{aligned} \quad (18.67)$$

Clearly there is a limit as to how large  $\Delta n_e$  can be for refraction to occur. If  $\Delta n_e > v_{\perp}^2(\ell) n_c / c^2$ , we have  $v_{\perp}^2(r) < 0$ , which is impossible. The only way out is then for the ray to have  $\Delta n_e = 0$ , i.e. the ray stays in the same cell and reflects with  $v_{\perp}(r) = -v_{\perp}(\ell)$ .

The basic algorithmic steps for tracing a single ray through all cells in one block (see Figure 18.4) can be summarized as follows: The initial situation has the ray positioned on one of the block's faces with velocity components such that the ray's direction of movement is into the block.

- Identify ray entry cell  $i_0$ , entry position  $\mathbf{r}_0$ , entry velocity  $\mathbf{v}_0$  and entry power  $P_0$ .
- Calculate  $n_{e0}$  and  $T_{e0}$  at the entry point using equations 18.62 and 18.63.
- Solve equation 18.41 for time for each possible cell face. Identify the cell crossing time  $t_{cross}$  as minimum of all solutions  $> 0$ .
- Using  $t_{cross}$  and equation 18.41 again, find the cell exit position  $\mathbf{r}_{exit}$ .
- Calculate  $\ln \Lambda[\mathbf{r}_0]$ ,  $\nu_{ib}(0)$ ,  $U$ ,  $W$ ,  $R$ ,  $S$  and evaluate  $P_{exit}$  using equations 18.47 and 18.59.
- Using  $t_{cross}$  and equation 18.40, find the ray's exit velocity  $\mathbf{v}_{exit}$ .
- Based on  $\mathbf{r}_{exit}$  and  $\mathbf{v}_{exit}$  determine new entry cell  $i_{exit}$ .
- Calculate the electron number density jump  $\Delta n_e$  between both cells  $i_0$  and  $i_{exit}$ , using equation 18.39.

- Check for reflection or refraction using Snell's law equation 18.67 and update  $\mathbf{v}_{exit}$  and possibly new entry cell  $i_{exit}$ .
- If cell  $i_{exit}$  is still in block, set all exit variables equal to entry variables and repeat the steps.

#### 18.4.4.2 Cubic Interpolation with Piecewise Parabolic Ray Tracing (CIPPRT)

The use of cubic interpolation schemes is an attempt at providing continuous  $n_e$  and  $T_e$  representations as well as continuous first derivatives  $\nabla n_e$  and  $\nabla T_e$  throughout the entire domain. This allows for the calculation of a more smoother path for each ray inside each cell when compared to the AVG algorithm. In effect, the cell-by-cell choppiness of the AVG algorithm can be avoided by taking many small ray steps inside each cell. Also, the troublesome  $n_e$  discontinuities at the cell boundaries and the application of Snell's law will disappear. The essential features of the cubic interpolation schemes are laid out in section 32.2. In what follows we show the piecewise parabolic ray tracing scheme.

The CIPPRT algorithm tries to map out the ray path inside each cell as a sequence of parabolic (i.e. constant acceleration) paths. It resembles the AVG algorithm and uses the same equations Eq.(18.40) and Eq.(18.41) for determination of velocity and position of each parabolic section of the path. The main difference between the AVG and the CIPPRT is that the latter uses the cubic interpolated ray acceleration field at each cell point, whereas the former assumes the same constant acceleration at each point throughout the cell. The CIPPRT can hence be viewed as a succession of AVG steps inside each cell. The energy deposition of each parabolic path section is also calculated like for the AVG algorithm, namely solving the exponential time integral in Eq.(18.47) using the second order Gaussian quadrature in Eq.(18.59). The required  $n_e$  and  $T_e$  at the Gaussian quadrature points are evaluated using the cubic interpolation equation Eq.(32.11). The following scheme illustrates the steps involved for tracing a ray through a cell, assuming the ray is somewhere located in the cell at  $\mathbf{r}_0$  with velocity  $\mathbf{v}_0$ :

1. Calculate, using Eq.(32.13), the electron number density gradient  $\nabla n_e(\mathbf{r}_0)$ .
2. Determine the time  $t$  to reach the next cell face using the quadratic time equation Eq.(18.41) and determine the ray's position  $\mathbf{r}(t)$  on the face.
3. Do two  $t/2$  steps and calculate the  $\mathbf{r}(2 \times t/2)$  position. The first  $t/2$  step is guaranteed to stay inside the cell. Hence  $\mathbf{r}(t/2) \in \text{cell}$ . However, despite  $\mathbf{r}(2 \times t/2)$  having the potential to step outside the cell, we only need it for comparison with  $\mathbf{r}(t)$ .
4. Form the error vector  $\mathbf{e} = \mathbf{r}(2 \times t/2) - \mathbf{r}(t)$ . If  $|\mathbf{e}| \leq \text{target accuracy}$ , accept the  $\mathbf{r}(t)$  position. If not, set  $t = t/2$  and repeat step 2).
5. Once a satisfactory stepping time has been determined, update the velocity using Eq.(18.40).

#### 18.4.4.3 Cubic Interpolation with Runge Kutta Integration (CIRK)

Instead of using the piecewise parabolic ray tracing approach, the CIRK algorithm advances the ray through cells using Runge Kutta (RK) integration 34.2. An advantage of using the RK integrator is that the rate of power loss ODE 18.42 can be directly incorporated into the RK solution vector, thereby gaining access over its error control. The ray tracing RK vector has 7 entries

$$\frac{d}{dt} \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \\ P \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ \mathbf{a} \\ -\nu_{ib}P \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ -c^2 \nabla n_e(\mathbf{r})/2n_e \\ -\nu_{ib}(\mathbf{r})P \end{pmatrix}, \quad (18.68)$$

and the independent variable  $t$  does not appear on the rhs. Since each ray must be traced on a cell-by-cell basis, we have to make sure that each RK step either stays within the cell or hits one of its walls. The CIRK algorithm therefore has to use the confined RK stepping routine.



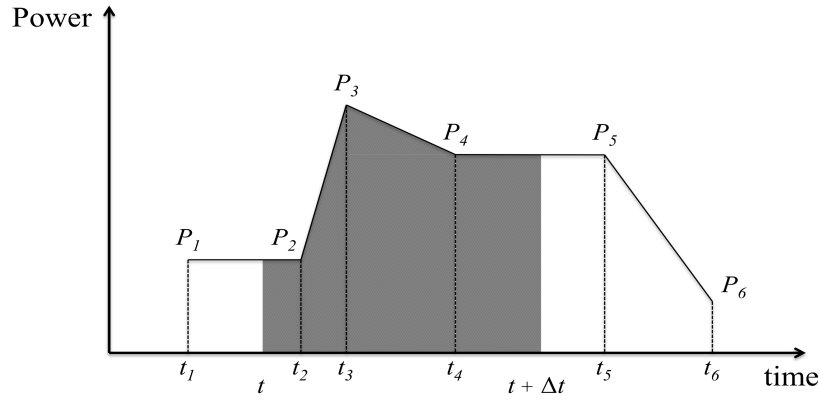


Figure 18.5: The structure of the laser pulse.

### 18.4.5 Setting up the Laser Pulse

The laser's pulse contains information about the laser's energy profile and can thus be characterized by a function of laser power in terms of time  $P(t)$ . The curve of  $P(t)$  gives the power of the laser pulse at any given simulation time. In FLASH, the form of  $P(t)$  is given by a series of points, which are connected via straight lines (see Figure 18.5). Each point denotes a power/time pair  $P_i(t_i)$ . The average laser power  $\bar{P}$  of the pulse during a time step from  $t \rightarrow t + \Delta t$  is then

$$\bar{P} = \frac{\int_t^{t+\Delta t} P(t) dt}{\Delta t}, \quad (18.69)$$

where the integration is performed piecewise between all the points  $P_i(t_i)$  contained within the relevant time frame. The average laser power will be distributed among all the rays that will be created during that particular time step. Note, that  $t_1$  is the time the laser first comes into existence. In Figure 18.5 there is therefore no line connecting the origin (zero power) with  $P_1$ . If the user wants a gradual buildup of laser power from zero, an explicit point  $P_1 = 0$  must be given at time  $t_1$ . The same consideration applies for the other end of the timescale.

### 18.4.6 Setting up the Laser Beam

The laser's beam contains all the information about orientation and shape of the laser. The orientation is characterized by specifying lens and target coordinates. The shape of the laser is given by the size and shape of the lens and target cross section areas and the cross-sectional ray power distribution law. Figure 18.6 is helpful in visualizing the position of the vectors for the formulas below. The most important vectors for setting up the rays are the two local elliptical semiaxes unit vectors  $\mathbf{u}_1$  and  $\mathbf{u}_2$ . These two unit vectors, together with the lens and target positions, are very convenient in calculating the rays lens and target coordinates. Note, that the unit vectors are the same for both the lens and the target, if defined from their corresponding local elliptical centers. In what follows, vectors in capital letters are defined in the global coordinate system and vectors in small letters are defined in the local target coordinate system.

#### 18.4.6.1 The Local Elliptical Semiaxis Unit Vectors

The laser beam originates at the lens, whose center is located at  $\mathbf{L}$ , and hits the target, whose center is located at  $\mathbf{T}$ . In 3D geometries, the elliptical target area is defined as an ellipse perpendicular to the lens-target line and whose largest semiaxis  $\mathbf{s}_1$  is positioned such that it makes a torsional angle  $\phi_1$  with the local target z-axis (see Figure 18.6). Let us define the beam vector  $\mathbf{b} = \mathbf{L} - \mathbf{T}$  pointing from the target to the

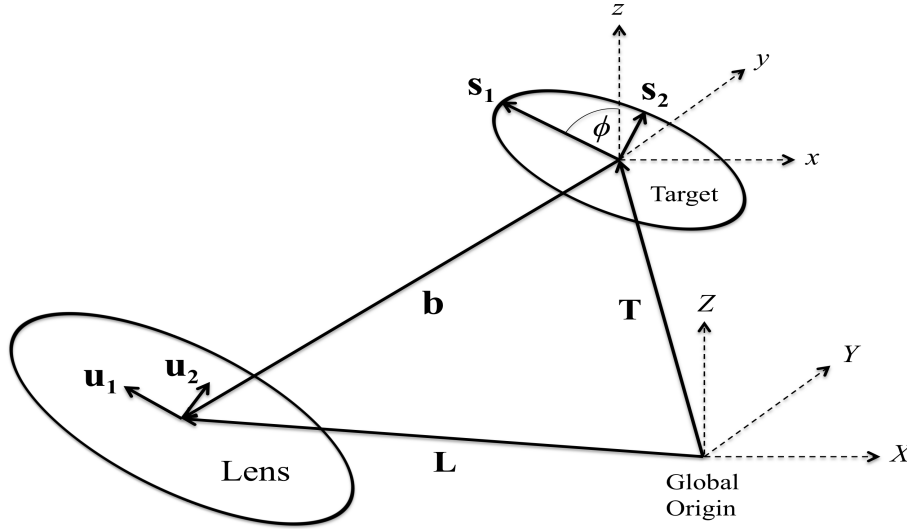


Figure 18.6: The laser beam.

lens and connecting the two respective centers. We then have the defining equations for  $\mathbf{s}_1$

$$s_{1x}b_x + s_{1y}b_y + s_{1z}b_z = 0 \quad (18.70)$$

$$s_{1x}^2 + s_{1y}^2 + s_{1z}^2 = \ell_1^2 \quad (18.71)$$

$$s_{1z} = \ell_1 \cos \phi_1 \cos(\theta - \pi/2), \quad (18.72)$$

where  $\ell_1$  is the length of  $\mathbf{s}_1$  and  $\theta$  is the angle that the beam vector  $\mathbf{b}$  makes with the local  $z$ -axis. The first equation 18.70 says that  $\mathbf{b}$  and  $\mathbf{s}_1$  are orthogonal. The second one 18.71 defines the length of  $\mathbf{s}_1$  and the third one 18.72 defines the local  $z$ -axis projection of  $\mathbf{s}_1$ . The last equation can be rewritten as

$$s_{1z} = \ell_1 \frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \cos \phi_1. \quad (18.73)$$

Forming an expression for  $s_{1y}^2$  from equation 18.70, an expression for  $s_{1z}^2$  from equation 18.72 and inserting the results into equation 18.71, we obtain a quadratic equation in  $s_{1x}^2$ , from which we can obtain all three components. We get after some algebra and simplifications

$$s_{1x} = \frac{\ell_1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_x b_z}{|\mathbf{b}|} \cos \phi_1 \pm b_y \sin \phi_1 \right] \quad (18.74)$$

$$s_{1y} = \frac{\ell_1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_y b_z}{|\mathbf{b}|} \cos \phi_1 \mp b_x \sin \phi_1 \right] \quad (18.75)$$

$$s_{1z} = \ell_1 \frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \cos \phi_1. \quad (18.76)$$

The two possible solutions for  $s_{1x}$  and  $s_{1y}$  correspond to the two possible definitions of the rotation angle  $\phi_1$  in either clockwise or counterclockwise rotation from the  $z$ -axis when looking along the beam vector  $\mathbf{b}$  from the lens to the target. Let us henceforth define  $\phi_1$  to be the clockwise rotation. Then the lower signs in  $s_{1x}$

and  $s_{1y}$  apply and dividing each component by  $\ell_1$  we obtain for the unit vector components

$$u_{1x} = \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_x b_z}{|\mathbf{b}|} \cos \phi_1 - b_y \sin \phi_1 \right] \quad (18.77)$$

$$u_{1y} = \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ -\frac{b_y b_z}{|\mathbf{b}|} \cos \phi_1 + b_x \sin \phi_1 \right] \quad (18.78)$$

$$u_{1z} = \frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \cos \phi_1. \quad (18.79)$$

The second elliptical semiaxis  $\mathbf{s}_2$  is perpendicular to the first and lays in the same elliptical target plane. If we define it to be at a right angle in clockwise direction from  $\mathbf{s}_1$ , the torsional angle it makes with the local z-axis is  $\phi_2 = \phi_1 + \pi/2$ . The formulas for its unit vector components are the same as for  $\mathbf{s}_1$  but with  $\phi_1$  replaced by  $\phi_2$ . From trigonometric sine and cosine relations we can re-express them in terms of  $\phi_1$

$$u_{2x} = \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ \frac{b_x b_z}{|\mathbf{b}|} \sin \phi_1 - b_y \cos \phi_1 \right] \quad (18.80)$$

$$u_{2y} = \frac{1}{\sqrt{b_x^2 + b_y^2}} \left[ \frac{b_y b_z}{|\mathbf{b}|} \sin \phi_1 + b_x \cos \phi_1 \right] \quad (18.81)$$

$$u_{2z} = -\frac{\sqrt{b_x^2 + b_y^2}}{|\mathbf{b}|} \sin \phi_1. \quad (18.82)$$

Note the importance of the  $\sqrt{b_x^2 + b_y^2}$  term. If this term is equal to zero, then the unit vectors become undefined. This corresponds to the laser beam being parallel to the global z-axis (and thus coinciding with the local z-axis). Then both elliptical semiaxes are not defined uniquely through a z-axis torsional angle of zero. In this case the torsional angle must be defined through one of the other coordinate axis. The following coordinate index permutations in the above formulas apply:

$$\phi_1 \text{ defined through x-axis} = x \rightarrow y, y \rightarrow z, z \rightarrow x \quad (18.83)$$

$$\phi_1 \text{ defined through y-axis} = x \rightarrow z, y \rightarrow x, z \rightarrow y. \quad (18.84)$$

In 2D geometries, the beam's lens and target areas shrink to a line segment. All z-components are zero and the torsional angle is equal to  $\pi/2$ . The components of the unit vector can be deduced from the equations 18.77 and 18.78 as

$$u_x = -\frac{b_y}{|\mathbf{b}|} \quad (18.85)$$

$$u_y = \frac{b_x}{|\mathbf{b}|}. \quad (18.86)$$

#### 18.4.6.2 Extremum Values for the Elliptical Target Zone

In 3D simulations, the planar elliptical target zone can be placed in any possible way inside the domain by specifying the lens  $\mathbf{L}$  and target  $\mathbf{T}$  positions, both elliptical target zone semiaxes lengths  $\ell_1$  and  $\ell_2$ , the first semiaxis torsion angle  $\phi_1$  and the coordinate axis to which  $\phi_1$  refers to. We wish to enforce in the code a complete containment of the entire target plane inside the domain boundaries. To check this condition we need the extremum coordinate values of the elliptical boundary curve of the target. The collection of all points  $\mathbf{e}$ , based on the local target coordinate system, can be given in the following implicit form

$$\mathbf{e} = \ell_1 \mathbf{u}_1 \cos(\omega) + \ell_2 \mathbf{u}_2 \sin(\omega), \quad 0 \leq \omega \leq 2\pi. \quad (18.87)$$

Differentiating with respect to  $\omega$  and equating to zero we get the minimax condition on all coordinate components as

$$\omega_{min/max} = \arctan\left(\frac{\ell_2 \mathbf{u}_2}{\ell_1 \mathbf{u}_1}\right), \quad (18.88)$$

where  $\omega_{min/max}$  denotes the vector of the minimax angles for each cartesian component. The equation 18.88 has two possible answers for each  $\omega$  component in the range  $0 \leq \omega \leq 2\pi$ , corresponding to the minimum and the maximum value. The  $\omega_{min}$  and  $\omega_{max}$  angles differ by  $\pi$  radians for each cartesian component. The corresponding minima and maxima on the elliptical boundary curve are obtained by inserting the  $\omega_{min}$  and  $\omega_{max}$  angles into equation 18.87.

In order to simplify things, we note that what we need are the sine and cosine values of  $\omega_{min}$  and  $\omega_{max}$ . From the definition of the trigonometric functions based on the length of the three sides of a right-angled triangle (a = opposite side, b = adjacent side, c = hypotenuse for an angle  $\omega$ ), we have, using  $\sin \omega = a/c$ ,  $\cos \omega = b/c$ ,  $\tan \omega = a/b$  and  $c^2 = a^2 + b^2$

$$\sin(\arctan(a/b)) = \sin(\omega) = a/c = \frac{a}{\sqrt{a^2 + b^2}} \quad (18.89)$$

$$\cos(\arctan(a/b)) = \cos(\omega) = b/c = \frac{b}{\sqrt{a^2 + b^2}}. \quad (18.90)$$

When applied to equation 18.87, we obtain

$$\mathbf{e}_{min/max} = \pm \sqrt{(\ell_1 \mathbf{u}_1)^2 + (\ell_2 \mathbf{u}_2)^2}. \quad (18.91)$$

The corresponding minimax equation for the 2D geometries (ellipse  $\rightarrow$  line segment) is

$$\mathbf{e}_{min/max} = \pm \ell \mathbf{u}, \quad (18.92)$$

with  $\ell$  being half the length of the target line segment and  $\mathbf{u}$  the line segment unit vector.

### 18.4.7 Setting up the Rays

For each Energy Deposition unit call, the code sets up the initial collection of rays, defined to be located on the domain boundary with specific velocity components such that they will hit the target area at precise locations if they would cross an empty domain. Key concepts in setting up the rays initial position, velocity and power are the elliptical local square, radial, delta or statistical grids and the beam cross section power function. Rays will be launched from the lens grid intersection points to the corresponding target grid intersection points. Using a square grid, a uniform beam cross-sectional ray density will be achieved, although this could be relaxed in the future to include the possibility of rectangular grids leading to different cross-sectional ray densities along the two elliptical semiaxes directions. A radial grid places the rays on concentric ellipses.

#### 18.4.7.1 The Elliptical Lens/Target Local Square Grid

In 3D geometries, both the lens and target areas are defined as two similar ellipses (different size, same shape). Given a specific elliptical shape by defining the lengths of both semiaxes  $\ell_1 \geq \ell_2$ , we can set up a square grid inside the ellipse, such that the number of grid intersection points matches closely the number of rays  $N_{rays}$  specified by the user. The grid is defined by the separation  $\Delta$  between two grid points and the placement of the grid's origin at the center of the ellipse. Our goal is to find this  $\Delta$  value.

Denote by  $N_{rectangle}$  the number of grid intersection points for the circumscribing rectangle with sides  $2\ell_1$  and  $2\ell_2$ . We then have

$$\begin{aligned} N_{rectangle} &= \text{Number of ellipse points} \frac{\text{Area rectangle}}{\text{Area ellipse}} \\ &= \frac{4N_{rays}}{\pi}, \end{aligned} \quad (18.93)$$

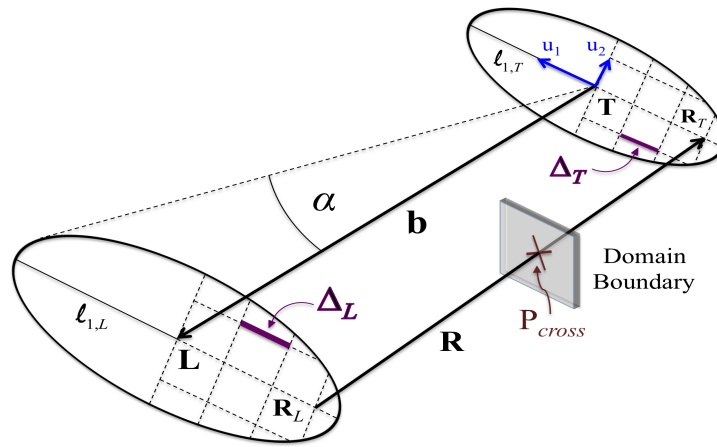


Figure 18.7: Setting up the rays between the beam's lens and target area using a square grid. Only half of the square elliptical grids are shown for clarity.

this relation holding only approximately due to the finite resolution of the grid. Let us denote by  $r = \ell_1/\ell_2$  the ratio between the largest and smallest semiaxis. If  $n_1$  is the number of tics along the  $\ell_1$ -semiaxis starting at the grid origin (0,0), then the number of tics along the  $\ell_2$ -semiaxis is approximately equal to  $n_1/r$ . Looking at the circumscribed rectangle area, the total number of grid intersection points laying on the grid's axes will be twice those on each semiaxis plus the grid's origin:  $2n_1 + 2n_1/r + 1$ . The total number of grid intersection points not on any of the axes is  $4(n_1)(n_1/r)$ . Both numbers together should then equal to  $N_{rectangle}$ . Hence, from 18.93, we are lead to a quadratic equation in  $n_1$

$$\frac{4}{r}n_1^2 + \left(2 + \frac{2}{r}\right)n_1 - \left(\frac{4N_{rays}}{\pi} - 1\right) = 0, \quad (18.94)$$

whose solution is

$$n_1 = \frac{1}{4} \left[ -(1+r) + \sqrt{(1-r)^2 + \frac{16rN_{rays}}{\pi}} \right]. \quad (18.95)$$

Always  $n_1 > 0$ , which can easily be seen from the lowest possible value that  $n_1$  can attain for  $N_{rays} = 1$  and the lowest possible ratio  $r = 1$  (in this case  $n_1 = 0.06418\dots$ ). Since  $n_1$  has to be an integer we take the ceiling value of the solution 18.95. If the user specified  $N_{rays} = 1$ , the search for  $n_1$  is bypassed and the ray is placed at the elliptical origin.

Having  $n_1$ , the next task is to find the optimum (or close to optimum) grid spacing  $\Delta$ . This is done by defining a minimum and maximum grid spacing value

$$\Delta_{min} = \frac{\ell_1}{n_1 + 1} \quad (18.96)$$

$$\Delta_{max} = \frac{\ell_1}{n_1 - 1}. \quad (18.97)$$

A series of  $\Delta_{min} \leq \Delta_k \leq \Delta_{max}$  grid spacings is then tested, and for each  $\Delta_k$  the number of grid intersection points  $N_k$  inside the ellipse area is determined. Of all  $N_k$  obtained, a certain number of them will be closest to  $N_{rays}$ . The average over all these  $\Delta_k$  will then be taken as the final  $\Delta$  value. For this  $\Delta$  we compute the final number of grid intersection points, which will then replace the user's specified  $N_{rays}$ .

Since the target  $\ell_1$  and  $\ell_2$  semiaxes are specified by the user, the  $\Delta_T$  for the target square grid is evaluated using the above algorithm. The corresponding lens  $\Delta_L$  value is set using the similarity in size between the

lens and the target. Using the user's specified  $\ell_1$  for the lens, the preservation of length relations between similar objects leads to:

$$\frac{\Delta_L}{\ell_{1,L}} = \frac{\Delta_T}{\ell_{1,T}} \quad (18.98)$$

and hence

$$\Delta_L = \frac{\ell_{1,L}}{\ell_{1,T}} \Delta_T. \quad (18.99)$$

In 2D geometries the situation is much simpler. Due to the linear shape of the lens and target areas, it is very easy to calculate the  $\Delta_T$  value of the linear target grid such that exactly  $N_{rays}$  grid points are obtained with the outer two grid points coinciding with the target end points. The corresponding  $\Delta_L$  is evaluated using equation 18.99.

#### 18.4.7.2 The Elliptical Lens/Target Local Radial Grid

In order to set up the radial lens and target grids, we use the implicit definition of the elliptical curve from 18.87. The grid will then be defined as the number of tics on the  $\ell_1, \ell_2$  pair and the number of tics on the angle  $\omega$  within the range  $0 \leq \omega \leq 2\pi$ . The number of tics on both of these 'axes' will be the same and will be denoted by  $n$ . The total number of radial grid points within the lens and target ellipses will be equal to  $1 + n(n + 1)$ , leading to the quadratic equation

$$n^2 + n + 1 = N_{rays}, \quad (18.100)$$

whose relevant solution is

$$n = \frac{1}{2} \left[ -1 + \sqrt{4N_{rays} - 3} \right]. \quad (18.101)$$

In order to reduce the error to  $N_{rays}$ , rounding to the nearest integer is applied for  $n$ . The individual grid points are calculated from

$$\mathbf{e}_{ij} = \frac{\ell_1 i}{n} \mathbf{u}_1 \cos\left(\frac{2\pi j}{n}\right) + \frac{\ell_2 i}{n} \mathbf{u}_2 \sin\left(\frac{2\pi j}{n}\right), \quad (18.102)$$

where the index ranges are

$$j = 0, 1, 2, \dots, n \quad (18.103)$$

$$i = \min(1, j), \dots, n. \quad (18.104)$$

In an effort to provide more control to the user, the default 'same number of tics on both radial and angular axes' has been extended, such that the user can enforce one or even both of these values. This is controlled by setting the requested number of radial and/or angular tics as runtime parameters for each beam. The resulting number of rays will then be recalculated and overwritten.

#### 18.4.7.3 The Elliptical Lens/Target Local Delta Grid

One of the drawbacks of the local square grid is that the user has no direct control of the resulting tic separation. The main goal when setting up the local square grid is to have the number of rays match as closely as possible the number of rays requested by the user. The local delta grid is provided to give the user full control over the tic separations on both semiaxes, relaxing at the same time the number of rays constraint. The delta grid becomes useful if the user wants to enforce a more smooth energy deposition in the cells hit by the 3D laser beam. Rays can be forced to hit each cell well away from the cell boundary, thereby avoiding the initial uneven ray distributions due to some rays hitting the block walls with corresponding nudging to one block or the other. As a consequence the delta grids are most useful if the target area of the beam has or is known to have a uniform refinement level.

The setup of the delta grid is simple: the user has to provide the tic separation values as beam runtime parameters. The beam setup code then adheres strictly to these values and excludes any grid points laying outside the elliptical beam boundary. The number of rays requested initially by the user is completely ignored and replaced by the number of rays determined for the delta grid. The user must make sure that the supplied memory requirements for ray storage are ok. This can be done by estimating the number of rays resulting from the delta grid on the area of the circumscribed rectangle with sides  $2\ell_1$  and  $2\ell_2$  and rescaling by the ellipse/rectangle area factor. If we denote the user supplied tic separations corresponding to the  $\ell_1$  semiaxis and  $\ell_2$  semiaxis by  $\Delta_1$  and  $\Delta_2$ , then the estimated number of rays would be:

$$\begin{aligned} N_{rays} &\approx 2 \left( \frac{\ell_1}{\Delta_1} \right) \times 2 \left( \frac{\ell_2}{\Delta_2} \right) \times \frac{\pi}{4} \\ &= \frac{\pi \ell_1 \ell_2}{\Delta_1 \Delta_2}. \end{aligned} \quad (18.105)$$

The first tics of the delta grid along each grid axis start at  $\Delta_1/2$  and  $\Delta_2/2$ . The grid center  $(0,0)$  is thus not part of the delta grid.

#### 18.4.7.4 The Elliptical Lens/Target Local Statistical Grid

The local statistical grid is defined by a statistical collection of  $(x, y)$  pairs within the range  $[-1, 1]$ , where the  $x$  and  $y$  denote fractions of the  $\ell_1$  and  $\ell_2$  semiaxis. A random number generator for the range  $[0, 1]$  is used and the numbers are shifted to the  $[-1, 1]$  range by multiplying by 2 and adding  $-1$ . Every  $(x, y)$  pair is checked, if it actually lays within the ellipse and retained if it does. The random  $(x, y)$  pair generation stops, once the requested number of rays is reached. In order to use different statistical grids for each time step, the statistical grid is regenerated afresh for each time step using a different random seed value.

#### 18.4.7.5 Beam Cross Section Power Function

The beam cross section power function describes the power distribution of the rays inside the beam at launching time. Currently there are two types of power distribution functions implemented in FLASH: 1) uniform (flat) distribution (equal power) and 2) gaussian decay from the center of the beam. The first one is trivial: every ray gets assigned an equal amount of power. The gaussian decay function assigns to each ray a relative weight according to the position inside the elliptical cross section of the beam. Using the local  $\mathbf{s}_1$  and  $\mathbf{s}_2$  target coordinate system located at the center of the beam, the gaussian weighting function for 2D target areas (3D geometries) reads

$$w = \exp \left[ - \left[ \left( \frac{x}{R_x} \right)^2 + \left( \frac{y}{R_y} \right)^2 \right]^\gamma \right], \quad (18.106)$$

where  $x$  and  $y$  denote the local coordinates inside the ellipse along the  $\mathbf{s}_1$  and  $\mathbf{s}_2$  axes, respectively,  $R_x$  and  $R_y$  are user defined decay radii values and  $\gamma$  is the user defined gaussian super exponent. For 1D target areas (2D geometries) the weighting function is

$$w = \exp \left[ - \left[ \left( \frac{x}{R_x} \right)^2 \right]^\gamma \right]. \quad (18.107)$$

In order to determine the actual power assigned to each ray, we use the average laser power from equation 18.69

$$P_{ray} = \bar{P} \frac{w_{ray}}{\sum w_{ray}}. \quad (18.108)$$

The sum of the weights over all rays plays the role of a sort of partition function for the beam grid and can thus be precomputed and become part of the beam properties when setting up the beams.

### 18.4.7.6 The Rays Initial Position and Velocity

In 3D, after the local lens and target grids have been set up, each ray has associated with it a local lens elliptical coordinate pair  $(x_L, y_L)$  and a local target elliptical coordinate pair  $(x_T, y_T)$ . The corresponding global coordinates can be obtained using the global lens and target center coordinates and the elliptical unit vectors (see Figure 18.7)

$$\mathbf{R}_L = x_L \mathbf{u}_1 + y_L \mathbf{u}_2 + \mathbf{L} \quad (18.109)$$

$$\mathbf{R}_T = x_T \mathbf{u}_1 + y_T \mathbf{u}_2 + \mathbf{T}. \quad (18.110)$$

Defining now the ray vector  $\mathbf{R}$  pointing from the lens to the target

$$\mathbf{R} = \mathbf{R}_T - \mathbf{R}_L, \quad (18.111)$$

we can state the parametric ray line equation

$$\mathbf{P} = \mathbf{R}_L + w\mathbf{R}. \quad (18.112)$$

The ray line equation is used to determine where and which domain boundary surface the ray will hit. Consider a domain boundary surface contained within a plane given by the equation

$$\begin{aligned} A_x x + A_y y + A_z z + D &= 0 \\ \mathbf{A} \cdot (x \ y \ z) + D &= 0. \end{aligned} \quad (18.113)$$

The value of the real  $w$  parameter where the ray line meets this plane is obtained by inserting the ray line equation into the plane equation. It is

$$w = -\frac{\mathbf{A} \cdot \mathbf{R}_L + D}{\mathbf{A} \cdot \mathbf{R}}. \quad (18.114)$$

Inserting this  $w$  value into the ray line equation 18.112 we obtain the location  $\mathbf{P}_{cross}$  of the crossing point on the plane. From Figure 18.7, for the ray vector to cross the plane, the value of  $w$  must be in the range  $0 \leq w \leq 1$ . A value of  $w = 0$  or  $w = 1$  indicates that the plane is crossing the lens or target area, respectively. If  $w$  is in proper range, it must next be checked if  $\mathbf{P}_{cross}$  is located within the domain boundary surface. If yes, that  $w$  value is accepted. Note, that several proper  $w$  values can be obtained. A ray crossing near the corner of a rectangular domain gives three proper  $w$  values corresponding to crossing the three rectangular planes defining the corner. Since the rays originate from the lens, the relevant  $w$  is the one with minimum value and the corresponding  $\mathbf{P}_{cross}$  will be taken as the rays initial position. The initial ray velocity components are determined from the unit vector of the ray vector

$$\mathbf{v} = v_0 \frac{\mathbf{R}}{|\mathbf{R}|}, \quad (18.115)$$

where  $v_0$  is the initial magnitude of the velocity (in most simulations this is the speed of light). For 2D geometries all the above equations remain the same, except for the ray global coordinates, for which the terms in  $\mathbf{u}_2$  are dropped and for the domain boundary planes, for which the terms involving the  $z$ -component in the defining equation 18.113 do not exist. The following steps summarize the determination of the initial ray positions and velocity components for each ray:

- Form the ray vector  $\mathbf{R}$  using 18.111.
- Find the collection  $\{w\}$  of all  $0 \leq w \leq 1$  values using 18.114 for all domain surface planes  $(\mathbf{A}, D)$ .
- Using the ray line equation 18.112, remove all  $w$  values from  $\{w\}$  which lead to plane crossing points  $\mathbf{P}_{cross}$  not contained within the domain boundary surface.
- Take the minimum of the remaining  $w$ 's in  $\{w\}$  and calculate the corresponding  $\mathbf{P}_{cross}$ . This is the ray's initial position vector.
- Using the ray vector  $\mathbf{R}$  again calculate the velocity components using 18.115.



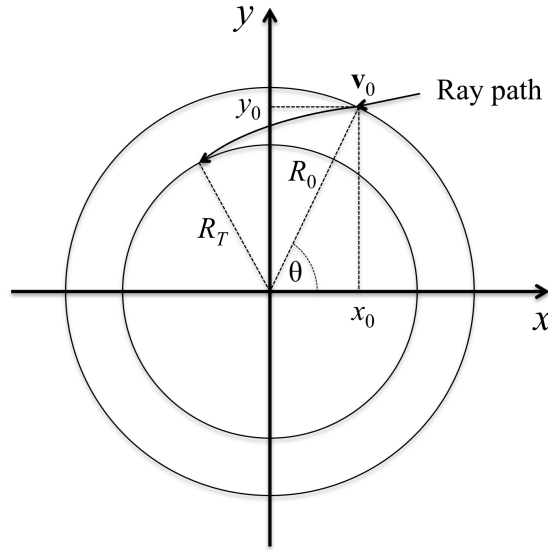


Figure 18.8: Tracing a ray in 3D in a 2D cylindrical domain. View onto the R-disk. The z-axis has been left out for clarity.

#### 18.4.8 3D Laser Ray Tracing in 2D Cylindrical Symmetry

Performing a pure 2D cylindrical ray tracing has an obvious disadvantage: each ray must also be treated as a 2D cylindrical symmetrical object. Each ray can only hit the R-disk of the cylindrical domain at precisely 90 degrees and no variation in this incident angle is possible. However, if one treats the cylinder as a true 3D object, then it is possible to trace each ray through this 3D cylinder. The advantage of retaining the 2D cylindrical description of the domain is obvious: only 2D storage is needed to describe the properties of the domain. 3D in 2D ray tracing is much more complicated than either the simple pure 3D or pure 2D counterparts. The interplay between the polar and the cartesian coordinates leads to ray tracing equations which can only be solved approximately via the use of elliptical integrals. Rather than approximating the integrals involved, a second approach decomposes the 3D cylinder into several identical cylindrical wedges with each wedge having planar boundaries. The larger the number of wedges the more accurate the 3D in 2D ray tracing will be. Both approaches (approximating the integrals and wedging the 3D cylinder) are entirely equivalent.

##### 18.4.8.1 The Exact 3D in 2D Ray Tracing Solution

We will first state the exact time-radial solution of an object of mass  $m$  moving in a central force environment with no external forces acting on it. The motion of such an object is characterized by constant energy and angular momentum and is hence confined to a 2D plane. There are two ways to describe the position  $\mathbf{r}$ , velocity  $\mathbf{v}$  and acceleration  $\mathbf{a}$  vectors of a particle in a 2D plane: using cartesian  $\hat{\mathbf{i}}, \hat{\mathbf{j}}$  or polar  $\hat{\mathbf{R}}, \hat{\boldsymbol{\theta}}$  unit vectors. They are interrelated by

$$(\hat{\mathbf{R}}, \hat{\boldsymbol{\theta}}) = (\hat{\mathbf{i}}, \hat{\mathbf{j}}) \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad (18.116)$$

Using these unit vectors and the differentiation chain rule we get

$$\mathbf{r} = x\hat{\mathbf{i}} + y\hat{\mathbf{j}} = R\hat{\mathbf{R}} \quad (18.117)$$

$$\mathbf{v} = \dot{x}\hat{\mathbf{i}} + \dot{y}\hat{\mathbf{j}} = \dot{R}\hat{\mathbf{R}} + R\dot{\theta}\hat{\boldsymbol{\theta}} \quad (18.118)$$

$$\mathbf{a} = \ddot{x}\hat{\mathbf{i}} + \ddot{y}\hat{\mathbf{j}} = (\ddot{R} - R\dot{\theta}^2)\hat{\mathbf{R}} + (R\ddot{\theta} + 2\dot{R}\dot{\theta})\hat{\boldsymbol{\theta}} \quad (18.119)$$

where each dot on the dotted variables stands for  $d/dt$ . The angular momentum of the object in polar coordinates is

$$\begin{aligned}\mathbf{L} &= \mathbf{r} \times m\mathbf{v} \\ &= R\hat{\mathbf{R}} \times m(\dot{R}\hat{\mathbf{R}} + R\dot{\theta}\hat{\boldsymbol{\theta}}) \\ &= mR^2\dot{\theta}(\hat{\mathbf{R}} \times \hat{\boldsymbol{\theta}})\end{aligned}\tag{18.120}$$

and its magnitude

$$L = mR^2\dot{\theta}\tag{18.121}$$

Since only a central force component is present, this force depends only on the radial part

$$\mathbf{F}(R) = m\mathbf{a}_R = m(\ddot{R} - R\dot{\theta}^2)\hat{\mathbf{R}}.\tag{18.122}$$

We are now ready to state the equation of motion of the object in polar coordinates. Let's assume the object has initially at time  $t_0$  the coordinates  $(R_0, \theta_0)$ , and let  $E$  be its constant energy.  $E$  is composed of two parts: kinetic and potential energy. Both parts will change as time passes by. The potential energy  $U$  can only change along a change in  $R$ , because the force has only a radial component. We have

$$\begin{aligned}E &= \frac{1}{2}mv^2 + U(R) \\ &= \frac{1}{2}m(\dot{R}^2 + R^2\dot{\theta}^2) + U(R) \\ &= \frac{1}{2}m\dot{R}^2 + \frac{1}{2}\frac{L^2}{mR^2} + U(R)\end{aligned}\tag{18.123}$$

where in the last step we have eliminated the angular dependence through the use of Eq. (18.121). The last equation for  $E$  is a first order differential equation allowing for separation of variables

$$\int_{R_0}^{R_T} \left[ \frac{2}{m}(E - U(R)) - \frac{L^2}{m^2 R^2} \right]^{-1/2} dR = \int_{t_0}^{t_T} dt\tag{18.124}$$

The potential energy can be obtained via integration

$$\begin{aligned}U(R) &= - \int_{R_0}^R F(R') dR' + U(R_0) \\ &= -m \int_{R_0}^R a_R(R') dR' + U(R_0).\end{aligned}\tag{18.125}$$

The value of the initial potential energy is arbitrary and is conveniently set to zero. Assuming further that  $t_0 = 0$ , we obtain

$$t_T = \int_{R_0}^{R_T} \left[ \frac{2E}{m} - \frac{L^2}{m^2 R^2} + 2 \int_{R_0}^R a_R(R') dR' \right]^{-1/2} dR.\tag{18.126}$$

Given an initial radial  $R_0$  and a final target radial  $R_T$  value,  $t_T$  is the time the object will take in travelling from  $R_0$  to  $R_T$  under the specified initial conditions at  $R_0$  (values of  $E$  and  $L$ ). If the object will not reach  $R_T$  then  $t_T$  is either negative or complex. The time equation (18.126) can be solved analytically only for a small number of cases (like for example if  $a_R = 0$ ). All other cases require numerical approximation to the radial integral. For our case, when the ray moves through one radial 2D cylindrical zone using the AVG approximation 18.4.4.1, the central force acceleration is constant, i.e  $a_R = a$ . For this case we have

$$t_T = \int_{R_0}^{R_T} \left[ \frac{2E}{m} - \frac{L^2}{m^2 R^2} + 2a(R - R_0) \right]^{-1/2} dR,\tag{18.127}$$

for which it is possible to give an analytical solution in terms of elliptical integrals. However, the analytical solution is far too complicated to be of practical use.

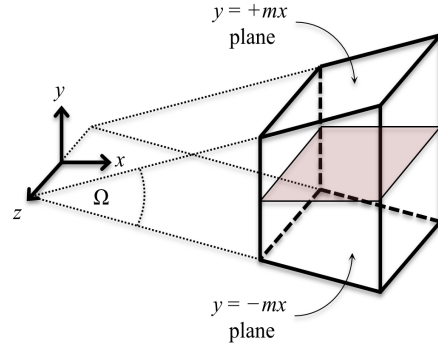


Figure 18.9: Shape and location of the TW cells.

#### 18.4.8.2 The Approximate 3D in 2D Ray Tracing Solution

The main reason for the complicated time equation has its roots in the  $\theta$  polar variable describing the rotational symmetry. An approximate treatment can be formulated, in which each circle is treated as a linear polygon with  $n$  sides ( $n$  not necessarily integer), where the value of  $n$  determines the quality of the approximation. The exact 3D radial acceleration cylindrical problem in  $R, \theta, z$  coordinates is transformed into an approximate 3D cartesian problem in  $x, y, z$  coordinates, where the  $z$  coordinate remains unaffected. Each torus cell is hence approximated as a collection of truncated wedge (TW) cells. The trapezoidal sides of these wedges are parallel to the  $x, y$  plane and the rectangular sides are perpendicular to this plane. Since all TW cells have exactly the same size and shape when coming from the same torus cell, it suffices to concentrate on just one of them. These representative TW cells will be placed inside the  $x, y$  plane, such that the positive  $x$  axis divides the trapezoidal sides into two equal areas. The collection of all representative TW cells has the shape of a wedge with opening angle  $\Omega$ . In order to trace the rays through the TW cells, we need the equation of all the cell boundary planes. The equations of the cell planes perpendicular to the  $z$  axis (containing the trapezoidal sides) are simply  $z = z_{cell}$ . The equations of the cell planes perpendicular to the  $x$  axis are similarly  $x = x_{cell}$ . The remaining cell plane equations corresponding to the non-coplanar cell faces are (see Figure 18.9)

$$\begin{aligned} y &= \pm \tan\left(\frac{\Omega}{2}\right) x \\ &= \pm mx. \end{aligned} \quad (18.128)$$

The quadratic time equations to be solved for the  $x = x_{cell}$  and  $z = z_{cell}$  plane equations are of the same form as equation 18.65. For the plane equations in 18.128 we obtain, after inserting the appropriate forms into the general cartesian quadratic time equation 18.64 and considering the zero gradient approximation in  $y$  direction

$$-\frac{c^2}{4n_c} [\mp m \langle \nabla n_e \rangle_x] t^2 + [\mp m v_{0x} + v_{0y}] t + [\mp m r_{0x} + r_{0y}] = 0 \quad (18.129)$$

We now describe the three main tasks for the 3D in 2D ray tracing: 1) proper setup of lenses and corresponding targets, 2) initial placement of the 3D rays on the 2D cylindrical domain and 3) tracing of the rays through the truncated wedges.

#### 18.4.8.3 Extremum Global Radial 3D and 2D Distance Values for 3D Elliptical Lens and Target Zones

In curved domain setups (cylindrical and spherical), it becomes necessary to calculate extremum global radial distance values for a 3D elliptical curve from the domain origin in order to check proper placements of the

lens and target zones. We start by stating the global equation of the elliptical target curve  $\mathbf{E}$  (the treatment of the lens ellipse is analogous) in 3D in implicit form (see Figure 18.6):

$$\mathbf{E} = \mathbf{T} + \mathbf{s}_1 \cos(\omega) + \mathbf{s}_2 \sin(\omega), \quad 0 \leq \omega \leq 2\pi. \quad (18.130)$$

The 2D projections of this elliptical 3D curve in either the  $(x,y)$ ,  $(x,z)$  or  $(y,z)$  plane are themselves ellipses, however the 2D projections of  $\mathbf{s}_1$  and  $\mathbf{s}_2$  do no longer correspond to the 2D elliptical major and minor semi-axes. They generally are oblique to each other and their scalar product is non-zero. We will therefore develop general formulas taking the possible reduction of 3D to 2D into account. The square of the radial distance for any point on that curve from the global origin is

$$\begin{aligned} |\mathbf{E}|^2 &= |\mathbf{T}|^2 + 2\mathbf{s}_1 \cdot \mathbf{T} \cos(\omega) + 2\mathbf{s}_2 \cdot \mathbf{T} \sin(\omega) \\ &\quad + \mathbf{s}_1 \cdot \mathbf{s}_1 \cos^2(\omega) + \mathbf{s}_2 \cdot \mathbf{s}_2 \sin^2(\omega) + 2\mathbf{s}_1 \cdot \mathbf{s}_2 \cos(\omega) \sin(\omega). \end{aligned} \quad (18.131)$$

Differentiating with respect to  $\omega$  and setting equal to zero leads to the minimax equation

$$B \cos(\omega) - A \sin(\omega) - C \sin(\omega) \cos(\omega) + D[\cos^2(\omega) - \sin^2(\omega)] = 0, \quad (18.132)$$

where

$$A = \mathbf{s}_1 \cdot \mathbf{T} \quad (18.133)$$

$$B = \mathbf{s}_2 \cdot \mathbf{T} \quad (18.134)$$

$$C = \mathbf{s}_1 \cdot \mathbf{s}_1 - \mathbf{s}_2 \cdot \mathbf{s}_2 \quad (18.135)$$

$$D = \mathbf{s}_1 \cdot \mathbf{s}_2. \quad (18.136)$$

The minimax equation 18.132 contains a mixed trigonometric term and can only be solved by eliminating either the sine or the cosine function. This leads in general to a quartic equation

$$q_4 x^4 + q_3 x^3 + q_2 x^2 + q_1 x + q_0 = 0, \quad (18.137)$$

whose coefficients are, depending on if  $x$  is either  $\cos(\omega)$  or  $\sin(\omega)$ :

	$x = \cos(\omega)$	$x = \sin(\omega)$
$q_4$	$C^2 + 4D^2$	$C^2 + 4D^2$
$q_3$	$2AC + 4BD$	$-2BC + 4AD$
$q_2$	$A^2 + B^2 - C^2 - 4D^2$	$A^2 + B^2 - C^2 - 4D^2$
$q_1$	$-2AC - 2BD$	$2BC - 2AD$
$q_0$	$D^2 - A^2$	$D^2 - B^2$

All four  $A, B, C, D$  can have a value of zero. The coefficient of  $x^4$  plays a special role. Note, that for this coefficient to be zero, both  $C$  and  $D$  must be zero, which corresponds to a circular curve in both 3D and 2D. For a circular target the quartic reduces to a quadratic. For the 3D cases we always have  $D = 0$ . The following table 18.4 shows the different cases that can arise for 3D, together with a description of the geometrical shapes of the target figures involved and the corresponding maximum and minimum radial distances: In this table we have used our convention  $|\mathbf{s}_1| \geq |\mathbf{s}_2|$  and placement of the elliptical semi-axes in such a way that the angle they make with  $\mathbf{T}$  is  $\geq 90^\circ$ . The cases that arise for the 2D are summarized in the table 18.5: When solving the quartic or quadratic product equations, we must remember to eliminate the two non-valid extra solutions introduced due to squaring. All real solutions between  $-1$  and  $+1$  are inserted into 18.131 and the maximum and minimum of all four possible values are selected.

#### 18.4.8.4 Initial Placement of the 3D Rays on the 2D Cylindrical Domain

There are two kinds of domain surfaces for a cylindrical domain. The first kind corresponds to the flat circular surface at both ends of the cylinder. These surfaces are planar and the determination of the ray

Table 18.4: Different cases arising in 3D, depending on values of  $A, B, C, D$ 

$D$	$C$	$A$	$B$	Shape	$ \mathbf{E} _{min}^2$	$ \mathbf{E} _{max}^2$
0	0	0	0	Circle (radius $ \mathbf{s}_1 $ )	$ \mathbf{T} ^2 +  \mathbf{s}_1 ^2$	$ \mathbf{T} ^2 +  \mathbf{s}_1 ^2$
0	0	0	$\neq 0$	Tilted circle	$ \mathbf{T} - \mathbf{s}_2 ^2$	$ \mathbf{T} + \mathbf{s}_2 ^2$
0	0	$\neq 0$	0	Tilted circle	$ \mathbf{T} - \mathbf{s}_1 ^2$	$ \mathbf{T} + \mathbf{s}_1 ^2$
0	0	$\neq 0$	$\neq 0$	Double-tilted circle	Solve quadratic	
0	$\neq 0$	0	0	Ellipse	$ \mathbf{T} + \mathbf{s}_2 ^2$	$ \mathbf{T} + \mathbf{s}_1 ^2$
0	$\neq 0$	0	$\neq 0$	Tilted ellipse	$ \mathbf{T} - \mathbf{s}_2 ^2$	$ \mathbf{T} + \mathbf{s}_1 ^2$
0	$\neq 0$	$\neq 0$	0	Tilted ellipse	Solve quadratic product	
0	$\neq 0$	$\neq 0$	$\neq 0$	Double-tilted ellipse	Solve quartic	

Table 18.5: Different cases arising in 2D, depending on values of  $A, B, C, D, |\mathbf{s}_1|, |\mathbf{s}_2|$ 

$D$	$C$	$A$	$B$	$ \mathbf{s}_1 $	$ \mathbf{s}_2 $	Shape	$ \mathbf{E} _{min}^2$	$ \mathbf{E} _{max}^2$
0	0	0	0	0	0	Point	$ \mathbf{T} ^2$	$ \mathbf{T} ^2$
0	$\neq 0$	0	$0, \neq 0$	0	$\neq 0$	Line (length $ \mathbf{s}_2 $ )	$ \mathbf{T} - \mathbf{s}_2 ^2$	$ \mathbf{T} + \mathbf{s}_2 ^2$
0	$\neq 0$	$0, \neq 0$	0	$\neq 0$	0	Line (length $ \mathbf{s}_1 $ )	$ \mathbf{T} - \mathbf{s}_1 ^2$	$ \mathbf{T} + \mathbf{s}_1 ^2$
0	0	$0, \neq 0$	$0, \neq 0$	$\neq 0$	$\neq 0$	Circle (radius $ \mathbf{s}_1 $ )	$( \mathbf{T}  -  \mathbf{s}_1 )^2$	$( \mathbf{T}  +  \mathbf{s}_1 )^2$
0	$\neq 0$	0	$\neq 0$	$\neq 0$	$\neq 0$	Ellipse ( $ \mathbf{s}_2  \mathbf{T} $ )	Solve quadratic product	
0	$\neq 0$	$\neq 0$	0	$\neq 0$	$\neq 0$	Ellipse ( $ \mathbf{s}_1  \mathbf{T} $ )	$( \mathbf{T}  -  \mathbf{s}_1 )^2$	$( \mathbf{T}  +  \mathbf{s}_1 )^2$
$0, \neq 0$	$\neq 0$	$\neq 0$	$\neq 0$	$\neq 0$	$\neq 0$	Tilted ellipse	Solve quartic	

intersection position on these surfaces proceeds along the same lines as shown in section 18.4.7.6. The second kind of surface is the cylinder mantle, whose geometrical equation is:

$$\begin{aligned} x^2 + y^2 - D &= 0 \\ (x \ y) \cdot (x \ y) - D &= 0. \end{aligned} \quad (18.138)$$

Inserting the parametric ray line equation 18.112, we obtain a quadratic equation in  $w$ :

$$(\mathbf{R} \cdot \mathbf{R})w^2 + (2\mathbf{R} \cdot \mathbf{R}_L)w + (\mathbf{R}_L \cdot \mathbf{R}_L - D) = 0, \quad (18.139)$$

where only the  $x$  and  $y$  components of the vectors are taken to form the scalar products. Having found the appropriate  $w$  for each ray and the corresponding crossing point  $\mathbf{P}_{cross}$  on the domain cylindrical surface, we need to translate the ray 3D domain crossing coordinates  $P_x, P_y, P_z$  into the initial ray 2D cylindrical wedge coordinates  $W_R, W_y, W_z$ , where  $W_R$  and  $W_z$  correspond to the 2D cylindrical domain and  $W_y$  is the linearized angular coordinate from 18.128. Since the starting angular point is irrelevant (there is no preference of any point on the circle), we set this point equal to zero. Thus:

$$W_R = \sqrt{P_x^2 + P_y^2} \quad (18.140)$$

$$W_y = 0 \quad (18.141)$$

$$W_z = P_z \quad (18.142)$$

To determine the initial ray wedge velocities, we define two unit vectors of the 3D ray vector: 1) the unit vector in the  $(x,y)$ -plane and 2) the unit vector along the  $z$ -axis:

$$\mathbf{u}_{xy} = \mathbf{R}_{xy}/|\mathbf{R}| \quad (18.143)$$

$$\mathbf{u}_z = \mathbf{R}_z/|\mathbf{R}|, \quad (18.144)$$

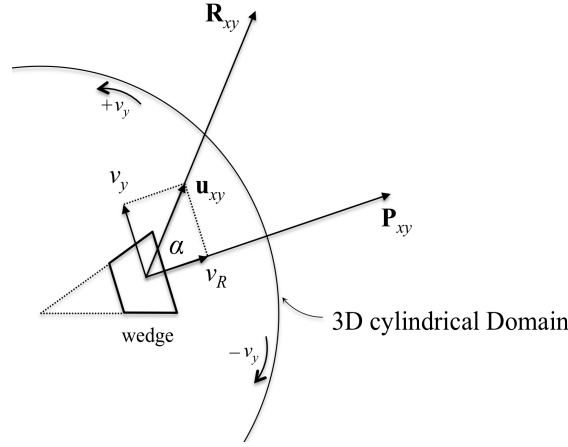


Figure 18.10: Initial radial ( $v_R$ ) and angular ( $v_y$ ) ray velocity components on the wedge from vectors  $\mathbf{P}_{xy}$  and  $\mathbf{R}_{xy}$ .

where  $\mathbf{R}_{xy}$  is the  $(x,y)$ -plane projection of the 3D ray vector. Likewise, the crossing point vector is also split into two components:

$$\mathbf{P}_{cross} = \mathbf{P}_{xy} + \mathbf{P}_z. \quad (18.145)$$

Since the origin of the cylindrical domain is located at  $(x, y) = (0, 0)$ , the vector  $\mathbf{P}_{xy}$  is a radial vector in the  $(x,y)$ -plane. The ray vector component  $\mathbf{R}_{xy}$  on the other hand is usually not radially oriented, because the ray's origin (on the lens) does not necessarily lay on the  $(x, y) = (0, 0)$  line. Both vectors  $\mathbf{P}_{xy}$  and  $\mathbf{R}_{xy}$  are useful in order to determine the ray's initial wedge velocities (see Figure 18.10). Denote the angle between  $\mathbf{R}_{xy}$  and  $\mathbf{P}_{xy}$  by  $\alpha$ . Then

$$\cos(\alpha) = \frac{\mathbf{R}_{xy} \cdot \mathbf{P}_{xy}}{|\mathbf{R}_{xy}| |\mathbf{P}_{xy}|} \quad (18.146)$$

and the initial ray wedge velocities are

$$v_R = v_0 * \cos(\alpha) |\mathbf{u}_{xy}| \quad (18.147)$$

$$v_y = v_0 * \text{sign}(\mathbf{R}_{xy} \times \mathbf{P}_{xy}) * \sin(\alpha) |\mathbf{u}_{xy}| \quad (18.148)$$

$$v_z = v_0 * |\mathbf{u}_z|, \quad (18.149)$$

where the sine and the cross product vector present in the expression for  $v_y$  are calculated as

$$\sin(\alpha) = +\sqrt{1 - \cos^2(\alpha)} \quad (18.150)$$

and

$$\mathbf{R}_{xy} \times \mathbf{P}_{xy} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ R_x & R_y & 0 \\ P_x & P_y & 0 \end{vmatrix} = (R_x P_y - R_y P_x) \mathbf{k}. \quad (18.151)$$

The sign of the cross product refers to the sign of the  $\mathbf{k}$  component. Note, that the wedge's  $y$ -direction refers to the cylindrical  $\theta$ -coordinate and the convention is that counterclockwise rotation along  $\theta$  is considered positive.

#### 18.4.8.5 Tracing the Rays through the Truncated Wedges

When a ray hits the  $R$ - or  $z$ -planes of a truncated wedge, the  $R$ - or  $z$ -components of the ray's position and velocity change the same way as for a pure 2D cylindrical simulation. The  $y$ -components stay the same.

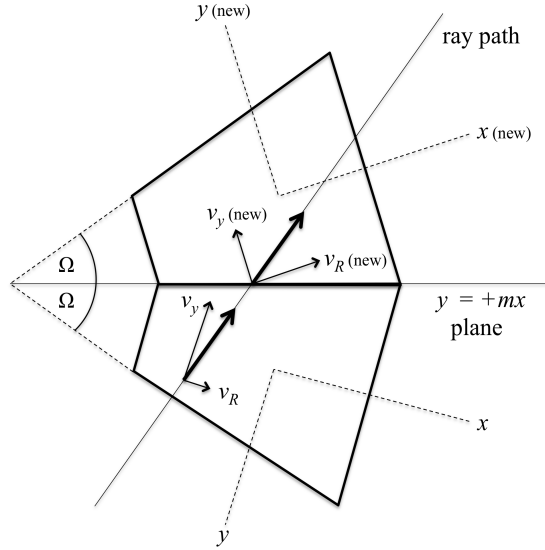


Figure 18.11: Change in position and velocity components when ray moves into a new wedge. For clarity, the velocity components in the original wedge are shifted from the boundary crossing point.

When the ray hits the  $y$ -planes of the truncated wedge, it will cross into a new rotated wedge and the  $y$ -position and the  $R$ - and  $y$ -components of the velocity must change to reflect that rotation. The change in  $y$ -position is simple: just invert its sign. For the velocity component changes we need the cosine and sine of the wedge's opening angle  $\Omega$ . The new components are thus

$$W_y(\text{new}) = -W_y \quad (18.152)$$

$$v_R(\text{new}) = v_R * \cos(\Omega) \pm v_y * \sin(\Omega) \quad (18.153)$$

$$v_y(\text{new}) = v_y * \cos(\Omega) \mp v_R * \sin(\Omega), \quad (18.154)$$

where the sign options in  $v_R$  and  $v_y$  correspond to the two possible  $y$ -planes defined in 18.128 (see Figure 18.11). The change in direction of both velocity vector components is not a rotational effect on the ray due to the medium through which it travels but rather a geometrical effect due to changing wedges. When crossing a common  $y$ -plane between two wedges, the ray is still on the same position in the  $(R, z)$  2D cylindrical plane. Note, that  $\Omega$  must not necessarily be a divisor of  $360^\circ$ . Since  $\Omega$  is fixed through the entire simulation,  $\cos(\Omega)$  and  $\sin(\Omega)$  are conveniently computed only once during initialization.

### 18.4.9 Ray Tracing in 3D Cylindrical Domains

Ray tracing in 3D cylindrical domains starts from the acceleration vector definition at a point  $\mathbf{r}$  in 3D cylindrical coordinates  $\mathbf{r} = (R, \phi, z)$  in terms of their rotating unit vectors  $(\mathbf{R}, \boldsymbol{\phi}, \mathbf{z})$ , the ray equation of motion in Eq.(18.36) and the definition of the gradient operator in 3D cylindrical coordinates:

$$\mathbf{a} = (\ddot{R} - R\dot{\phi}^2)\mathbf{R} + (R\ddot{\phi} + 2\dot{R}\dot{\phi})\boldsymbol{\phi} + \ddot{z}\mathbf{z}, \quad (18.155)$$

the ray equation of motion Eq.(18.36):

$$\mathbf{a} = \nabla \left( -\frac{c^2}{2} \frac{n_e(\mathbf{r})}{n_c} \right) \quad (18.156)$$

and the definition of the gradient operator in 3D cylindrical coordinates:

$$\nabla = \left( \frac{\partial}{\partial R}, \frac{1}{R} \frac{\partial}{\partial \phi}, \frac{\partial}{\partial z} \right) \quad (18.157)$$

The 3D cylindrical ray equation of motion equations become:

$$\ddot{R} - R\dot{\phi}^2 = n_R(\mathbf{r}) \quad (18.158)$$

$$R^2\ddot{\phi} + 2R\dot{R}\dot{\phi} = n_\phi(\mathbf{r}) \quad (18.159)$$

$$\ddot{z} = n_z(\mathbf{r}), \quad (18.160)$$

where  $(\mathbf{r}) = (R, \phi, z)$  and:

$$n_R(\mathbf{r}) = -\frac{c^2}{2n_c} \frac{\partial n_e(\mathbf{r})}{\partial R} \quad (18.161)$$

$$n_\phi(\mathbf{r}) = -\frac{c^2}{2n_c} \frac{\partial n_e(\mathbf{r})}{\partial \phi} \quad (18.162)$$

$$n_z(\mathbf{r}) = -\frac{c^2}{2n_c} \frac{\partial n_e(\mathbf{r})}{\partial z}. \quad (18.163)$$

Since  $\phi$  is given in dimensionless radian units, the units of the partial 3D cylindrical gradients of the electron number density are:  $n_R$  in  $\text{cm/s}^2$ ,  $n_\phi$  in  $\text{cm}^2/\text{s}^2$  and  $n_z$  in  $\text{cm/s}^2$ . The radial acceleration and velocity is  $\ddot{R}$  and  $\dot{R}$  in  $\text{cm/s}^2$  and  $\text{cm/s}$ , the angular acceleration and velocity is  $\ddot{\phi}$  and  $\dot{\phi}$  in  $\text{rad/s}^2$  and  $\text{rad/s}$ .

As the use of angular acceleration  $\ddot{\phi}$  and velocity  $\dot{\phi}$  is computationally somewhat inconvenient, we will transform these to the tangential acceleration  $\dot{v}_\phi$  and velocity  $v_\phi$  by using:

$$v_\phi = \dot{\phi}R \quad (18.164)$$

$$\dot{v}_\phi = \ddot{\phi}R + \dot{\phi}\dot{R} \quad (18.165)$$

Using the tangential variables, Eqs.(18.158-18.160) can be cast into the six 1st order ODEs:

$$\dot{R} = v_R \quad (18.166)$$

$$\dot{\phi} = v_\phi/R \quad (18.167)$$

$$\dot{z} = v_z \quad (18.168)$$

$$\dot{v}_R = n_R(\mathbf{r}) + v_\phi^2/R \quad (18.169)$$

$$\dot{v}_\phi = n_\phi(\mathbf{r})/R - v_\phi v_R/R \quad (18.170)$$

$$\dot{v}_z = n_z(\mathbf{r}). \quad (18.171)$$

Even for constant electron number density gradients, there are no analytical solutions for  $R$  and  $\phi$  in time, and hence one must resort to numerical methods like Runge-Kutta (RK). Starting from initial values  $R, \phi, z, v_R, v_\phi, v_z$  inside a 3D cylindrical cell or on one of its walls, a RK time step is performed such that: 1) we stay inside the cell or hit one of its walls, 2) the RK step is within accuracy bounds and 3) is computationally stable. We would like to have a RK method which allows one to directly calculate the time it would take for one RK step to hit exactly the cell walls. This would avoid an iterative approach to the cell walls, which is computationally very time consuming. The second order RK2 methods are ideally suited for that purpose. A generic RK2 time step  $\Delta t$  on the above set of six 3D cylindrical ODE's gives new updated values (left updated variables, right old variables):

$$R = R + \left(1 - \frac{1}{2\alpha}\right) v_R \Delta t + \frac{1}{2\alpha} v_R^* \Delta t \quad (18.172)$$

$$\phi = \phi + \left(1 - \frac{1}{2\alpha}\right) \left[\frac{v_\phi}{R}\right] \Delta t + \frac{1}{2\alpha} \left[\frac{v_\phi^*}{R^*}\right] \Delta t \quad (18.173)$$

$$z = z + \left(1 - \frac{1}{2\alpha}\right) v_z \Delta t + \frac{1}{2\alpha} v_z^* \Delta t \quad (18.174)$$



and new velocity values:

$$v_R = v_R + \left(1 - \frac{1}{2\alpha}\right) \left[ n_R(\mathbf{r}) + \frac{v_\phi^2}{R} \right] \Delta t + \frac{1}{2\alpha} \left[ n_R(\mathbf{r}^*) + \frac{v_\phi^{*2}}{R^*} \right] \Delta t \quad (18.175)$$

$$v_\phi = v_\phi + \left(1 - \frac{1}{2\alpha}\right) \left[ \frac{n_\phi(\mathbf{r})}{R} - \frac{v_\phi v_R}{R} \right] \Delta t + \frac{1}{2\alpha} \left[ \frac{n_\phi(\mathbf{r}^*)}{R^*} - \frac{v_\phi^* v_R^*}{R^*} \right] \Delta t \quad (18.176)$$

$$v_z = v_z + \left(1 - \frac{1}{2\alpha}\right) n_z(\mathbf{r}) \Delta t + \frac{1}{2\alpha} n_z(\mathbf{r}^*) \Delta t, \quad (18.177)$$

where  $(\mathbf{r}^*) = (R^*, \phi^*, z^*)$  and:

$$R^* = R + \alpha v_R \Delta t \quad (18.178)$$

$$\phi^* = \phi + \alpha [v_\phi / R] \Delta t \quad (18.179)$$

$$z^* = z + \alpha v_z \Delta t \quad (18.180)$$

$$v_R^* = v_R + \alpha [n_R(\mathbf{r}) + v_\phi^2 / R] \Delta t \quad (18.181)$$

$$v_\phi^* = v_\phi + \alpha [n_\phi(\mathbf{r}) / R - v_\phi v_R / R] \Delta t \quad (18.182)$$

$$v_z^* = v_z + \alpha n_z(\mathbf{r}) \Delta t \quad (18.183)$$

are the intermediate state values.  $\alpha$  is a numerical parameter and although it can in principle be set to any value for the generic RK2 scheme to be second order, by far the most common choice is to set  $\alpha$  in the range  $[1/2, 1]$ . Due to the second order nature of the generic RK2 schemes, it allows us to find the  $\Delta t$  values to hit the cell walls by solving simple quadratic equations. Using Eqs.(18.172-18.174) and the intermediate values in Eqs.(18.178-18.183), we have  $\alpha$  independent radial and axial cell wall polynomials in  $\Delta t$  and an  $\alpha$  dependent angular cell wall polynomial:

$$\Delta R = v_R \Delta t + \frac{1}{2} \left[ n_R(\mathbf{r}) + \frac{v_\phi^2}{R} \right] \Delta t^2 \quad (18.184)$$

$$\Delta z = v_z \Delta t + \frac{1}{2} n_z(\mathbf{r}) \Delta t^2 \quad (18.185)$$

$$\Delta \phi = \left[ \frac{v_\phi}{R} - \alpha \frac{v_R \Delta \phi}{R} \right] \Delta t + \left[ (\alpha - 1) \frac{v_\phi v_R}{R^2} + \frac{n_\phi(\mathbf{r})}{2R^2} \right] \Delta t^2, \quad (18.186)$$

where

$$\Delta R = R_{wall} - R \quad (18.187)$$

$$\Delta \phi = \phi_{wall} - \phi \quad (18.188)$$

$$\Delta z = z_{wall} - z \quad (18.189)$$

Note that these quadratic equations do not involve electron number density gradients at the intermediate points, like for example  $n_R(\mathbf{r}^*)$ . Higher order RK methods will introduce these density gradients in expressions for  $R$ ,  $\phi$  and  $z$  thus leading to much more complex equations in  $\Delta t$ . Also cell sizes, especially for highly refined domains, are usually small, hence a higher order RK method, besides having the problem of complicated radial, angular and axial  $\Delta t$  expansions, would involve lots of computational work for a modest gain in number of RK steps inside the cell. Typically one would need only up to three RK2 steps vs maybe one RK4 step inside a cell. The effort of a higher order RK method simply does not pay off.

Let us denote the minimum  $\Delta t$  (excluding zero) obtained from solving all six (two for each kind of cell wall) polynomials in Eqs.(18.184-18.186) as  $\Delta t^{RK2}$ . Using  $\Delta t^{RK2}$  to obtain new positions via Eqs.(18.172-18.174) guarantees that we stay inside the cell, but using it to determine the intermediate state via Eqs.(18.178-18.180) might take us outside of the cell, i.e.  $(R^*, \phi^*, z^*) \notin \text{cell}$ . To enforce the intermediate state inside the cell, we determine also the minimum  $\Delta t$  (excluding zero) obtained from solving all six (two for each kind of cell wall) linear intermediate state equations in Eqs.(18.178-18.180) by replacing  $R^*, \phi^*, z^*$  by the

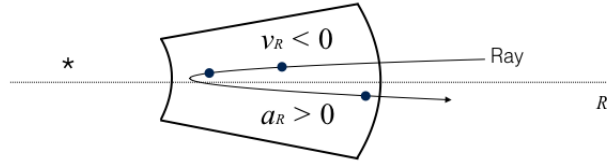


Figure 18.12: Showing a ray turning inside a 3D cylindrical cell due to opposite radial velocity and acceleration direction.

corresponding cell wall values. Denoting this minimum intermediate state linear time as  $\Delta t^*$ , we form the allowed stepping time  $\Delta t_{step}$  inside the cell as:

$$\Delta t_{step} = \min(\Delta t^{RK2}, \Delta t^*). \quad (18.190)$$

$\Delta t_{step}$  might take the ray to a cell wall ( $\Delta t^{RK2} \leq \Delta t^*$ ) or to a position still inside the cell ( $\Delta t^{RK2} > \Delta t^*$ ). The last case happens, when the intermediate state lays outside the cell and is an indication that the ray wants to turn into opposite direction for a particular coordinate (opposite velocity and acceleration direction), as shown in Figure 18.12. Each bullet represents a different stepping time inside the cell. From Eqs.(18.178-18.180) we see, that generally  $\Delta t^*$  will be largest the smaller  $\alpha$  is, hence we choose  $\alpha = 1/2$  (midpoint RK2 method). The following 3D cylindrical ray tracing algorithm has been implemented:

- From a position in or on a cell wall, determine minimum parabolic RK2 time  $\Delta t^{RK2}$  (excluding zero time) to one of the cell walls.
- From same position, determine minimum linear intermediate state time  $\Delta t^*$  (excluding zero time) to one of the cell walls.
- Do a RK2 step (Eqs.(18.172-18.177)) with stepping time  $\Delta t_{step} = \min(\Delta t^{RK2}, \Delta t^*)$  followed by 2 RK2 steps with half stepping time  $\Delta t_{step}/2$ .
- Calculate and analyze errors in position and velocity. If any of the errors is too large, estimate safe stepping time  $\Delta t_{safe}$  and redo the RK2 step. Since  $\Delta t_{safe} \leq \Delta t_{step}$ , we still stay inside the cell.
- If not yet on a cell wall, repeat.

Currently the 3D cylindrical ray tracing has only been implemented for the cell average AVG algorithm (section 18.4.4.1), in which the radial, angular and axial electron number gradients in Eqs.(18.161-18.163) are constant for each cell. Only convex 3D cylindrical domains are treatable in which the angular domain coordinates extend over the full circle. Concave 3D cylindrical domains with partial angular domain coordinates are not planned for the near future. Error bounds in position are defined as a predefined fraction (tolerance) of cell dimensions along each coordinate. Error bounds in velocity are defined as a predefined fraction of the ray speed in the cell before applying the RK2 step. Rather than using the speed  $|v| = \sqrt{v_R^2 + v_\phi^2 + v_z^2}$  itself, we use the sum of the absolute components  $|v_R| + |v_\phi| + |v_z|$ , as this is computationally much more economical to evaluate and is in error compared to the speed by a factor of  $\sqrt{3}$  at most.

Rays crossing the 3D cylindrical axis at  $R = 0$  need a special treatment, as 3D cylindrical blocks touching the axis have no meaningful neighbor block information. To outline the strategy, we consider the 3D cylindrical axis to be hollow and to have a finite tiny radius  $R$  with no radial, angular and axial acceleration terms ( $n_R = n_\phi = n_z = 0$ ). Figure 18.13 shows a ray crossing the hollow axis. If no acceleration terms are present, the 3D cylindrical ray equation of motion ODEs in Eqs.(18.166-18.171) have an exact analytical

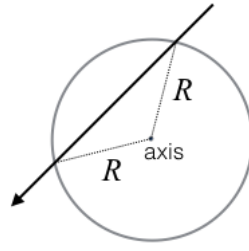


Figure 18.13: Showing a ray passing through a hollow 3D cylindrical axis.

solution:

$$R = \sqrt{(R + v_R t)^2 + v_\phi^2 t^2} \quad (18.191)$$

$$\phi = \phi + \arctan [v_\phi t / (R + v_R t)] \quad (18.192)$$

$$z = z + v_z t \quad (18.193)$$

$$v_R = [(v_R^2 + v_\phi^2)t + R] / \sqrt{(R + v_R t)^2 + v_\phi^2 t^2} \quad (18.194)$$

$$v_\phi = v_\phi R / \sqrt{(R + v_R t)^2 + v_\phi^2 t^2} \quad (18.195)$$

$$v_z = v_z, \quad (18.196)$$

which represents movement along a straight line in 3D cylindrical coordinates. When a ray enters the hollow axis at  $R$ , it must also exit the axis at  $R$ , hence the time it takes to cross the hollow axis is the solution in  $t$  of Eq.(18.191):

$$t = -\frac{2Rv_R}{(v_R^2 + v_\phi^2)}, \quad (18.197)$$

which, when inserted in Eqs.(18.191-18.196) gives:

$$R = R \quad (18.198)$$

$$\phi = \phi + \arctan [2v_R v_\phi / (v_R^2 - v_\phi^2)] \quad (18.199)$$

$$z = z + v_z t \quad (18.200)$$

$$v_R = -v_R \quad (18.201)$$

$$v_\phi = v_\phi \quad (18.202)$$

$$v_z = v_z. \quad (18.203)$$

Taking the limit  $R \rightarrow 0$  in which the hollow axis becomes a line, we have  $t = 0$  and  $z$ ,  $v_\phi$ ,  $v_z$  retain their values while the radial velocity changes sign. The new angle after crossing the axis is only dependent on the initial angle and the radial and tangential angular velocity and given by Eq.(18.199). Figure 18.14 shows various cases of new ray angle values depending on initial radial and tangential angular velocities. Direct application of Eq.(18.199) is, however, not recommended, due to the possibility of zero or near zero of the denominator  $v_R^2 - v_\phi^2$ , leading to computational overflow. We can transform the arctan expression into two equivalent forms by dividing both nominator and denominator by  $v_R^2$  or  $v_\phi^2$  and using the arctan addition formula, giving:

$$\phi = \phi + 2 \arctan [v_\phi / v_R] \quad (18.204)$$

$$\phi = \phi - 2 \arctan [v_R / v_\phi] \quad (18.205)$$

We apply the first equation whenever  $v_R \geq v_\phi$  and the second otherwise, thus ensuring an arctan value  $\leq 1$  for all cases.

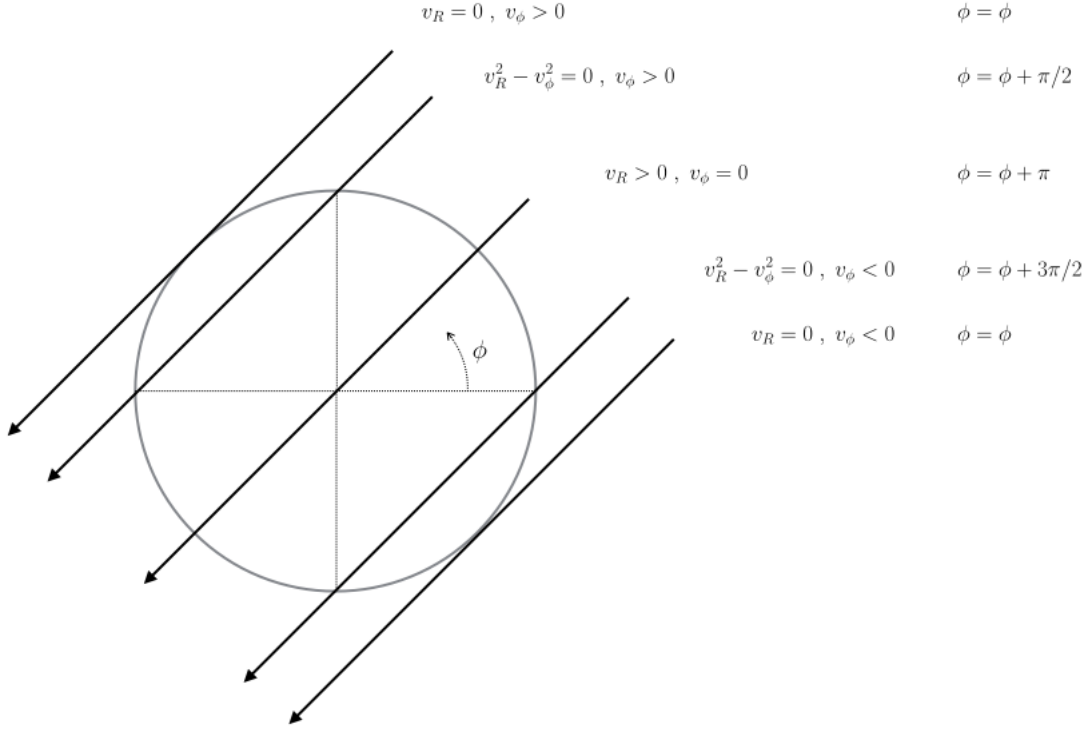


Figure 18.14: Various examples of new ray angle values depending on initial radial and tangential angular velocities.

#### 18.4.10 Synchronous and Asynchronous Ray Tracing

The default laser implementation uses the Lagrangian particle framework to move rays (modeled as particles) across block and process boundaries. We generally find that particle movement consumes a relatively small portion of total runtime even in large-scale simulations with millions of particles (Dubey et al., 2012). This is not always case in laser-driven FLASH simulations, where realistic simulations can spend the majority of runtime moving rays between blocks. The reason for the discrepancy in performance is because a ray generally travels a much larger distance than a tracer particle in a single time-step. In the tracer particle case we only need one application of the communication subroutines in a single time-step because a particle never moves further than a nearest-neighbor block (see CFL limit). In contrast a ray can move to the other side of the domain in a single time-step which may involve visiting hundreds of intermediate blocks.

There are various implementations of particle exchange in the GridParticles sub-unit. In order to understand the performance bottlenecks, we describe the default implementation used in FLASH applications configured with Paramesh. The particle movement communication consists of an `MPI.Allreduce` to determine the number of particles and messages that each MPI rank will receive, followed by point to point messages which exchange the actual particles. In addition to the communication in the GridParticles unit, there is also an `MPI.Allreduce` in the laser unit which determines if all the rays have either left the domain or have been absorbed. This is the termination criteria and controls whether another round of particle exchange is needed.

Performance profiles of simulations using the laser indicate that most time is spent in `MPI.Allreduce` calls in a particle movement subroutine. The trace in Figure 18.15 shows that this happens because of load imbalance and not slowness of the underlying collective communication call. The two main colors in this trace are red and purple; red is time spent in a computational kernel named `ed_traceBlockRays2Dcy13D` and purple is time spent in `MPI.Allreduce`. We see that the color map is dominated by purple which indicates most MPI ranks are waiting for the `MPI.Allreduce` to complete. The cascading red pattern shows the ray paths as rays move between blocks assigned to different MPI ranks. It is clear that there are many round

of bulk synchronous particle exchange and that some MPI ranks must wait a long time before receiving any rays. The biggest impediment to good performance is the dependency between MPI ranks which happens

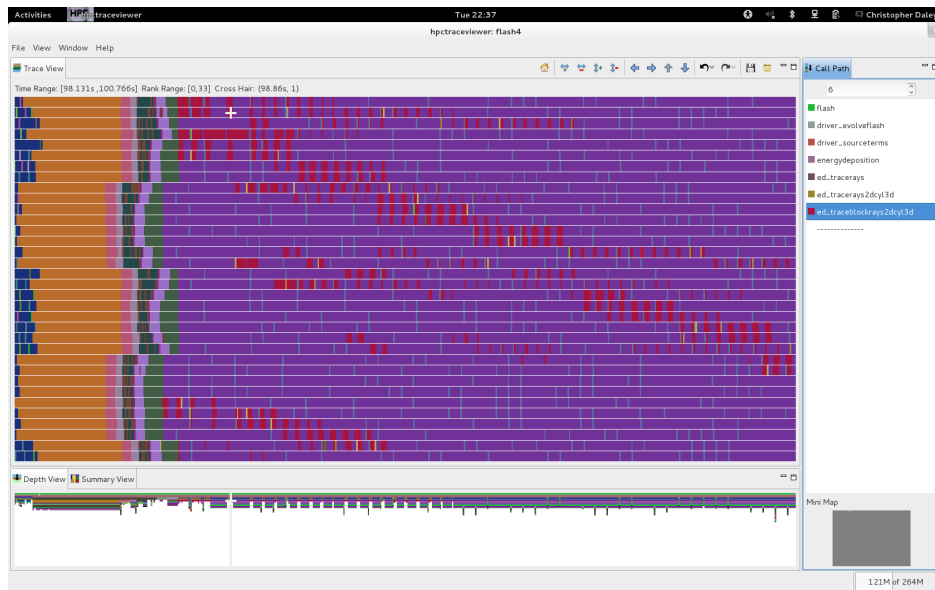


Figure 18.15: Performance trace of MPI ranks 0-33 in a laser simulation using synchronous communication and run with 2048 MPI ranks on Intrepid BG/P.

because ray paths are not known ahead of time. Performance quickly degrades at higher processor counts because increased domain decomposition means rays must cross more process boundaries. One approach to improve performance would be to use the mesh replication feature of FLASH to trivially parallelize over the rays, but this is not ideal because it introduces redundant computations of e.g. hydrodynamics in each mesh. The approach we take is to introduce pipelined parallelism to hide the dependency between MPI ranks; rays are sent to nearest-neighbor blocks on different MPI ranks as early as possible to minimize the time MPI ranks must wait for work. Computation and communication are overlapped, which is different to the current bulk synchronous approach where there are distinct computation and communication phases. The “asynchronous ray trace” is included in a FLASH application by setting up with the shortcut `+asyncLaser`. We describe the ray exchange and termination communication kernels in the asynchronous scheme below.

The ray exchange communication kernel requires all MPI ranks set up designated send and receive communication channels with only those MPI ranks which own nearest-neighbor blocks. This is a relatively small subset because of the block locality in Paramesh. We post one speculative non-blocking receive for each neighbor and then test all communication channels (at a rate determined by runtime parameter `ed.commRaysBetweenMsgTest`) for new rays. When an MPI rank receives a message we copy the rays and then post a new speculative non-blocking receive. This ensures that a message can be received from any nearest neighbor at any time. We send rays using non-blocking sends from source MPI ranks when send buffers (of size determined by runtime parameter `ed.commChannelSize`) are full. At idle phases, we also send rays before the send buffer is full to prevent deadlock and ensure good global progress. When exchange is complete all MPI ranks send a zero byte message to each neighboring MPI rank to match the remaining speculative receives.

The termination communication kernel must also be non-blocking to allow progress in the ray exchange communication kernel. This rules out using a blocking `MPI_Allreduce` as is used in the default synchronous scheme. We have multiple non-blocking implementations of the termination kernel which all have different library and MPI dependencies. The default implementation, which has no special dependencies, uses a master-slave approach to determine completion. It works by sending a count of local inactive rays from slaves to a master which maintains a count of global inactive rays. We send a notification message from the master to the slaves when the global number of inactive rays is equal to the initial count of active rays. Even

though we have demonstrated that this implementation *can* work when using 8192 MPI ranks on Intrepid BG/P, there is a danger of exceeding internal MPI limits on the number of messages the master rank can receive. For this reason we have created another implementation which uses a non-blocking `MPI_Iallreduce` which is part of MPI-3. We recommend using this implementation because there is less danger of exceeding internal MPI limits. It can be used by setting up a FLASH application with the setup shortcut `+mpi3`. Given that the MPI-3 standard is still quite new, it is likely that some MPI implementations on today's production machines do not support non-blocking collectives. Therefore, we also support the non-blocking reduction in the Non Blocking Collectives (NBC) library <http://hpcr.inf.ethz.ch/research/nbcoll/libnbc/>. The necessary reduction name substitutions happen when a FLASH application is setup with `+libnbc`.

We repeat the experiment in Figure 18.15 with the asynchronous communication scheme and show results in Figure 18.16. The number of ranks on the y-axis and time scale on the x-axis are the same as before. The color map is much more chaotic than before because all MPI ranks are either computing or spinning in various subroutines looking for more work. As before, the red color indicates time spent in a computational kernel named `ed_traceBlockRays2Dcy13D`. The red is now much more clustered on the left side of the figure indicating MPI ranks wait less time for rays than before. We see large segments of green for MPI rank 0 which indicates time spent testing for new messages. This is happening because we made use of the master-slave termination kernel in this test. A performance comparison of the synchronous and asynchronous communication schemes is shown in Figure 18.17. The parameters `ed_commChannelSize` and



Figure 18.16: Performance trace of MPI ranks 0-33 in a laser simulation using asynchronous communication and run with 2048 MPI ranks on Intrepid BG/P.

`ed_commRaysBetweenMsgTest` give some control over the memory overhead and performance of the asynchronous scheme. The `ed_commChannelSize` parameter should be kept relatively small because it determines the amount of space available for rays in send and receive buffers. Large values lead to a very high memory footprint because there is a designated portion of space in the buffers for all communication channels (i.e. all possible pairs of communicating processors) and there can be 10-100s of communication channels. The parameter also determines the number of rays which are buffered before being sent from source processors to destination processors, and so smaller values will improve overall load balance. All messages are non-blocking and so the source processor returns immediately to computation and is free to send rays to different destination processors. However, the source processor will block (actually spin on various `MPI_Test` statements with no danger of deadlock) if a ray should be sent to the original processor and the original message is not yet delivered. This means it is important for the destination processor to frequently check for new messages not only to start work as soon as possible but also to enable the source processor to return to useful work. The

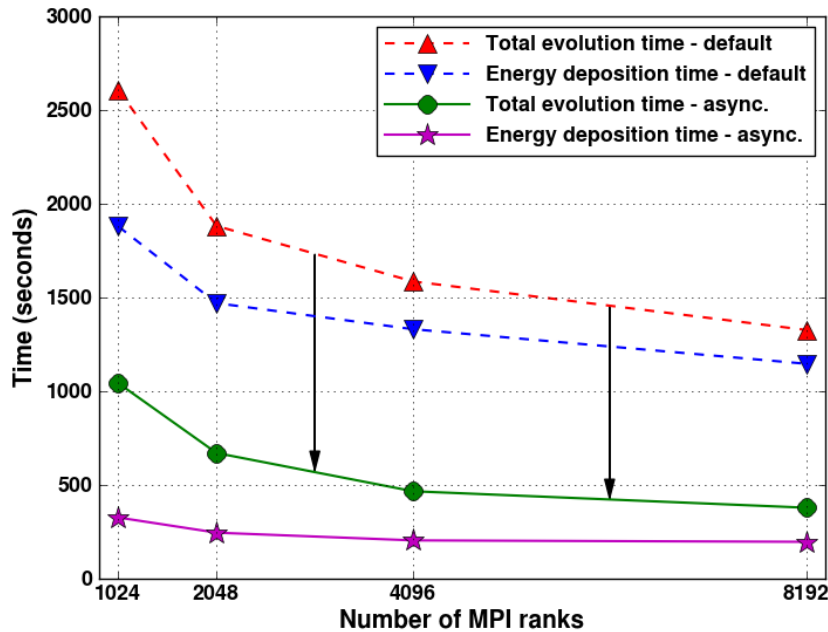


Figure 18.17: Strong scaling of a laser simulation run on Intrepid BG/P with the synchronous and asynchronous communication scheme.

parameter `ed_commRaysBetweenMsgTest` should therefore be kept relatively small so that the destination processor performs less computational work between message checks.

We set the default values of `ed_commRaysBetweenMsgTest=50` and `ed_commChannelSize=100`. Very small values for both parameters improve load balance but introduce new overheads such as the cost of continually testing for new messages and the cost of preparing the laser computation routines for the newly received rays. The preparation work includes sorting the rays in block order and then scanning the ray array to find the start and end index for each block. Sorting improves the memory-access pattern and is a sensible optimization when computational time is large relative to the sort time (as always happens in the original synchronous communication scheme), however, it hurts performance when computational work is very fine-grained. Hence, we do not advise using parameter values much smaller than the default values. In future it would be interesting to see if removing the serial optimization improves the global time to solution. This could happen because more processors are doing useful work despite there being worse memory-access on each processor.

The asynchronous communication scheme conflicts with various capabilities in FLASH. It only works with Paramesh at the current time, although support for UG could probably be added without a huge amount of effort. We initially did not support UG because we did not know the MPI rank IDs of the corner neighboring blocks, although this is no longer the case. It also conflicts with the Laser I/O.

The asynchronous communication scheme is able to run without the GridParticles dependency by setting up with `useGridParticles=False`. This is generally a good idea because the GridParticles sub-unit has initialization code which allocates large send and receive buffers for bulk synchronous communication. These buffers are not needed in the asynchronous scheme and so removing the dependency significantly reduces the memory footprint. The only down-side is that a slower sorting subroutine will be used in place of `Grid.SortParticles`, which is able to achieve a fast sort by using the space in the bulk synchronous communication buffers.

The new asynchronous ray tracing implementation works and has been tested with several laser problems using multiple compilers and MPI implementations. However, since this is a rather new implementation, the asynchronous ray tracing mode should still be considered as being in its experimental stage. Both



synchronous and asynchronous ray tracing modes are currently made available, with synchronous mode being the default. The asynchronous mode can be activated by using the appropriate setup parameter.

### 18.4.11 Usage

To include the use of the Energy Deposition unit, the following should be included into the setup line command:

```
+laser ed_maxPulses=<number> ed_maxPulseSections=<number> ed_maxBeams=<number>
```

The +laser is a shortcut that handles all the logistics for properly including the `EnergyDeposition` unit. By default, the ray tracing algorithm used is the AVG algorithm 18.4.4.1 in its synchronous communication mode 18.4.10. To activate the cubic interpolation ray tracing schemes, replace the +laser shortcut by the shortcut +laserCubicInterpolation. This automatically includes the relevant routines for cubic interpolation and bypasses the ones responsible for the AVG algorithm. The default ray tracing method for the cubic interpolation is the piecewise parabolic ray tracing method 18.4.4.2. To activate the cubic interpolation using the Runge Kutta integrator one has to set the shortcut +laserCubicInterpolationRK. To activate the asynchronous ray tracing communication mode, replace the +laser shortcut by the shortcut +asynchLaser. This enables the asynchronous communication routines and suppresses the synchronous ones. Only the AVG ray tracing algorithm is currently implemented in asynchronous communication mode. The other three setup variables fix the dimensions needed for the pulses and beams:

- **ed\_maxPulses:** The maximum number of different laser pulses for the simulation.
- **ed\_maxPulseSections:** The maximum number of power/time pairs per pulse.
- **ed\_maxBeams:** The maximum number laser beams for the simulation.

The Energy Deposition unit reads all the information it needs to construct the laser beams and pulses from runtime parameters specified in the flash.par file. Below is the list of runtime parameters that is needed to properly build the laser. For clarification and figuring out the input to a particular laser simulation, the reader is encouraged to have Figures 18.5, 18.6 and 18.7 at hand.

#### 18.4.11.1 Laser Pulses Runtime Parameters

- **ed\_numberOfSections\_n:** Indicates the number of power/time pairs that are going to be used to set up the shape of the n-th laser pulse. There must be at least as many of these runtime parameters as there are number of laser pulses defined, i.e.  $n = 1, \dots, \text{ed\_numberOfPulses}$ .
- **ed\_power\_n\_i:** Sets the i-th power of the i-th power/time pair of the n-th laser pulse. The ranges of the indices must be at least:  $i = 1, \dots, \text{ed\_numberOfSections\_n}$  and  $n = 1, \dots, \text{ed\_numberOfPulses}$ .
- **ed\_time\_n\_i:** Sets the i-th time of the i-th power/time pair of the n-th laser pulse. The ranges of the indices must be at least:  $i = 1, \dots, \text{ed\_numberOfSections\_n}$  and  $n = 1, \dots, \text{ed\_numberOfPulses}$ .

#### 18.4.11.2 Laser Beams Runtime Parameters

- **ed\_adjustInitialRaySpeed:** If true, the initial ray speed of the speed of light will be adjusted (lowered) by a factor of  $\sqrt{1 - n_e/n_c}$  when the ray enters the domain depending on the electron number density  $n_e$  encountered in the first entry cell (n-th beam).
- **ed\_crossSectionFunctionType\_n:** Beam cross section power function (flat or gaussian decay) type. For a flat profile use 'uniform', for gaussian decay use 'gaussian1D' (two-dimensional beams) or 'gaussian2D' (three-dimensional beams) (n-th beam).
- **ed\_gaussianCenterMajor\_n:** The Gaussian center location along the major elliptical semiaxis (n-th beam).



- **ed\_gaussianCenterMinor\_n**: The Gaussian center location along the minor elliptical semiaxis (n-th beam).
- **ed\_gaussianExponent\_n**: The Gaussian super exponent  $\gamma$  for the beam cross section power function in equation 18.106 (n-th beam).
- **ed\_gaussianRadiusMajor\_n**: The Gaussian radius (e-folding length)  $R_x$  along the major elliptical semiaxis in equation 18.106 (n-th beam).
- **ed\_gaussianRadiusMinor\_n**: The Gaussian radius (e-folding length)  $R_y$  along the minor elliptical semiaxis in equation 18.106 (n-th beam).
- **ed\_gridDeltaSemiAxisMajor\_n**: For delta grid types, the tic separation along the major elliptical semiaxis (n-th beam).
- **ed\_gridDeltaSemiAxisMinor\_n**: For delta grid types, the tic separation along the minor elliptical semiaxis (n-th beam).
- **ed\_gridnAngularTics\_n**: For radial grid types, the number of wanted angular slices (n-th beam).
- **ed\_gridnRadialTics\_n**: For radial grid types, the number of wanted grid positions along each radial spike (n-th beam).
- **ed\_gridnSemiAxisMajorTics\_n**: For rectangular grid types, the number of wanted grid positions along the major semi axis (n-th beam).
- **ed\_gridnSemiAxisMinorTics\_n**: For rectangular grid types, the number of wanted grid positions along the minor semi axis (n-th beam).
- **ed\_gridType\_n**: Specifies the grid type to be used for placing the rays inside the beam. For two-dimensional beams the options are 'regular1D' or 'statistical1D'. For three-dimensional beams the options are 'square2D', 'radial2D' or 'statistical2D'. (n-th beam).
- **ed\_ignoreBoundaryCondition\_n**: Ignore the domain boundary conditions (reflective) when rays enter the domain (n-th beam)?
- **ed\_lensSemiAxisMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the elliptical lens area (n-th beam).
- **ed\_lensX\_n**: The x-component of the global lens center position vector  $\mathbf{L}$  (n-th beam).
- **ed\_lensY\_n**: The y-component of the global lens center position vector  $\mathbf{L}$  (n-th beam).
- **ed\_lensZ\_n**: The z-component of the global lens center position vector  $\mathbf{L}$  (n-th beam).
- **ed\_numberOfRays\_n**: Number of rays to be created for the beam. Might be overwritten in 3D geometrical cases (n-th beam).
- **ed\_pulseNumber\_n**: The pulse shape identification number (n-th beam).
- **ed\_semiAxisMajorTorsionAngle\_n**: The major elliptical semiaxis torsion angle  $\phi_1$  along the beam's lens target center line (n-th beam).
- **ed\_semiAxisMajorTorsionAxis\_n**: The major elliptical semiaxis torsion axis ('x', 'y' or 'z') from which the torsion angle  $\phi_1$  is defined (n-th beam).
- **ed\_targetSemiAxisMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the elliptical target area (n-th beam).
- **ed\_targetSemiAxisMinor\_n**: The minor (smallest) semiaxis length  $\ell_2$  for the elliptical target area (n-th beam).

- **ed\_targetX\_n**: The x-component of the global target center position vector  $\mathbf{T}$  (n-th beam).
- **ed\_targetY\_n**: The y-component of the global target center position vector  $\mathbf{T}$  (n-th beam).
- **ed\_targetZ\_n**: The z-component of the global target center position vector  $\mathbf{T}$  (n-th beam).
- **ed\_wavelength\_n**: The wavelength of the laser (n-th beam).

#### 18.4.11.3 Laser General Runtime Parameters

- **ed\_adjustBeamsTargetIntensity**: If true, 1D cartesian and 2D cartesian beams target intensities (power/area) will be adjusted to mimic a cylindrical 3D beam with radius equal to **ed\_targetSemiAxisMajor**.
- **ed\_adjustBySymmetryX**: If true and **ed\_createRaysExpandX** is also true and one of the x boundary is reflecting and another x boundary is not reflecting, multiply ray power and ray count by half to account for double counting in x direction.
- **ed\_adjustBySymmetryY**: If true and **ed\_createRaysExpandY** is also true and one of the y boundary is reflecting and another y boundary is not reflecting, multiply ray power and ray count by half to account for double counting in y direction.
- **ed\_adjustBySymmetryZ**: If true and **ed\_createRaysExpandZ** is also true and one of the z boundary is reflecting and another z boundary is not reflecting, multiply ray power and ray count by half to account for double counting in z direction.
- **ed\_beamsCheckExpandX**: If true, check beams in both computational domain and expanded (by reflection or translation in x direction) domain.
- **ed\_beamsCheckExpandY**: If true, check beams in both computational domain and expanded (by reflection or translation in y direction) domain.
- **ed\_beamsCheckExpandZ**: If true, check beams in both computational domain and expanded (by reflection or translation in z direction) domain.
- **ed\_cellStepTolerance**: This factor times the smallest dimension of each cell is taken as the positional error tolerance for step-sized based ray tracing methods.
- **ed\_cellTimeEnergyDeposition**: If set true, the energy deposition by the laser rays is calculated based only on the time spent in each cell by calculating cell center values of  $\ln \Lambda$  and  $\nu_{ib}$  using Eq.(18.46) and replacing the time integration term in Eq.(18.47) by  $\nu_{ibt}$ , i.e. treating  $\nu_{ib}$  as constant in the cell. If false, the more expensive time integration formalism from Eq.(18.59) is used.
- **ed\_cellWallThicknessFactor**: Controls the (imaginary) thickness of the cell walls to ensure computational stability of the laser code. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to  $10^{-6}$  and should only very rarely be changed.
- **ed\_computeGradNeleR**: (AVG algorithm 18.4.4.1 only) If false, the radial components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.
- **ed\_computeGradNeleP**: (AVG algorithm 18.4.4.1 only) If false, the azimuthal (phi) components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.
- **ed\_computeGradNeleT**: (AVG algorithm 18.4.4.1 only) If false, the polar (theta) components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.
- **ed\_computeGradNeleX**: (AVG algorithm 18.4.4.1 only) If false, the x-components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.
- **ed\_computeGradNeleY**: (AVG algorithm 18.4.4.1 only) If false, the y-components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.

- **ed\_computeGradNeleZ**: (AVG algorithm 18.4.4.1 only) If false, the z-components of the gradients  $\langle \nabla n_e \rangle$  are not computed, i.e. set to zero.
- **ed\_createRaysExpandX**: If true, create rays in both computational domain and expanded (by reflection or translation in x direction) domain. If the created ray is in the expanded portion, then map the ray to the computational domain.
- **ed\_createRaysExpandY**: If true, create rays in both computational domain and expanded (by reflection or translation in y direction) domain. If the created ray is in the expanded portion, then map the ray to the computational domain.
- **ed\_createRaysExpandZ**: If true, create rays in both computational domain and expanded (by reflection or translation in z direction) domain. If the created ray is in the expanded portion, then map the ray to the computational domain.
- **ed\_cubicInterpolationZeroDerv**: If true, all cubic interpolation vertex derivatives will be set equal to zero, enforcing monotonicity for interpolation between all vertices of a cell.
- **ed\_enforcePositiveNele**: (AVG algorithm 18.4.4.1 only) If true, the x-, y- and z-components of the gradients  $\langle \nabla n_e \rangle$  will be rescaled such that they always deliver a positive (greater or equal zero) value for the number of electrons in a cell.
- **ed\_enforcePositiveTele**: (AVG algorithm 18.4.4.1 only) If true, the x-, y- and z-components of the gradients  $\langle \nabla T_e \rangle$  will be rescaled such that they always deliver a positive (greater or equal zero) value for the electron temperature in a cell.
- **ed\_gradOrder**: (AVG algorithm only) The order of approximation used for the electron number density  $n_e$  and the electron temperature  $T_e$  in a cell (equation 18.39). A value of 1 leads to linear and a value of 2 to parabolic (quadratic) ray trajectories inside the cell. The first case includes no gradients and only takes the cell's average value.
- **ed\_laser3Din2D**: If true, 3D rays will be traced on a 2D cylindrical grid.
- **ed\_laser3Din2DwedgeAngle**: The wedge opening angle  $\Omega$  (in degrees) for a 3D in 2D laser ray trace simulation.
- **ed\_maxRayCount**: The maximum number of rays that can be created on one processor.
- **ed\_numberOfBeams**: The number of laser beams that are going to be used.
- **ed\_numberOfPulses**: Controls the number of different laser pulses that are going to be used.
- **ed\_powerStepTolerance**: (Cubic interpolation with Runge Kutta only) This factor denotes the fractional error (unit = current power) for a Runge Kutta step in the CIRK 18.4.4.3 ray tracing method.
- **ed\_printBeams**: If true, it prints detailed information about the laser beams to a file with name `<basename>LaserBeamsPrint.txt`, where `<basename>` is the base name of the simulation.
- **ed\_printMain**: If true, it prints general information regarding the laser setup to a file with name `<basename>LaserMainDataPrint.txt`, where `<basename>` is the base name of the simulation.
- **ed\_printPulses**: If true, it prints detailed information about the laser pulses to a file with name `<basename>LaserPulsesPrint.txt`, where `<basename>` is the base name of the simulation.
- **ed\_printRays**: If true, it prints detailed information about the all rays initially generated on each processor to a file(s) with name(s) `<basename>LaserRaysPrint<PID>.txt`, where `<basename>` is the base name of the simulation and PID is the processor rank number.
- **ed\_rayZeroPower**: Below this value (erg/s), the ray is considered to have zero power.

- **ed\_RungeKuttaMethod**: (Cubic interpolation with Runge Kutta only) Specifies which Runge Kutta method to use for the CIRK 18.4.4.3 ray tracing method. Current options are: 'CashKarp45' (order 4, default), 'EulerHeu12' (order 1), 'BogShamp23' (order 2), 'Fehlberg34' (order 3) and 'Fehlberg45' (order 4).
- **ed\_saveOutOfDomainRays**: If true, the rays info will be stored into a separate saved ray array for those rays that exit the domain.
- **ed\_useRayCoords2GetBlockID**: If true, the Grid Particles unit uses ray coordinates to determine the new block ID when a ray exits a block. Otherwise block neighbor info is used to determine the new block ID.
- **ed\_velocityStepTolerance**: This factor times the current speed of a ray in a cell is taken as the velocity component error tolerance for step-sized based ray tracing methods.
- **useEnergyDeposition**: If false, the energy deposition is not activated, even if the code was compiled to do so. Bypasses the need to rebuild the code.
- **threadRayTrace**: If true, the innermost ray loop, tracing all rays through a block, is threaded. This runtime parameter can only be set during setup of the code.

### Simulating 3D Cylindrical Laser Beam Intensities using 1D/2D Cartesian Grids

Per implementation, in FLASH the concept of a cell volume for lower dimensional grids consists in setting a unit length for each missing dimension. For example, in 1D cartesian, if a 1D cell has length  $\Delta X$  cm, then the corresponding cell volume as used in FLASH would be:  $\Delta X$  cm x 1 cm x 1 cm =  $\Delta X$  cm<sup>3</sup>. This results in different beam intensities when using the same beam power for 1D/2D/3D cartesian laser simulations. To make these simulations comparable with equal beam intensities, one can either adjust the individual beam powers or one can use the same power and set the runtime parameter **ed\_adjustBeamsTargetIntensity** equal to true. The latter will then internally determine the adjustment factor to be used for 1D/2D cartesian beams to emulate a 3D cylindrical laser beam with radius equal to the beam's **ed\_targetSemiAxisMajor**. In case one wants to adjust the 1D/2D cartesian beam powers ( $P_{1D}, P_{2D}$ ), the following conversions should be used:

$$P_{1D}[W/cm^2] = P_{3D}[W]/\pi r^2 \quad (18.206)$$

$$P_{2D}[W/cm] = 2P_{3D}[W]/\pi r, \quad (18.207)$$

where  $r$  is the beam's radius (**ed\_targetSemiAxisMajor**).

### Simulating Axisymmetric Cylindrical Laser Beam Intensities using 1D Cylindrical Grids

For these kind of simulations, the user needs to adjust the 1D cylindrical beam power by a factor that mimics an axisymmetrical cylindrical laser beam. The beams **ed\_targetSemiAxisMajor** is now irrelevant, but instead the radial position of the target and the laser intensity on target ( $I$ ) should be considered. The 1D laser cylindrical beam power ( $P_{1D}^{cyl}$ ) should be given as:

$$P_{1D}^{cyl}[W/cm] = 2\pi r_t I[W/cm^2], \quad (18.208)$$

where  $r_t$  is the radial position of the target **ed\_targetX**.

#### 18.4.11.4 LaserIO Runtime Parameters and Usage

Often times it is useful to be able to visualize the paths of the individual rays in a simulation. This can be very helpful in ensuring that the laser beams are set up in the intended manner. The LaserIO directory in the source tree located at:

source/physics/sourceTerms/EnergyDeposition/EnergyDepositionMain/Laser/LaserIO

By default, this unit is requested by the Laser package (for example, when the `+laser` setup shortcut is specified).

The LaserIO package gives users the ability to write the trajectory of a certain number of rays to plot files when normal FLASH plot file writes occur. This feature is only compatible with parallel HDF5 IO implementations (for example, with `+parallelIO` or `+hdf5typeio`). Only three runtime parameter options need to be set to use LaserIO. These are:

- **ed\_useLaserIO**: Set to `.true.` to activate LaserIO
- **ed\_laserIOMaxNumberOfRays**: Sets the *approximate* maximum number of rays to write to the plot file. This number can be less than the total number of rays in the simulation. When greater than the maximum number of rays, every ray trajectory is written.
- **ed\_laserIOMaxNumberOfPositions**: Sets the size of the LaserIO buffer. A good estimate of this value is  $5 \times \text{NXB} \times \text{NYB} \times \text{NZB} \times \text{ed\_laserIOMaxNumberOfRays}$ .

The `ed_laserIOMaxNumberOfRays` parameter sets the approximate maximum number of rays to write out. The exact number of rays that will be written is equal to:

$$\max \left[ \text{int} \left( \frac{N_{\text{rays}}}{\text{ed\_laserIOMaxNumberOfRays}} \right), 1 \right],$$

where  $N_{\text{rays}}$  is the sum of the number of rays to launch for each beam. The LaserIO ray information is written to the normal HDF5 plot files in the `RayData` dataset. The `extract_rays.py` script that is distributed with FLASH in the `tools/scripts` directory can be used to extract this data. This is a python script which requires the NumPy and PyTables python packages to operate. The script takes as arguments a list of plot files. For each plot file, the script will generate a corresponding VTK file which can be loaded into VisIt. The `RayPower_Watts` Pseudocolor in VisIt can be plotted to show the ray trajectories. The rays are colored based on their power (in watts). See the `LaserSlab` simulation [Section 35.7.5](#) for an example which uses LaserIO.

#### 18.4.11.5 Laser Energy Density Output

Laser energy density (energy per volume – see [18.4.3](#)) is very useful for visualizing lasers in the simulation space and can be output to the variable “lase” in checkpoint and plot files. This is an experimental feature in FLASH 4.4 and is implemented **only for 3D Cartesian geometry**. To enable “lase” output, edit your simulation’s Config file (*e.g.*, for the `LaserSlab` example, `Simulation/SimulationMain/LaserSlab/Config`) to include the line:

```
VARIABLE lase TYPE: PER_VOLUME
```

Then setup and build FLASH to use 3D Cartesian ray tracing (otherwise, the added variable is ignored).

### 18.4.12 Unit Tests for 3D/2D Cartesian Domain Geometries

These unit tests set up a series of beams that launch rays onto a domain with a particular electron number density and electron temperature distribution in such a way that an analytical solution is possible for the ray paths and their power deposited inside the domain. The tests are 3D cartesian extensions to the 2D analytical test solution for the quadratic trough as presented in the paper by Kaiser (2000). [Figure 18.18](#) shows the basic structure. 2D cartesian versions of the tests is also available.

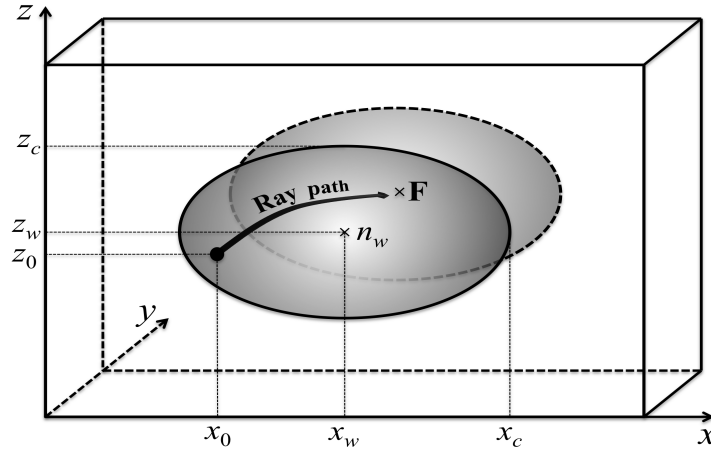


Figure 18.18: One ray moving through an ellipsoidal tube with a quadratic electron number density profile and reaching a focal point  $\mathbf{F}$ .

#### 18.4.12.1 Analytic Path Solution for the Ellipsoidal Quadratic Potential Tube

Let us define a quadratic ellipsoidal electron number density profile in a 3D space along the  $xz$ -plane and having no variation along the  $y$ -axis:

$$n_e(\mathbf{r}) = n_w + A(x - x_w)^2 + B(z - z_w)^2, \quad (18.209)$$

Here  $x_w$  and  $z_w$  are the center coordinates of the ellipsoidal tube cross section,  $n_w$  is the value of the electron number density at the center of the ellipse and  $A, B$  are the scaling factors for the individual components. The values of  $A$  and  $B$  are determined by the boundary conditions of  $n_e(\mathbf{r})$ . For example, if one wants  $n_e(\mathbf{r})$  to be equal to a critical value  $n_c$  at specific coordinates  $x_c$  and (separately)  $z_c$ , then we have:

$$A = (n_c - n_w)/(x_c - x_w)^2 \quad (18.210)$$

$$B = (n_c - n_w)/(z_c - z_w)^2. \quad (18.211)$$

Using the ray equation of motion

$$\frac{d^2 \mathbf{r}}{dt^2} = \nabla \left( -\frac{c^2}{2} \frac{n_e(\mathbf{r})}{n_c} \right), \quad (18.212)$$

we can solve for the position vector  $\mathbf{r}$  under several circumstances (boundary conditions). The first thing to note is that  $n_e(\mathbf{r})$  does not contain mixed variable terms, hence the differential ray equation is separable into the individual components. Let us take the  $x$ -component. We have

$$\begin{aligned} \frac{d^2 x}{dt^2} &= -\frac{c^2}{2n_c} \frac{d}{dx} A(x - x_w)^2 \\ &= -k(x - x_w), \end{aligned} \quad (18.213)$$

where  $k = c^2 A/n_c$ . This is the differential equation for a simple harmonic oscillator around the point  $x_w$ . The boundary conditions are such that at time  $t = 0$  the ray will be at an initial position  $x_0$  and will possess a certain velocity component  $v_x$

$$x = x_0 \text{ (at } t = 0) \quad (18.214)$$

$$\frac{dx}{dt} = v_x \text{ (at } t = 0). \quad (18.215)$$

To solve the simple harmonic oscillator equation, we make the ansatz

$$x(t) = x_w + (x_0 - x_w) \cos(\beta t) + b \sin(\beta t) \quad (18.216)$$

and find the expressions for  $b$  and  $\beta$  satisfying the boundary conditions. We have

$$\frac{dx}{dt} = -\beta(x_0 - x_w) \sin(\beta t) + b\beta \cos(\beta t) \quad (18.217)$$

$$\begin{aligned} \frac{d^2x}{dt^2} &= -\beta^2(x_0 - x_w) \cos(\beta t) - b\beta^2 \sin(\beta t) \\ &= -\beta^2(x - x_w) \end{aligned} \quad (18.218)$$

from which we deduce that

$$\beta = \sqrt{k} \quad (18.219)$$

$$b = \frac{v_x}{\beta}. \quad (18.220)$$

The solution to the complete ray equation can thus be stated as

$$x(t) = x_w + (x_0 - x_w) \cos(\sqrt{k}t) + \frac{v_x}{\sqrt{k}} \sin(\sqrt{k}t) \quad (18.221)$$

$$y(t) = y_0 + v_y t \quad (18.222)$$

$$z(t) = z_w + (z_0 - z_w) \cos(\sqrt{k'}t) + \frac{v_z}{\sqrt{k'}} \sin(\sqrt{k'}t), \quad (18.223)$$

where  $\mathbf{r}_0 = (x_0, y_0, z_0)$  is the ray's initial position,  $\mathbf{v}_0 = (v_x, v_y, v_z)$  the initial velocity and

$$k = \frac{c^2 A}{n_c} \quad (18.224)$$

$$k' = \frac{c^2 B}{n_c}. \quad (18.225)$$

Let us now define a focal point  $\mathbf{F}$ , where all rays launched will meet. The obvious choice is a focal point along the tube's center line  $(x_w, z_w)$ . Without specifying the y-coordinate of this point yet, we ask the question: at what times  $t$  and  $t'$  will  $x(t) = x_w$  and  $z(t') = z_w$  for the first time after launch? From the ray equation solutions we get

$$t = \frac{\arctan[-\sqrt{k}(x_0 - x_w)/v_x]}{\sqrt{k}} \quad (18.226)$$

$$t' = \frac{\arctan[-\sqrt{k'}(z_0 - z_w)/v_z]}{\sqrt{k'}}, \quad (18.227)$$

where the angular region of interest for the tangent is  $0 \leq \theta \leq \pi$ . It is interesting that there is both a lower and an upper limit to these times. While the lower limit of  $t \rightarrow 0$  makes sense (if we shoot the ray with infinite velocity towards the tube's center) the upper limit of  $t \rightarrow \pi/\sqrt{k}$  is less intuitive: there is no escape from the tube no matter what initial outward velocity we give the ray due to increasing pulling back force towards the center as the ray moves away from it. If the velocity component is zero we have  $t = \pi/(2\sqrt{k})$ .

In general, both times  $t$  and  $t'$  will be different. What this means is that the ray will not hit the focal point. For that to occur, both times must be equal, hence we must have

$$\frac{\arctan[-\sqrt{k}(x_0 - x_w)/v_x]}{\sqrt{k}} = \frac{\arctan[-\sqrt{k'}(z_0 - z_w)/v_z]}{\sqrt{k'}}. \quad (18.228)$$

Given  $A$  and  $B$  values (and thus  $k$  and  $k'$  values via equations 18.224 and 18.225) defining the tube's electron number density layout, this last relation imposes a condition among the variables  $x_0, z_0, v_x, v_z$ . Only if this condition is fulfilled will a focal point at  $(x_w, y_{\mathbf{F}}, z_w)$  be reached. The y-coordinate  $y_{\mathbf{F}}$  of the focal point is

then obtained from the y-solution 18.222 of the ray equation by inserting the value of the time  $t$  obtained through either side of 18.228. For the unit test we must identify  $y_{\mathbf{F}}$  to be sitting on the domain boundary. Hence when launching a series of ray's for the unit test, all  $y_{\mathbf{F}}$  must be the same. A first scheme I thus emerges for setting up a very general unit test:

#### SCHEME I

- Fix the center  $(x_w, z_w)$  of the ellipsoidal tube cross section, set up the x- and z-dimensions of the domain, and assign a value to the electron number density  $n_w$  at the center.
- Set up the ellipsoidal tube parameters  $A$  and  $B$ , as shown in 18.210 and 18.211, and choose a constant  $v_y$  velocity component.
- For one trial ray, given the initial position and velocity x-components  $x_0$  and  $v_x$ , determine the time  $t$  to reach the focal point  $\mathbf{F}$  using equation 18.226.
- Using  $t$  and  $v_y$ , calculate  $y_{\mathbf{F}}$  from 18.222, and set up the y-dimensions of the domain.
- Set up the initial x-component positions  $x_0$  for the whole set of rays that are going to be launched.
- Using  $t$ , determine the corresponding x-component velocities  $v_x$  for all rays from equation 18.226.
- For each ray, choose a  $z_0$  location and determine the needed velocity component  $v_z$  using the relation 18.228.
- Launch all rays and collect all the x- and z-components of the domain exit points  $(x_{exit}, z_{exit})$ .
- For each ray, check how close  $(x_{exit}, z_{exit})$  is to the expected  $(x_w, z_w)$ .

A second simplified scheme II (and the one that is currently implemented in FLASH) has all rays launched vertically onto the xz-face of the domain. This means that  $v_x = v_z = 0$ . By looking at the time condition 18.228, we see that the only possibility for a focal point in this case is if  $k = k'$ , and the time associated with it is equal to  $t = \pi/(2\sqrt{k})$ . This means that we must have  $A = B$  (a circular tube). Scheme II is set up as follows:

#### SCHEME II

- Fix the center  $(x_w, z_w)$  of the circular tube cross section, set up the x- and z-dimensions of the domain, and assign a value to the electron number density  $n_w$  at the center.
- Set up the circular tube parameter  $A$ , as shown in 18.210, and choose a constant  $v_y$  velocity component.
- Calculate the time  $t = (\pi/2c)\sqrt{n_c/A}$  to reach the focal point  $\mathbf{F}$ .
- Using  $t$  and  $v_y$ , calculate  $y_{\mathbf{F}}$  from 18.222, and set up the y-dimensions of the domain.
- Set up the initial positions  $(x_0, z_0)$  for the whole set of rays that are going to be launched.
- Launch all rays and collect all the x- and z-components of the domain exit points  $(x_{exit}, z_{exit})$ .
- For each ray, check how close  $(x_{exit}, z_{exit})$  is to the expected  $(x_w, z_w)$ .

Note, that for scheme II there is no restriction on the initial ray positions  $(x_0, z_0)$  like in scheme I. Rays launched vertically from anywhere onto the xz-plane with the same  $v_y$  velocity will reach the same focal point  $\mathbf{F}$ . Their paths might be of different lengths, but the time it takes for them to reach  $\mathbf{F}$  is always the same.



### 18.4.12.2 Analytic Power Deposition Solution for the Ellipsoidal Quadratic Potential Tube

According to equation 18.47, the power remaining after the ray exits the ellipsoidal tube is

$$P_{exit} = P_0 \exp \left\{ - \int_0^{t_{cross}} \nu_{ib}[\mathbf{r}(t)] dt \right\}, \quad (18.229)$$

where  $P_0$  and  $P_{exit}$  denote the ray's power at the tube's entry and exit point and  $t_{cross}$  is the time it takes for the ray to reach the focal point  $\mathbf{F}$ . From 18.46, the inverse-Bremsstrahlung rate is

$$\nu_{ib}[\mathbf{r}(t)] = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{e^4}{n_c k^{3/2}} \frac{Z[\mathbf{r}(t)] n_e[\mathbf{r}(t)]^2 \ln \Lambda[\mathbf{r}(t)]}{T_e[\mathbf{r}(t)]^{3/2}}. \quad (18.230)$$

Note the dependence of  $Z$  on position, since we are dealing with the entire domain and not with individual cells. In order to obtain an analytical integrand that is easy to integrate, the first assumption that we make is that both  $Z$  and  $\Lambda$  are independent of position. For convenience we chose:

$$Z[\mathbf{r}(t)] = 1 \quad (18.231)$$

$$\ln \Lambda[\mathbf{r}(t)] = 1. \quad (18.232)$$

To change the quadratic dependence on the electron number density to a linear dependency, we set the electron temperature as

$$T_e[\mathbf{r}(t)] = T_w \left[ \frac{n_e[\mathbf{r}(t)]}{n_w} \right]^{2/3} \quad (18.233)$$

with a specific electron temperature  $T_w$  at the elliptical origin. Using these choices we obtain

$$\begin{aligned} \nu_{ib}[\mathbf{r}(t)] &= \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_w e^4}{n_c k^{3/2} T_w^{3/2}} n_e[\mathbf{r}(t)] \\ &= \nu_{ib,w} \frac{n_e[\mathbf{r}(t)]}{n_w}, \end{aligned} \quad (18.234)$$

where

$$\nu_{ib,w} = \frac{4}{3} \left( \frac{2\pi}{m_e} \right)^{1/2} \frac{n_w^2 e^4}{n_c k^{3/2} T_w^{3/2}} \quad (18.235)$$

is the inverse-Bremsstrahlung rate at the origin. The integral in the exponential becomes

$$\begin{aligned} \int_0^{t_{cross}} \nu_{ib}[\mathbf{r}(t)] dt &= \frac{\nu_{ib,w}}{n_w} \int_0^{t_{cross}} n_e[\mathbf{r}(t)] dt \\ &= \frac{\nu_{ib,w}}{n_w} \int_0^{t_{cross}} n_w + A(x[t] - x_w)^2 + B(z[t] - z_w)^2 dt. \end{aligned} \quad (18.236)$$

Inserting the analytical path solutions 18.221 and 18.223, we can perform the integration. Let us concentrate on the x-component part only. We are lead to an integral of the form

$$\begin{aligned} \int_0^{t_{cross}} (x[t] - x_w)^2 dt &= \int_0^{t_{cross}} \left[ (x_0 - x_w) \cos(\sqrt{k}t) + \frac{v_x}{\sqrt{k}} \sin(\sqrt{k}t) \right]^2 dt \\ &= \frac{v_x(x_0 - x_w)}{2k} + \frac{v_x^2 + k(x_0 - x_w)^2}{2k} t_{cross}, \end{aligned} \quad (18.237)$$

where  $t_{cross}$  is given by the expression in 18.226. For this integration, known sine and cosine integration rules have been used as well as the fact that  $\sin(\arctan a) = a/\sqrt{1+a^2}$  and  $\cos(\arctan a) = 1/\sqrt{1+a^2}$ . The

z-component integration is similar. Collecting everything together and using the expressions for  $k, k', A$  and  $B$  from 18.224, 18.225, 18.210 and 18.211 we obtain

$$\begin{aligned} \int_0^{t_{cross}} \nu_{ib}[\mathbf{r}(t)] dt &= \nu_{ib,w} \left[ t_{cross} + \frac{n_c}{2c^2 n_w} \{ v_x(x_0 - x_w) + v_z(z_0 - z_w) + (v_x^2 + v_z^2)t_{cross} \} \right. \\ &\quad \left. + \frac{n_c - n_w}{2n_w} \left\{ \frac{(x_0 - x_w)^2}{(x_c - x_w)^2} + \frac{(z_0 - z_w)^2}{(z_c - z_w)^2} \right\} t_{cross} \right]. \end{aligned} \quad (18.238)$$

This is the general expression of the exponent integral for evaluating the power deposition of the rays for scheme I. For the circular tube of scheme II we have  $v_x = v_z = 0$ ,  $t_{cross} = (\pi/2c)\sqrt{n_c/A}$ ,  $A = B$ , and the exponent integral simplifies to

$$\int_0^{t_{cross}} \nu_{ib}[\mathbf{r}(t)] dt = \nu_{ib,w} t_{cross} \left[ 1 + \frac{A}{2n_w} \{ (x_0 - x_w)^2 + (z_0 - z_w)^2 \} \right]. \quad (18.239)$$

If all rays start with the same power and imposing equal power deposition for all rays, then the term inside the curly brackets must be equal to a constant value

$$(x_0 - x_w)^2 + (z_0 - z_w)^2 = R^2. \quad (18.240)$$

This condition implies that the rays have to be launched on a circle with radius  $R$  around the tube origin  $(x_w, z_w)$ . This ensures equal path lengths for all rays and hence equal power deposition.

#### 18.4.12.3 Unit Tests Parameters and Results

The current Energy Deposition unit test coded into FLASH is based on the analytic solutions for scheme II. The circular quadratic potential tube is defined through the following parameters and constants:

Parameters	Constants in FLASH
$x_c, z_c = 10 \text{ cm}$	$m_e = 9.10938215 \times 10^{-28} \text{ g}$
$x_w, z_w = 5 \text{ cm}$	$e = 4.80320427 \times 10^{-10} \text{ esu}$
$R = 3 \text{ cm}$	$k = 1.3806504 \times 10^{-16} \text{ erg/K}$
$\lambda = 1 \times 10^{-4} \text{ cm}$	$c = 2.99792458 \times 10^{10} \text{ cm/sec}$
$v_y = c$	
$T_w = 10 \text{ keV}$	
$P_0 = 1 \text{ erg/sec}$	

With these values we obtain the following additional variable values (in parenthesis are the equations used), completing the picture:

$$\begin{aligned} n_c &= 1.1148542 \dots \times 10^{21} \text{ cm}^{-3} & (18.35) \\ n_w &= n_c/2 \\ t_{cross} &= 5\pi/c\sqrt{2} \text{ sec} & (18.226 \text{ with } v_x = 0) \\ y_{\mathbf{F}} &= 5\pi/\sqrt{2} \text{ cm} & (18.222) \\ \nu_{ib,w} &= 8.1002987 \dots \times 10^8 \text{ sec}^{-1} & (18.235) \\ P_{exit} &= 0.7017811 \dots \text{ erg/sec} & (18.239 \text{ and } 18.229) \end{aligned}$$

Evaluation of each cell's average electron number density is done by integrating equation 18.209 over the entire cell xz-area

$$\begin{aligned} \langle n_e \rangle &= \frac{\int_{z_1}^{z_2} \int_{x_1}^{x_2} n_e(x, y) dx dz}{(x_2 - x_1)(z_2 - z_1)} \\ &= n_w + A(\bar{x} - x_w)^2 + A(\bar{z} - z_w)^2 + \frac{A}{3}(\Delta x^2 + \Delta z^2), \end{aligned} \quad (18.241)$$

where  $x_1, z_1$  are the lower and  $x_2, z_2$  the upper corresponding cell edge coordinates and

$$\bar{x} = \frac{x_1 + x_2}{2} \quad (18.242)$$

$$\Delta x = \frac{x_1 - x_2}{2} \quad (18.243)$$

with corresponding equations for the z-part. The average cell's electron temperature is evaluated directly from the average electron number density using 18.233

$$\langle T_e \rangle = T_w \left[ \frac{\langle n_e \rangle}{n_w} \right]^{2/3}. \quad (18.244)$$

The gradients are obtained from the cell average values using the symmetrized derivative formula (shown here for the  $i$ -th cell x-component of the electron number density gradient)

$$\frac{\partial}{\partial x} n_e(i) = \frac{\langle n_e \rangle_{i+1} - \langle n_e \rangle_{i-1}}{d_{i-1,i+1}}, \quad (18.245)$$

where  $d_{i-1,i+1}$  is twice the distance between neighboring cells on the x-axis.

#### TEST I

For test I, a series of 8 rays (from 8 beams with 1 ray each) with radial distances of  $3cm$  from  $(x_w, z_w) = (5cm, 5cm)$  are launched with the following initial  $(x_0, z_0)$  coordinates (all in cm):

Ray	$x_0$	$z_0$
1	8	5
2	5	8
3	2	5
4	5	2
5	$5 + 3/\sqrt{2}$	$5 + 3/\sqrt{2}$
6	$5 + 3/\sqrt{2}$	$5 - 3/\sqrt{2}$
7	$5 - 3/\sqrt{2}$	$5 + 3/\sqrt{2}$
8	$5 - 3/\sqrt{2}$	$5 - 3/\sqrt{2}$

Results are shown only for the ray 1 and ray 5, the other ones are related to these two by symmetry. Also  $z_{exit}$  is not given, because for ray 1 we have  $z_{exit} = 5cm$  and for ray 5 we have  $z_{exit} = x_{exit}$ . The percentage errors were calculated as  $|x_{exit} - x_w|/x_w$  and likewise for the power deposition.

Refinement Level	$x_{exit}$	% error	$P_{exit}$	% error
Ray 1				
1	5.04894	0.98	0.69487	0.98
2	5.16343	3.27	0.69864	0.45
3	5.02737	0.55	0.70107	0.10
4	4.98971	0.21	0.70185	0.01
5	5.00004	0.00	0.70176	0.00
6	5.00234	0.05	0.70174	0.00
Ray 5				
1	5.35171	7.03	0.69283	1.23
2	4.77272	4.55	0.70200	0.03
3	4.96221	0.76	0.70115	0.09
4	4.94433	1.11	0.70204	0.04
5	4.95092	0.98	0.70235	0.08
6	4.97987	0.40	0.70198	0.02
Analytical				
$\infty$	5.00000		0.70178	

## TEST II

The second test is also based on scheme II and aims at launching a large number of rays using only one beam centered along the tube's origin with a specific cross section power function, such that all rays reaching the focal point have an equal exit power. Since the power deposited depends on the ray path length, the cross section power function must be such that each ray gets the appropriate initial power to account for the differences in path lengths. For a specific ray we have for the exit power

$$P_{exit,ray} = \bar{P} \frac{w_{ray}}{\sum w_{ray}} \exp \left\{ -\nu_{ib,w} t_{cross} \left[ 1 + \frac{AR_{ray}}{2n_w} \right] \right\}, \quad (18.246)$$

where equations 18.239, 18.229 and 18.108 have been used and  $R_{ray} = (x_0 - x_w)^2 + (z_0 - z_w)^2$  is the ray's radial distance from the tube origin. If we set the ray weighting function equal to the inverse gaussian function

$$w_{ray} = e^{+\frac{A\nu_{ib,w} t_{cross}}{2n_w} R_{ray}^2}, \quad (18.247)$$

we obtain

$$P_{exit,ray} = \frac{\bar{P}}{Z} e^{-\nu_{ib,w} t_{cross}}, \quad (18.248)$$

where we have used  $Z$  for the power partition function  $\sum w_{ray}$ . For a particular number of rays per beam,  $Z$  is constant and therefore the exit power of all rays are equal. To get independent of the number of rays, the unit test records the partition scaled exit power  $P_{exit,ray}Z$  for each ray as well as their focal  $(x_{exit}, z_{exit})$  coordinates. A maximum square radial deviation

$$\Delta R_{focal}^2 = \max \{ (x_{exit} - x_w)^2 + (z_{exit} - z_w)^2 \} \quad (18.249)$$

from the focal point is then defined and recorded. The maximum absolute deviation in the partition scaled power is registered as

$$\Delta P_{focal} = \max |P_{exit,ray}Z - \bar{P}e^{-\nu_{ib,w} t_{cross}}|. \quad (18.250)$$

Both quantities are then checked against tolerance values.

### 18.4.13 Unit Tests for 3D Cylindrical Domain Geometries

In these unit tests a ring of laser rays is shot on to a 3D cylindrical can and their domain exit properties are recorded and analyzed.

#### 18.4.13.1 Laser Ring on Mantle of 3D Cylindrical Can with No Domain Acceleration

This test reveals how well a straight line movement by the rays is maintained in 3D cylindrical coordinates. As rays enter the 3D cylindrical domain, their radial and tangential angular velocity change continuously along their straight line path through the can. On exit the laser ring should have the same radius as on entry and all rays should have the same initial speed, with deviations depending on the domain refinement level. Figure 18.19 shows schematically the unit test.

#### 18.4.13.2 Unit Test Parameters and Results

The 3D cylindrical can is specified by the following domain min/max coordinates:  $R_{max} = 1.0$  cm,  $z_{min} = -1.0$  cm and  $z_{max} = 1.0$  cm. The laser ring consists of  $10^6$  rays, has a radius of 0.5 cm and is shot along the 3D cartesian  $y$  direction on the mantle center at  $z = 0$  cm. Each ray carries a power of 1 erg/s. A uniform electron number density (half the critical density) is set throughout the can to ensure no radial and angular

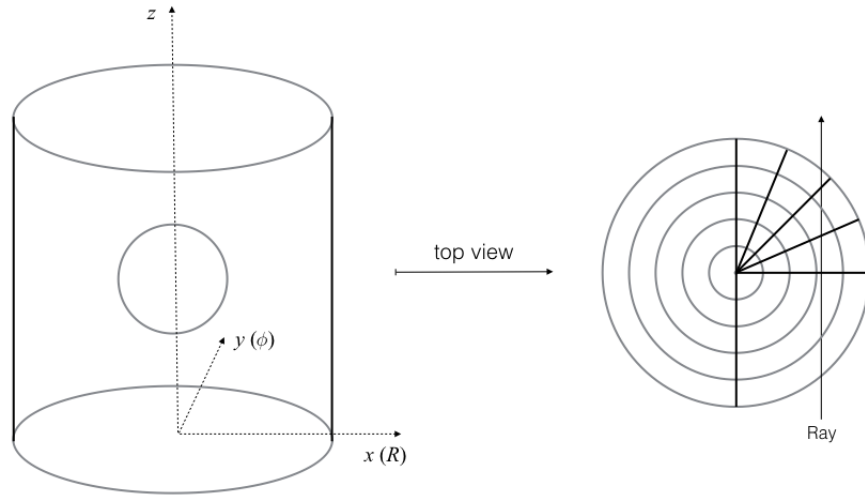


Figure 18.19: Schematic drawing of the incident laser ring on the mantle of a 3D cylindrical can with no domain ray acceleration. Linear path of a ray through the 3D cylindrical domain blocks/cells is also shown.

acceleration. Speed adjustment of the rays as they enter the domain is suppressed, hence all rays will travel with the speed of light  $c$  through the can. A uniform inverse Bremsstrahlungsrate of

$$\nu_{ib} = -c \ln(0.5)/2 \quad (18.251)$$

is used, to accomplish a power reduction by 0.5 for those rays that travel with the speed of light the full diameter of 2 cm of the can. In the top view of Figure 18.19, we denote by  $y$  half the distance traveled by the ray inside the can. Hence the ray travels  $2y$  cm through the can in  $2y/c$  seconds and for this ray an exit power reduction of

$$P_{exit}/P_0 = \exp(-\nu_{ib}2y/c) = 2^{-y} \quad (18.252)$$

is expected. The unit test collects three relative quantities (exit value / expected value) for each ray: 1) the ring radius, 2) the ray speed and 3) the ray power. Min/max of these relative quantities are then formed over all rays and deviations from 1 are checked against a predefined benchmark for passing the test.

An additional test is being done to see if the laser ring retained its shape while entering and traveling through the can. On creation in the beam, all rays of the laser ring are placed uniformly on the ring with equal separation among each other. On exit from the can the rays should still be uniformly spaced in their angular position. Since checking for nearest neighbor rays for each ray is in our case a  $N^2 = 10^6 \times 10^6 = 10^{12}$  process, we resort to a simpler method. We define an angular tic  $\phi_{tic}$  value as the minimum angular separation of each ray in the beam:

$$\phi_{tic} = 2\pi/10^6. \quad (18.253)$$

On exit from the can, the angular position variable  $\phi$  of each ray is divided by  $\phi_{tic}$  and the nearest integer is formed, giving the label  $T$  of a tic in which the ray would be found:

$$T = \text{nint}(\phi/\phi_{tic}). \quad (18.254)$$

If the laser ring retained its shape, then there should be only 1 ray for each tic, i.e. the list of labels  $\{T\}$  should consist of consecutive natural numbers, which can easily be checked by defining a zero tic array and adding 1 to each tic whenever a ray is found to belong to it. A final tic array containing only 1's indicates success in retaining the original laser ring shape. Since straight line movement in 3D cylindrical coordinates is a considerable stress test for the code, we allow a maximum tic occupancy of 2 (and hence a tic occupancy of 0 for other tics) for this test to succeed.

### 18.4.13.3 Laser Ring on Lid of 3D Cylindrical Can with Uniform Radial Acceleration

The purpose of this test is to check proper ray crossing at the 3D cylindrical axis. The laser ring contracts towards the axis in a singular point at  $R = 0$  and expands afterwards regaining its original shape. On exit the laser ring should have the same radius as on entry and all rays should have the same initial speed, which increases towards the singular point when the ring contracts and decreases afterwards when the ring expands. Inside the can the laser ring traces an hourglass shape with a single point at its center.

### 18.4.13.4 Unit Test Parameters and Results

The 3D cylindrical can is specified by the following domain min/max coordinates:  $R_{max} = 1.0$  cm,  $z_{min} = -1.0$  cm and  $z_{max} = 1.0$  cm. The laser ring consists of  $10^6$  rays, has a radius of 0.5 cm and is shot along the 3D cartesian  $z$  direction on the lid center at  $x, y = 0$  cm. Each ray carries a power of 1 erg/s. A linear electron number density gradient

$$n_e(R) = n_c R / R_{max} \quad (18.255)$$

is set throughout the can to ensure a uniform radial acceleration of (from Eq.(18.161) and using  $R_{max} = 1.0$ )

$$n_R(\mathbf{r}) = -c^2/2 \quad (18.256)$$

towards the 3D cylindrical axis. As the initial radial velocity  $v_R$  is zero, the time it takes for each ray to exit the can on the opposite lid is equal to twice the time it takes for it to reach the axis at  $R = 0$ :

$$t_{exit} = 2t_{axis} = 2\sqrt{2}/c. \quad (18.257)$$

Using a uniform inverse Bremsstrahlungsrate of

$$\nu_{ib} = -c \ln(0.5)/2, \quad (18.258)$$

the power reduction rate for each ray at exit is

$$P_{exit}/P_0 = \exp(-\nu_{ib} 2\sqrt{2}/c) = 2^{-\sqrt{2}}. \quad (18.259)$$

In order for the rays not to exceed the speed of light inside the can, initial ray speed adjustment is set true, with each ray entering the can at  $R = 0.5$  with an axial velocity of:

$$v_z = c * \sqrt{1 - n_R(0.5)/n_c} = c/\sqrt{2} \quad (18.260)$$

At the axis, the rays will have gained a radial velocity of  $v_R = -c/\sqrt{2}$  due to the radial acceleration. On exit the radial velocity is again  $v_R = 0$ , while the axial velocity remains unchanged. The speed of each ray at the axis and at the exit point will therefore be:

$$|v|_{axis} = c \quad (18.261)$$

$$|v|_{exit} = c/\sqrt{2}. \quad (18.262)$$

The unit test collects three relative quantities at the exit point (exit value / expected value) for each ray: 1) the ring radius, 2) the ray speed and 3) the ray power. Min/max of these relative quantities are then formed over all rays and deviations from 1 are checked against a predefined benchmark for passing the test. Since pure radial movement of the rays is considerably easier in 3D cylindrical coordinates when compared to straight line movements, the laser ring shape test is only successful, if all tics are strictly occupied only once.

## 18.5 Heatexchange

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

$$\frac{\partial \rho E_{Ele}}{\partial t} = \kappa_{i,e}(T_{Ion} - T_{Ele}) + \kappa_{e,r}(T_{Rad} - T_{Ele}) \quad (18.263)$$

$$\frac{\partial E_{Rad}}{\partial t} = -\kappa_{e,r}(T_{Rad} - T_{Ele}) \quad (18.264)$$

$$\frac{\partial \rho E_{Ion}}{\partial t} = -\kappa_{i,e}(T_{Ion} - T_{Ele}) \quad (18.265)$$

This unit is required because FLASH operator splits the electron-ion equilibration term from the rest of the calculations. All realistic 3T HEDP simulations should include the `Heatexchange` unit since it is responsible for equilibrating the ion/electron temperatures over time. The radiation terms in (18.263) are *only* included when the legacy gray radiation diffusion unit is used. The more sophisticated multigroup radiation diffusion unit, described in [Chapter 25](#), includes the emission and absorption internally.

There are several different implementations residing under Heatexchange unit,

- **Immediate:** Instantaneous complete equilibration.
- **Constant:** `hx_couplingConst12`, `hx_couplingConst13` and `hx_couplingConst23` can be used for setting the constant coupling terms.
- **ConstCoulomb:** Uses a constant coulomb logarithm to compute electron-ion equilibration. `hx_coulombLog` can be used to set this value.
- **Spitzer:** Uses a Spitzer electron-ion equilibration timescale.
- **LeeMore:** Uses a Spitzer-like electron-ion equilibration timescale but with the Coulomb logarithm from Lee & More 1984.

The **Constant** and **ConstCoulomb** implementations are really designed to be used for testing purposes. The **Spitzer** and **LeeMore** implementation should be used for simulations of actual HEDP experiments since it is more accurate and physically realistic. This is described in detail in [Section 18.5.1](#) and [Section 18.5.2](#).

`hx_relTol` runtime parameter affects the time step computed by `Heatexchange.computeDt`. Basically, if the max (abs) temperature adjustment that would be introduced in any nonzero component in any cell is less than `hx_relTol`, then the time step limit is relaxed. Set to a negative value to inherit the value of runtime parameter `eos_tolerance`.

**Usage:** To use any implementation of Heatexchange unit, a simulation should include the Heatexchange using an option like `-with-unit=physics/sourceTerms/Heatexchange/HeatexchangeMain/ConstCoulomb` on the setup line, or add

REQUIRES `physics/sourceTerms/Heatexchange/HeatexchangeMain/ConstCoulomb` in the Config file. The same process can be repeated for other implementations. Constant implementation is set as the default.

### 18.5.1 Spitzer Heat Exchange

The **Spitzer** implementation described here and **LeeMore** implementation described later in [Section 18.5.2](#) should be used to model electron-ion equilibration in realistic HEDP simulations. See [Section 35.7.5](#) for an example of how to use the **Spitzer Heatexchange** implementation in an HEDP simulation. Note, that this implementation only models electron-ion equilibration, and does not include any radiation related physics. In that sense it is designed to be used in conjunction with the multigroup radiation unit described in [diffusion Chapter 25](#), since this unit includes the emission and absorption terms internally. Thus the equations solved by the **Spitzer** implementation are:

$$\frac{de_{ion}}{dt} = \frac{c_{v,ele}}{\tau_{ei}}(T_{ele} - T_{ion}) \quad (18.266a)$$

$$\frac{de_{ele}}{dt} = \frac{c_{v,ele}}{\tau_{ei}}(T_{ion} - T_{ele}) \quad (18.266b)$$

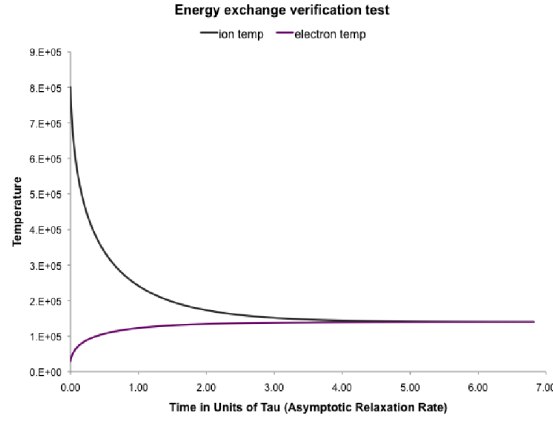


Figure 18.20: Verification test for heat exchange code units, showing electrons and ions at different temperatures coming into mutual equilibrium, as expected from the form of the electron-ion coupling. The time axis unit is equal to the asymptotic value of the exponential decay time.

where:

- $T_{\text{ele}}$  is the electron temperature
- $T_{\text{ion}}$  is the ion temperature
- $e_{\text{ele}}$  is the electron specific internal energy
- $e_{\text{ion}}$  is the ion specific internal energy
- $c_{v,\text{ele}}$  is the electron specific internal energy
- $\tau_{ei}$  is the electron-ion equilibration time

We then replace  $de$  with  $c_v dT$  for the ions and electrons to obtain:

$$\frac{dT_{\text{ion}}}{dt} = \frac{m}{\tau_{ei}} (T_{\text{ele}} - T_{\text{ion}}) \quad (18.267a)$$

$$\frac{dT_{\text{ele}}}{dt} = \frac{1}{\tau_{ei}} (T_{\text{ion}} - T_{\text{ele}}) \quad (18.267b)$$

where  $m = c_{v,\text{ele}}/c_{v,\text{ion}}$  is the ratio of specific heats. (18.267) can be solved by assuming that the specific heats and electron-ion equilibration times are constant over a time step,  $\Delta t$ . The result is that the ion/electron temperatures equilibrate exponentially over a time step. This is shown in (18.268).

$$T_{\text{ion}}^{n+1} = \left( \frac{T_{\text{ion}}^n + mT_{\text{ele}}^n}{1+m} \right) - m \left( \frac{T_{\text{ele}}^n - T_{\text{ion}}^n}{1+m} \right) \exp \left[ -(1+m) \frac{\Delta t}{\tau_{ei}} \right] \quad (18.268a)$$

$$T_{\text{ele}}^{n+1} = \left( \frac{T_{\text{ion}}^n + mT_{\text{ele}}^n}{1+m} \right) + \left( \frac{T_{\text{ele}}^n - T_{\text{ion}}^n}{1+m} \right) \exp \left[ -(1+m) \frac{\Delta t}{\tau_{ei}} \right] \quad (18.268b)$$

The **Spitzer** implementation does not directly update the temperatures, however, since this would lead to energy conservation errors. Rather, the specific internal energies are directly updated according to:

$$e_{\text{ele}}^{n+1} = e_{\text{ele}}^n + c_{v,\text{ele}}^n (T_{\text{ele}}^{n+1} - T_{\text{ele}}^n) \quad (18.269a)$$

$$e_{\text{ion}}^{n+1} = e_{\text{ion}}^n + c_{v,\text{ion}}^n (T_{\text{ion}}^{n+1} - T_{\text{ion}}^n) \quad (18.269b)$$

The final temperatures are then computed using a call to the equation of state.



The subroutine `PlasmaState_ieEquilTime` is responsible for computing the electron-ion equilibration time,  $\tau_{ei}$ . Currently, the formula used is consistent with that given in (Atzeni, 2004) and the NRL Plasma Formulary (Huba, 2011). The value of  $\tau_{ei}$  is shown in (18.270).

$$\tau_{ei} = \frac{3k_B^{3/2}}{8\sqrt{2\pi}e^4} \frac{(m_{\text{ion}}T_{\text{ele}} + m_{\text{ele}}T_{\text{ion}})^{3/2}}{(m_{\text{ele}}m_{\text{ion}})^{1/2} \bar{z}^2 n_{\text{ion}} \ln \Lambda_{ei}}. \quad (18.270)$$

where:

- $k_B$  is the Boltzmann constant
- $e$  is the electron charge
- $m_{\text{ion}}$  is the average mass of an ion
- $m_{\text{ele}}$  is the mass of an electron
- $\bar{z}$  is the average ionization as computed by the EOS
- $n_{\text{ion}}$  is the ion number density in the plasma
- $\ln \Lambda_{ei}$  is the Coulomb Logarithm associated with electron-ion collision, further description below.

The Coulomb logarithm is the logarithmic ratio of the maximum impact parameter,  $b_{\text{max}}$ , to minimum impact parameter,  $b_{\text{min}}$ , for electron-ion collisions, i.e.,

$$\ln \Lambda_{ei} = \ln \left[ 1 + \left( \frac{b_{\text{max}}}{b_{\text{min}}} \right) \right] \quad (18.271)$$

In general, the scales of  $b_{\text{min}}$  and  $b_{\text{max}}$  should be well separated and therefore the Coulomb logarithm should always be a dimensionless, positive number with a value of a few or greater. The expressions for  $b_{\text{max}}$  and  $b_{\text{min}}$  used in the **Spitzer** heat exchange are given by (Brysk, 1974),

$$b_{\text{max}} = \sqrt{\frac{k_B T_{\text{ele}}}{4\pi e^2 n_{\text{ele}}}} \quad (18.272)$$

and

$$b_{\text{min}} = \max \left( \frac{\bar{z}e^2}{3k_B T_{\text{ele}}}, \frac{\hbar}{2\sqrt{3k_B T_{\text{ele}} m_{\text{ele}}}} \right). \quad (18.273)$$

where  $n_{\text{ele}}$  is the electron number density and  $\hbar$  is Planck's constant.

Note that in HED plasmas, there are situations where  $b_{\text{max}} \sim b_{\text{min}}$  resulting in a very small or negative value. Therefore, the Coulomb logarithm is forced to be greater than or equal to 1.0 by default (as in Brysk, 1974). This floor value can be changed by setting the runtime parameter `logLambdaFloor`.

Finally, note that the runtime parameter `hx_ieTimeCoef` multiplies  $\tau_{ei}$  and can be used to scale  $\tau_{ei}$ .

### 18.5.2 LeeMore Heat Exchange

The **LeeMore** heat exchange uses (18.270), but with a different expression for the Coulomb logarithm,  $\ln \Lambda_{ei}$ . The expression for  $\ln \Lambda_{ei}$  comes from Sec. III.B. of Lee & More 1984,

$$\ln \Lambda_{ei} = \frac{1}{2} \ln \left[ 1 + \left( \frac{b_{\text{max}}}{b_{\text{min}}} \right)^2 \right] \quad (18.274)$$

where  $b_{\text{max}}$  is equal to the Debye-Huckel screening length,  $\lambda_{\text{DH}}$ , in an expression that includes the ion temperature and mean ionization state,

$$\frac{1}{\lambda_{\text{DH}}^2} = \frac{4\pi n_{\text{ele}} e^2}{kT_{\text{ele}}} + \frac{4\pi n_{\text{ion}} (e\bar{z})^2}{kT_{\text{ion}}} \quad (18.275)$$

and  $b_{\min}$  is the same as used in the **Spitzer** heat exchange model, i.e. (18.273).

Like the **Spitzer** heat exchange model, the Coulomb logarithm in the **LeeMore** heat exchange is forced to be greater or equal to `logLambdaFloor`. Lee & More 1984 cite numerical results that conclude that the floor on the Coulomb logarithm should be 2.0 (instead of 1.0 as in the **Spitzer** heat exchange model).

It should be reiterated that the **LeeMore** heat exchange model just described only differs from the **Spitzer** heat exchange in the calculation of the Coulomb logarithm. The expression in Lee & More 1984 for the electron relaxation time (their Eq. 24) also contains Fermi-Dirac factors that we neglect here for simplicity. These factors may be incorporated into the **LeeMore** heat exchange model at a later date. Regardless, the **LeeMore** heat exchange module should be more accurate than the **Spitzer** heat exchange because of the improvement to the Coulomb logarithm.

## 18.6 Flame

The **Flame** unit implements an artificial reaction front model based on a simplified single-variable reaction-diffusion model. This is intended to be used to model the behavior of under-resolved reaction fronts propagated spatially by thermal diffusion, commonly called flames or deflagration fronts. As the reaction-diffusion front is also carried with the fluid, this model is sometimes termed an advection-diffusion-reaction (ADR) model. The principal usage to date of this model has been during the deflagration phase of models of Type Ia Supernovae. See Townsley et al. (2007), from which some text in this section has been adapted, and works that reference it for examples.

### 18.6.1 Reaction-Diffusion Forms

Generally, an ADR scheme characterizes the location of a flame front using a reaction progress variable,  $\phi$ , which increases monotonically across the front from 0 (fuel) to 1 (ash). Evolution of this progress variable is accomplished via an advection-diffusion-reaction equation of the form

$$\frac{\partial \phi}{\partial t} + \vec{v} \cdot \nabla \phi = \kappa \nabla^2 \phi + \frac{1}{\tau} R(\phi) , \quad (18.276)$$

where  $\vec{v}$  is the local fluid velocity, and the reaction term,  $R(\phi)$ , timescale,  $\tau$ , and the diffusion constant,  $\kappa$ , are chosen so that the front propagates at the desired speed. Vladimirova et al. (2006) showed that the step-function reaction rate leads to a substantial amount of unwanted acoustic noise. They studied a suitable alternative, the Kolmogorov Petrovski Piskunov (KPP) reaction term which has an extensive history in the study of reaction-diffusion equations. In the KPP model the reaction term is given by

$$R(\phi) = \frac{1}{4} \phi (1 - \phi) . \quad (18.277)$$

The symmetric and low-order character of this function gives it very nice numerical properties, leading to amazingly little acoustic noise. Following Vladimirova et al. (2006), we adopt  $\kappa \equiv sb\Delta x/16$  and  $\tau \equiv b\Delta x/16s$ , where  $\Delta x$  is the grid spacing,  $s$  is the desired propagation speed, and  $b$  sets the desired front width scaled to represent approximately the number of zones.

The KPP reaction term, however, has two serious drawbacks. Formally, the flame speed is only single valued for initial conditions that are precisely zero (and stay that way) outside the burned region (Xin 2000), which cannot really be effected in a hydrodynamics simulation. This can lead to an unbounded increase of the propagation speed, which is precisely the property we wish to have under good control. Secondly, the progress variable  $\phi$  takes an infinite amount of time to actually reach 1 (complete consumption of fuel). While not a fatal flaw like the flame speed problem, this is a problem for our simulations in which we would like to have a localized flame front so that fully-burned ash can be treated as pure NSE material.

Both of these drawbacks can be ameliorated by a slight modification of the reaction term (S. Asida, private communication) to

$$R(\phi) = \frac{f}{4} (\phi - \epsilon_0)(1 - \phi + \epsilon_1) , \quad (18.278)$$

where  $0 < \epsilon_0, \epsilon_1 \ll 1$  and  $f$  is an additional factor that depends on  $\epsilon_0$  and  $\epsilon_1$  and the flame width so that the flame speed is preserved with the same constants as for KPP above. This “sharpened” KPP (sKPP) has

truncated tails in both directions (thus being sharpened), making the flame front fully localized, and is a bi-stable reaction rate and thus gives a unique flame speed (Xin 2000). The price paid is that since  $R(\phi) = 0$  for  $\phi \leq 0$  and  $\phi \geq 1$ , (18.278) is discontinuous, adding some noise to the solution. Since the suppression of the tails is stronger for higher  $\epsilon_0$  and  $\epsilon_1$ , we adjusted  $\epsilon_0$  and  $\epsilon_1$  so that for a particular flame width we could meet noise goals (Townsend et al. 2007). The parameter the default parameter values obtained are  $\epsilon_0 = \epsilon_1 = 10^{-3}$ ,  $f = 1.309$ , and  $b = 3.2$ . The noise properties of these choices are discussed in Townsend et al. (2007).

Diffusive flames are known to be subject to a curvature effect that affects the flame speed when the radius of curvature is similar to the flame thickness, a frequent circumstance with modestly-resolved flame front structure. In testing, the curvature effect of the step-function reaction rate proved surprisingly strong, likely due to the exponential “nose” that the flame front possesses (Vladimirova et al. 2006). Both KPP and sKPP show significantly better curvature properties.

## 18.6.2 Unit Structure

The unit is divided into several subunits to allow flexibility in how the propagation speed is determined and how the energy release of the reaction front takes place. The main subunit `FlameMain` just implements the actual the reaction and diffusion terms defined above. There is currently only one implementation, `RDSplit5point`, that uses operator-split steps for the reaction and diffusion, and a 5 point stencil for the Laplacian operator.

The order of operations occurs as follows, for each block of the mesh in sequence: The progress variable,  $\phi$ , stored in `FLAM_MSCALAR` is extracted, the Unit-internal function `fl_flameSpeed()` is called to fill a temporary array with the flame speed evaluated for each cell in the current block. The mesh scalar `FLAM_MSCALAR` is then updated based on this flame speed while saving temporarily the effective  $\phi$  for this time step. This information is then passed on to the internal function `fl_effects()` which performs any energy release or other desired side effects of the propagation of the RD front. Note both the temporary arrays storing the flame speed and the  $\phi$  for the current block are destroyed before processing the next block. However some implementations of `fl_flameSpeed()` may save this information in the main grid arrays for informational or other purposes.

### 18.6.2.1 Flame Speed

The flame speed is generally determined by various physical quantities on the grid. However, since the flame is generally an artificial reaction front, many aspects other than the composition may influence the speed. The simplest alternative implementation for determining the flame speed, which is the default one, is `Flame/FlameSpeed/Constant`. This simply sets a constant flame speed everywhere on the grid.

The flame speed can also include a turbulence-flame interaction (TFI). The implemented options are discussed in Jackson, Townsend, & Calder (2014, submitted). This is accomplished by including one of the implementations under, for example `Flame/FlameSpeed/Constant/TFI`. Since the TFI always modifies some base flame speed, it is generally included as a implementation of that type of flame speed, in this case a constant flame speed. In order to facilitate sharing of the TFI components, they are stored in their own implementation directories under `Flame/FlameSpeed/turbulent` and included as needed. All of the TFI options require the inclusion of the `Turb` unit which provides a measurement of the local turbulence strength.

### 18.6.2.2 Flame Effects

The flame effects subunit performs functions related to the actual energy deposition or changes in abundances caused by advance of the reaction front being modeled by the reaction-diffusion progress variable  $\phi$ . The simplest implementation, included under `Flame/FlameEffects/EIP`, implements fixed energy input and change in  $\bar{A}$  and  $\bar{Z}$  across the reaction front. This is constructed to utilize the fully ionized electron-ion plasma EOS called the “Helmholtz” EOS in Flash.

## 18.7 Turbulence Measurement

The **Turb** unit implements a method to measure the local turbulence strength at the grid scale. The implementation is based on the operator  $OP_2$  in Colin et al. (2000),

$$v' = OP_2(\vec{v}) = c_2 \Delta_x^3 \nabla^2 (\nabla \times \vec{v}) . \quad (18.279)$$

Here  $c_2$  is a calibration constant, set with the **turb\_c2** runtime variable. The default value, 0.9, was determined for the PPM hydrodynamics method by Jackson et al. (2014, submitted) based on simulations of stirred turbulence.  $\Delta_x$  is the grid spacing and  $v$  is the velocity field on the grid. Two options are implemented for sampling the velocity field, each of which uses a 5-point stencil for the Laplacian operator. If **turb\_stepSize** is set to 1, the stencil samples adjacent cells, if set to 2 every other cell is sampled, for an effective stencil size of 9 points. The latter option requires a guardcell fill between the curl and Laplacian operators. The result of equation 18.279 is stored in the **TURB** mesh variable.

## 18.8 Circuit

The **Circuit** unit implements various routines for calculating current supplied by an external circuit. The main purpose is to provide a boundary condition for current-driven systems such as a Z-pinch. There are two parameters that may be used by other units for such a boundary condition: **circ\_IloadOld** and **circ\_IloadNew**. These represent the current (in Amperes) at the current time step  $n$  before **Circuit** is called and the current at time step  $n + 1$  after **Circuit** has been called, respectively.

Some of the implementations will output useful information to a file named **circuit.dat**. The subsections that follow describe the different implementations that are currently included in FLASH. To activate a particular implementation, include the appropriate path in the **Config** file of your simulation directory.

### 18.8.1 Constant

In the **Constant** implementation is the simplest of the models and applies a constant current to the load. A constant current is useful when considering ideal cases or if actual data from a run is not available. A run time parameter that is defined by the **circ\_currConstant** variable sets the current. By setting this value in the **par** file this constant current will be applied to the load for the duration of the simulation. This implementation lies in the **physics/sourceTerms/Circuit/ CircuitMain/Constant** directory.

### 18.8.2 FileInput

The **FileInput** implementation reads in current data from a file, interpolates the data, and then applies the current to the load. The file input mode is useful when the exact properties of the current delivered to the load are known, but not much else about the system.

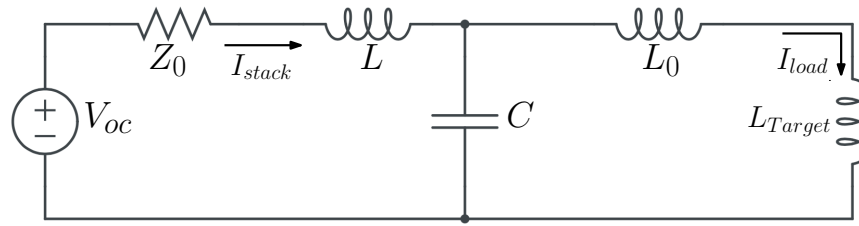
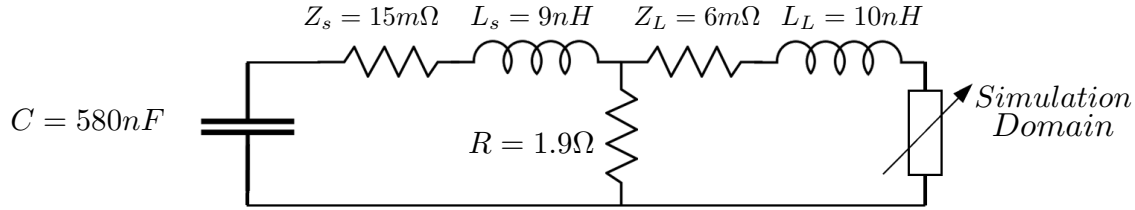
This unit works by taking in a run time parameter **circ\_currFile** in the **flash.par** which is the name of the file with the current and time data. The data should be given in MegaAmperes and nanoseconds. The unit first checks to make sure that the file exists. After running the check, a linear interpolation is used to obtain the current values for the simulation time which is then the current that is applied to the load.

This implementation lies in the **physics/sourceTerms/Circuit/ CircuitMain/FileInput** directory.

### 18.8.3 McBride

The **McBride** circuit model follows the work done by McBride et al Phys. Rev. ST Accel. Beams 13, 120401, 20102. This model is a purely reactive model that allows an input current which will be applied to a dynamic load.

This unit works by taking in a run time parameter **circ\_voltFile** in the **flash.par** which is the name of the file with the voltage and time data. The data should be given in Amps and seconds. As in **FileInput**, this unit checks if the file exists and then linearly interpolates the data. The obtained voltage is then applied as **Voc** to the following circuit LC circuit as seen in Figure 18.21. To calculate the current that is applied to the load the following equations are solved:

Figure 18.21: Schematic showing the circuit that is being solved in the **McBride** implementation.Figure 18.22: Schematic showing the circuit that is being solved in the **Cesar** implementation.

$$\begin{pmatrix} \frac{dV_c}{dt} \\ \frac{dI_{stack}}{dt} \\ \frac{dI_{load}}{dt} \end{pmatrix} = \begin{pmatrix} \frac{I_{stack} - I_{load}}{C} \\ \frac{-V_c - Z_o I_{stack} + V_{oc}}{L} \\ \frac{V_c - V_t}{L_o} \end{pmatrix}. \quad (18.280)$$

$V_t$  can be rewritten as

$$V_t = -\frac{d\Phi}{dt} = -\frac{\Phi^{new} - \Phi^{old}}{dt} \quad (18.281)$$

where  $\Phi$  is the magnetic flux in the domain. In the case of a one dimensional cylindrical domain there is a run time parameter `circ_cylLength` which is the length of the cylinder used for the flux calculation. The axial load current that is determined from the circuit model is used with Amperes Law to define the azimuthal magnetic field on the upper-r domain boundary. This field is used as a boundary condition for the implicit magnetic resistivity (diffusion) solver.

This implementation lies in the `physics/sourceTerms/Circuit/CircuitMain/McBride` directory.

#### 18.8.4 CESZAR

The **Cesar** circuit model replicates a simplified model of the Cesar Linear Transformer Driver. Unlike the McBride model, this model is both reactive and resistive driven by a discharging capacitor specified by its initial voltage. The target is a dynamic load.

This unit works by taking in a run time parameter `circ_CESZAR_V0` in the flash which defines the voltage across the capacitor in Volts. To calculate the current applied to the load the following equations are solved:

$$\begin{pmatrix} \frac{dI_s}{dt} \\ \frac{dI_l}{dt} \\ \frac{dV_c}{dt} \end{pmatrix} = \begin{pmatrix} \frac{V_c - I_s(Z_s + R) + I_l R}{L_s} \\ \frac{I_s R - I_l(Z_l + R) - V_t}{L_l} \\ \frac{-I_s}{C} \end{pmatrix}. \quad (18.282)$$

$V_t$  can be rewritten as

$$V_t = -\frac{d\Phi}{dt} = -\frac{\Phi^{new} - \Phi^{old}}{dt} \quad (18.283)$$

where  $\Phi$  is the magnetic flux in the domain. In the case of a one dimensional cylindrical domain there is a run time parameter `circ_cylLength` which is the length of the cylinder. The axial load current that is determined from the circuit model is used with Amperes Law to define the azimuthal magnetic field on the upper-r domain boundary. This field is used as a boundary condition for the implicit magnetic resistivity (diffusion) solver.

This implementation lies in the `physics/sourceTerms/Circuit/CircuitMain/CESZAR` directory.

## Chapter 19

# Diffusive Terms

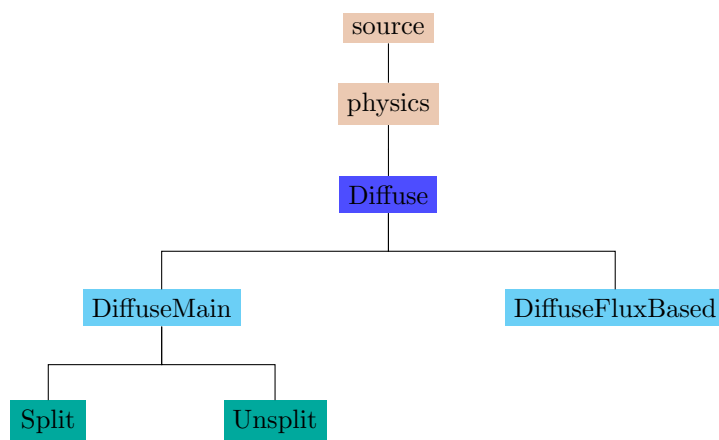


Figure 19.1: The organizational structure of the `Diffuse` unit.

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

### 19.1 Diffuse Unit

The `Diffuse` unit contains four things:

1. Flux-Based Diffusion Solvers: These are *explicit* diffusion solvers used to compute fluxes for *specific processes*, such as thermal conduction or magnetic resistivity. These fluxes can then be added to the total fluxes in the hydrodynamic solvers (see [Section 19.1.1](#)).
2. General Implicit Diffusion Solvers: The `Diffuse` unit also contains *implicit* diffusion solvers that are *general purpose*. These subroutines can be used to implicitly solve scalar diffusion problems in FLASH (see [Section 19.1.2](#)).
3. Flux Limiters: The `Diffuse` unit contains a general purpose subroutine which modifies diffusion coefficients to incorporate flux limiters (see [Section 19.1.3](#)).
4. Isotropic Thermal Diffusion: The `Diffuse` unit contains code which uses the general purpose implicit isotropic diffusion solver to solve an equation which models isotropic thermal conduction in a 3T plasma (see [Section 19.1.4](#)).

5. Anisotropic Thermal Diffusion: The `Diffuse` unit contains code which uses the general purpose implicit anisotropic diffusion solver to solve an equation which models anisotropic thermal conduction in a 3T plasma (see [Section 19.1.5](#))

### 19.1.1 Diffuse Flux-Based implementations

There are some important differences between other physics units and the flux-based `Diffuse` implementations:

- `DiffuseFluxBased` does not operate by modifying solution data arrays (like `UNK`, *etc.*) directly. `DiffuseFluxBased` modifies **flux arrays** instead.
- `DiffuseFluxBased` therefore depends on another physics unit to
  1. define and initialize the flux arrays (before `DiffuseFluxBased` calls add to them), and to
  2. apply the fluxes to actually update the solution data in `UNK`.
- `DiffuseFluxBased` calls that implement the diffusive effects are not made from `Driver.evolveFlash` as for other physics units and the `DiffuseMain` subunit. Rather, those `DiffuseFluxBased` calls need to be made from cooperating code unit that defines, initializes, and applies the flux arrays. As of FLASH4, among the provided unit implementations only the PPM implementation of the `Hydro` unit does this (by calls in the `hy_ppm_sweep` routine).
- The `DiffuseFluxBased` routines that implement the diffusive effects are called separately for each flux direction for each block of cells.

Other `Hydro` implementations, in particular the MHD implementations, currently implement some diffusive effects in their own flux-based way that does not use the `DiffuseFluxBased` unit.

To use `DiffuseFluxBased` routines of the `Diffuse` unit, a simulation should

- include the `Diffuse` unit using an option like `-with-unit=physics/Diffuse/DiffuseFluxBased` on the setup line, or `REQUIRES physics/Diffuse/DiffuseFluxBased` in a Config file;
- include a unit that makes `Diffuse` calls, currently the PPM implementation of `Hydro`;
- set `useXXX` runtimes parameters appropriately (see below).

The appropriate calls to `DiffuseFluxBased` routines will then be made by the following `Hydro` code (which can be found in `hy_ppm_sweep.F90`):

```
call Diffuse_therm(sweepDir, igeom, blockList(blk), numCells,&
                  blkLimits, blkLimitsGC, primaryLeftCoord, &
                  primaryRghtCoord, tempFlx, tempAreaLeft)

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

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

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



### 19.1.2 General Implicit Diffusion Solver

FLASH contains a general purpose implicit diffusion solver that can be used to include physics such as electron thermal conduction and multigroup diffusion. The actual code for these solvers is located in `GridSolvers`. Two implementations of these solvers exist in the `Diffuse` unit. An older and less useful version lies under the `Split` directory. A newer solver which uses the HYPRE library is contained in the `Unsplit` directory. As the name suggests, the split solver is directionally split (default: Strang) to interact with the pencil grid infrastructure of the `GridSolvers/Pfft` implementation of the `Grid` unit. The runtime parameter `useDiffuse` must be set to `.true.` to activate the general purpose implicit solver.

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

$$A \frac{\partial f}{\partial t} + C f = \nabla \cdot B \nabla f + D, \quad (19.1)$$

where  $f$  is the variable to be diffused in time, and  $A$ ,  $B$ ,  $C$ ,  $D$  are parameters with spatial variation. The infrastructure is provided for a general diffusion equation and can be customized to solve specific equations. The API function `Diffuse_solveScalar` provides a comprehensive interface to solve different types of equations.

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

$$A^n \frac{f^{n+1} - f^n}{\delta t} + C^n f^{n+1} = \theta (\nabla \cdot B^n \nabla f^{n+1}) + (1 - \theta) (\nabla \cdot B^n \nabla f^n) + D, \quad (19.2)$$

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

$$A^n \frac{f^{n+\frac{1}{3}} - f^n}{\delta t} + \frac{1}{3} C^n f^{n+\frac{1}{3}} = \theta \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^{n+\frac{1}{3}}) + (1 - \theta) \frac{\partial}{\partial x} (B^n \frac{\partial}{\partial x} f^n) + \frac{1}{3} D \quad (19.3)$$

$$A^n \frac{f^{n+\frac{2}{3}} - f^{n+\frac{1}{3}}}{\delta t} + \frac{1}{3} C^n f^{n+\frac{2}{3}} = \theta \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{2}{3}}) + (1 - \theta) \frac{\partial}{\partial y} (B^n \frac{\partial}{\partial y} f^{n+\frac{1}{3}}) + \frac{1}{3} D \quad (19.4)$$

$$A^n \frac{f^{n+1} - f^{n+\frac{2}{3}}}{\delta t} + \frac{1}{3} C^n f^{n+1} = \theta \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+1}) + (1 - \theta) \frac{\partial}{\partial z} (B^n \frac{\partial}{\partial z} f^{n+\frac{2}{3}}) + \frac{1}{3} D \quad (19.5)$$

$\theta$	Scheme
0.0	Explicit
0.5	Crank Nicholson
1.0	Backward Euler

**Implicitness:** It should be noted that the parameters  $A$ ,  $B$ ,  $C$  are time lagged and evaluated at  $t^n$ . Whereas keeping with the spirit of the implicit scheme these should have been evaluated at  $t^{n+1}$ . Evaluating them at  $t^{n+1}$  makes the problem non-linear for non-constant values of  $A$ ,  $B$  and  $C$ . These equations are much more complicated to solve. Although time lagged coefficient essentially linearized the non-linear problem, the trade off is the scheme is no longer completely implicit. The scheme is fully implicit only in the constant case. This could possibly impact the overall stability characteristics of the scheme. In general, implicit schemes are numerically stable for any time step, so the runtime parameter `dt_diff_factor` could be set to a large value to ignore the diffusion time. However, there are cases in which HYPRE will have trouble converging on a solution, and decreasing `dt_diff_factor` less than 1.0 may resolve the issue.

The Laplacian term is discretized using a central difference scheme as follows,

$$\begin{aligned} \nabla \cdot B \nabla f = & \frac{B_{i+\frac{1}{2},j,k}(f_{i+1,j,k} - f_{i,j,k}) - B_{i-\frac{1}{2},j,k}(f_{i,j,k} - f_{i-1,j,k})}{\delta x^2} \\ & + \frac{B_{i,j+\frac{1}{2},k}(f_{i,j+1,k} - f_{i,j,k}) - B_{i,j-\frac{1}{2},k}(f_{i,j,k} - f_{i,j-1,k})}{\delta y^2} \\ & + \frac{B_{i,j,k+\frac{1}{2}}(f_{i,j,k+1} - f_{i,j,k}) - B_{i,j,k-\frac{1}{2}}(f_{i,j,k} - f_{i,j,k-1})}{\delta z^2} \end{aligned} \quad (19.6)$$

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

$$B_{i+\frac{1}{2},j,k} = \frac{B_{i+1,j,k} + B_{i,j,k}}{2}, \quad (19.7)$$

$$B_{i+\frac{1}{2},j,k} = \frac{B_{i-1,j,k} + B_{i,j,k}}{2}, \quad (19.8)$$

$$B_{i,j+\frac{1}{2},k} = \frac{B_{i,j+1,k} + B_{i,j,k}}{2}, \quad (19.9)$$

$$B_{i,j-\frac{1}{2},k} = \frac{B_{i,j-1,k} + B_{i,j,k}}{2}, \quad (19.10)$$

$$B_{i,j,k+\frac{1}{2}} = \frac{B_{i,j,k+1} + B_{i,j,k}}{2}, \quad (19.11)$$

$$B_{i,j,k-\frac{1}{2}} = \frac{B_{i,j,k-1} + B_{i,j,k}}{2} \quad (19.12)$$

**Paramesh:** In coming up with a numerical scheme in AMR, it is important to ensure that fluxes are conserved at fine-coarse boundary. This is typically done in FLASH using the `Grid.conserveFluxes` routine. However this is an option which is not available to the implicit solver. Since it uses HYPRE to solve the linear system, the flux conversation should be embedded in the system being solved by HYPRE.

Looking at standard diffusion equation mentioned earlier, for simplicity we set  $C = D = 0$ .

So,

$$A \frac{\partial f}{\partial t} = \nabla \cdot B \nabla f, \quad (19.13)$$

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

$$\vec{F} = B \nabla f, \text{ and} \quad (19.14)$$

$$A \frac{\partial f}{\partial t} = \nabla \cdot \vec{F}, \quad (19.15)$$

We discretize the equation as follows,

$$A^n \frac{f^{n+1} - f^n}{\delta t} \partial v = \theta(\vec{F}^{n+1} \cdot \partial \vec{s}) + (1 - \theta)(\vec{F}^n \cdot \partial \vec{s}) \quad (19.16)$$

$$(19.17)$$

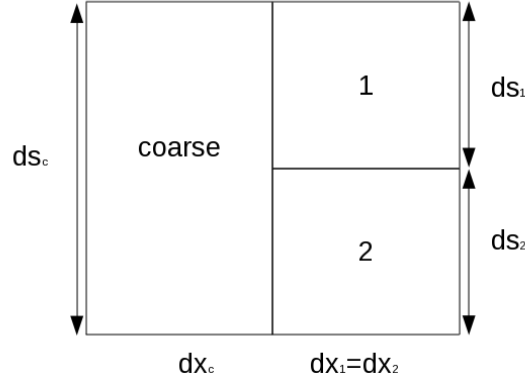
The idea is then to compute  $\vec{F} \cdot \partial \vec{s}$  on the fine-coarse boundary so as to conserve flux. In a 1D problem the  $\vec{F} \cdot \partial \vec{s}$  on a fine-coarse boundary is computed as

$$(F^L)_{i+\frac{1}{2}} = (F^R)_{i+\frac{1}{2}} = B_{i+\frac{1}{2}} \frac{f_{i+1} - f_i}{\frac{\delta x_{i+1} + \delta x_i}{2}}, \quad (19.18)$$

$$(F^L)_{i-\frac{1}{2}} = (F^R)_{i-\frac{1}{2}} = B_{i-\frac{1}{2}} \frac{f_i - f_{i-1}}{\frac{\delta x_{i-1} + \delta x_i}{2}}. \quad (19.19)$$

$$(19.20)$$

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



$$F_c ds_c = F_1 ds_1 + F_2 ds_2 \quad (19.21)$$

Where,

$$F_1 = B_{face} \frac{f_1 - f_c}{\frac{\delta x_1 + \delta x_c}{2}} \quad (19.22)$$

$$F_2 = B_{face} \frac{f_2 - f_c}{\frac{\delta x_2 + \delta x_c}{2}} \quad (19.23)$$

The value of  $B_{face}$  can then be evaluated from surrounding cells using paramesh interpolation.

**Usage:** To use the “Split” implementation of the Diffuse unit, a simulation should include the Diffuse unit using an option like `-with-unit=physics/Diffuse/DiffuseMain/Split` on the setup line, or add `REQUIRES physics/Diffuse/DiffuseMain/Split` in the Config file. To get the “Unsplit” implementation, replace `Split` with `Unsplit`. Split and Unsplit solver are mutually exclusive implementations and cannot exist in the same simulation.

Table 19.1: comparison of capabilities of Split and Unsplit diffusion.

Capability	Split	Unsplit
Paramesh	no	yes
Uniform grid	yes	yes
Cartesian 1D, 2D	yes	yes
Cartesian 3D	yes	yes
Cylindrical 1D	yes	yes
Cylindrical 2D (R-Z)	yes	yes
Spherical 1D	yes	yes

#### 19.1.2.1 Boundary Conditions

Unlike much of the rest of FLASH, the general purpose implicit diffusion solver does not implement boundary conditions explicitly through guard cell fills. Instead the diffusion matrix is modified to directly include

Table 19.2: General Implicit Diffusion Solver Boundary Conditions

Name	Integer	Effect
Neumann	GRID_PDE_BND_NEUMANN	$\nabla f \cdot \mathbf{n} = 0$
Dirichlet	GRID_PDE_BND_DIRICHLET	$f = c$
Vacuum	VACUUM	something like $\frac{1}{2}f + \frac{1}{4}\nabla f \cdot \mathbf{n} = 0$
Outstream	OUTSTREAM	free streaming radiation at outer boundary (1D spherical only)

boundary conditions. The `bcTypes` and `bcValues` arguments to `Diffuse_solveScalar` are used to indicate the boundary condition to use. Each of these arguments is an array of length six corresponding to the six faces of the domain: `imin`, `imax`, `jmin`, `jmax`, `kmin`, `kmax` in that order. The `bcTypes` argument is used to set the type of boundary condition to use. Several integer values are acceptable that are defined in the “constants.h” header file. The allowed integer values and the mathematical effect are shown in [Table 19.2](#). When a Dirichlet (constant value) boundary conditions is required, `bcTypes` stores the value to use.

### 19.1.3 Flux Limiters

The `Diffuse` unit contains the subroutine `Diffuse_fluxLimiter` which modifies a mesh variable representing a diffusion coefficient over the entire mesh to include a flux-limit. The subroutine modifies the diffusion coefficient to smoothly transition to some maximum flux,  $q_{\max}$ . The subroutine takes the following arguments:

subroutine `Diffuse_fluxLimiter(idcoef, ifunc, ifl, mode, blkcnt, blklst)`

where:

- `idcoef` [integer,input] Mesh variable representing the diffusion coefficient (e.g. `COND_VAR` for electron thermal conduction)
- `ifunc` [integer,input] Mesh variable representing the function being diffused (e.g. `TELE_VAR` for thermal conduction)
- `ifl` [integer,input] Mesh variable representing the flux limit (e.g. `FLLM_VAR` for electron thermal conduction)
- `mode` [integer,input] The type of flux-limiter to apply
- `blkcnt` [integer,input] Number of blocks
- `blklst` [integer array, input] List of blocks

The subroutine modifies the mesh variable whose index is given by `idcoef` by applying a flux-limit. Many methods of incorporating flux-limits into the diffusion coefficient exist. `Diffuse_fluxLimiter` currently supports three of these: “harmonic”, “min/max”, “Larsen”, and “none”. The `mode` argument selects the specific method to use. The modes are integers defined in the “constants.h” header file. String representations also exist and can be converted to the integer values using the `RuntimeParameters_mapStrToInt` subroutine. The string representations can be used to create runtime parameters which represent a flux-limiter mode. [Table 19.3](#) shows the integer constants used as arguments for `Diffuse_fluxLimiter` as well as the string representations of these constants.

Each flux limiter mode modifies the existing diffusion coefficient to incorporate the flux limiter in a different way. If  $D$  is the original diffusion coefficient and  $D_{fl}$  is the flux-limited diffusion coefficient then:

- the “none” flux limiter sets:

$$D_{fl} = D \quad (19.24)$$

- the “harmonic” flux limiter sets:

$$D_{fl} = \frac{1}{\frac{1}{D} + \frac{|\nabla f|}{q_{\max}}} \quad (19.25)$$

Table 19.3: String and Integer Representations of Flux-Limiter Modes

Mode	Integer Constant	String Representation
None	FL_NONE	"fl_none"
Harmonic	FL_HARMONIC	"fl_harmonic"
Min/Max	FL_MINMAX	"fl_minmax"
Larsen	FL_LARSEN	"fl_larsen"
Levermore-Pomraning	FL_LEVPOM	"fl_levermorepomraning1981"

- the "Larsen" flux limiter sets:

$$D_{fl} = \frac{1}{\left[ \left( \frac{1}{D} \right)^2 + \left( \frac{|\nabla f|}{q_{\max}} \right)^2 \right]^{1/2}} \quad (19.26)$$

- the "min/max" flux limiter sets:

$$D_{fl} = \min \left( D, \frac{q_{\max}}{|\nabla f|} \right) \quad (19.27)$$

where  $f$  is the value of the function being diffused. The gradient of  $f$  is computed using a second order central difference.

#### 19.1.4 Isotropic Thermal Diffusion

The **Diffuse** unit also contains a subroutine which solves the diffusion equations for the cases of electron and ion thermal conduction. This code is in the **diff\_advanceTherm** subroutine that is called by **Diffuse**. The isotropic diffusion equation for electrons has the form:

$$\rho c_{v,\text{ele}} \frac{\partial T_{\text{ele}}}{\partial t} = \nabla \cdot K_{\text{ele}} \nabla T_{\text{ele}} \quad (19.28)$$

where  $\rho$  is the mass density,  $c_{v,\text{ele}}$  is the electron specific heat, and  $K_{\text{ele}}$  is the electron thermal conductivity. Likewise, the isotropic diffusion equation for ions is:

$$\rho c_{v,\text{ion}} \frac{\partial T_{\text{ion}}}{\partial t} = \nabla \cdot K_{\text{ion}} \nabla T_{\text{ion}} \quad (19.29)$$

where each variable is as previously described, but now for ions.

The conductivity is computed by the **Conductivity** unit (see [Section 23.1](#) for more information). This equation is solved implicitly over a time step to avoid overly restricted time step constraints. In many cases a flux-limiter is needed in situations where overly large temperature gradients are present. The flux-limit used for electron thermal conduction,  $q_{\max,\text{ele}}$ , is defined as:

$$q_{\max,\text{ele}} = \alpha_{\text{ele}} n_{\text{ele}} k_B T_{\text{ele}} \sqrt{\frac{k_B T_{\text{ele}}}{m_{\text{ele}}}} \quad (19.30)$$

where:

- $\alpha_{\text{ele}}$  is the electron conductivity flux-limiter coefficient
- $n_{\text{ele}}$  is the electron number density
- $k_B$  is the Boltzmann constant
- $m_{\text{ele}}$  is the electron mass

There is an analogous flux-limit for ion thermal conduction as well.

The `Diffuse_solveScalar` subroutine is used for the implicit solve. The use of the isotropic thermal diffusion solver is demonstrated in several simulations. See [Section 35.8.1](#) and [Section 35.7.5](#) for examples. For thermal diffusion to function, the `useConductivity` and `useDiffuse` runtime parameters must be set to `.true..` The following runtime parameters control the behavior of the electron and ion thermal conduction package:

- `diff_useEleCond`: Set to `.true.` to activate electron thermal conduction (default behavior is isotropic)
- `diff_useIonCond`: Set to `.true.` to activate ion thermal conduction (default behavior is isotropic)
- `diff_eleFlMode`: String indicating which flux-limiter mode to use for electron conduction. Valid values are described in [Section 19.1.3](#).
- `diff_ionFlMode`: String indicating which flux-limiter mode to use for ion conduction. Valid values are described in [Section 19.1.3](#).
- `diff_eleFlCoef`: Sets the value of  $\alpha_{\text{ele}}$
- `diff_ionFlCoef`: Sets the value of  $\alpha_{\text{ion}}$
- `diff_thetaImplct`: A number between 0 and 1. When set to 0, an explicit treatment is used for the electron thermal diffusion solve. When set to 1, a fully implicit solve is used.
- `diff_ionThetaImplct`: A number between 0 and 1. When set to 0, an explicit treatment is used for the ion thermal diffusion solve. When set to 1, a fully implicit solve is used.

The runtime parameters:

```
diff_eleXlBoundaryType
diff_eleXrBoundaryType
diff_eleYlBoundaryType
diff_eleYrBoundaryType
diff_eleZlBoundaryType
diff_eleZrBoundaryType
```

```
diff_ionXlBoundaryType
diff_ionXrBoundaryType
diff_ionYlBoundaryType
diff_ionYrBoundaryType
diff_ionZlBoundaryType
diff_ionZrBoundaryType
```

are strings which are used to set the boundary conditions for electron and ion conduction on each of the six faces of the domain. Acceptable values are:

- “DIRICHLET”, “Dirichlet”, or “dirichlet” to indicate a Dirichlet boundary condition where the appropriate  $T$  is set to zero on the boundary
- “neumann”, “zero-gradient”, to indicate a Neumann boundary condition where the gradient of the appropriate  $T$  is set to zero on the boundary

### 19.1.5 Anisotropic Thermal Diffusion

The `Diffuse` unit also has the capability to solve the anisotropic thermal diffusion equations for both electrons and ions. The anisotropic equations are quite different from the isotropic ones of [\(19.28\)](#) and [\(19.29\)](#) above. The general anisotropic thermal diffusion equation has the form:

$$\rho c_v \frac{\partial T}{\partial t} = \nabla \cdot (K_{\parallel} \mathbf{b} (\mathbf{b} \cdot \nabla T) + K_{\perp} \mathbf{b} \times (\nabla T \times \mathbf{b}) + K_{\wedge} (\mathbf{b} \times \nabla T)) \quad (19.31)$$

where  $\mathbf{b}$  is a unit vector in the direction of the magnetic field,  $K_{\parallel}$  is the thermal conductivity in the direction of the field,  $K_{\perp}$  is the thermal conductivity perpendicular to the field, and  $K_{\wedge}$  is thermal conductivity perpendicular to both the field and the temperature gradient. There are different values of these conductivities for both electrons and ions, which govern the diffusion of the electron temperature and ion temperature, respectively (in a 3T plasma). The particular values used for the conductivities are calculated by the `Conductivity` unit (see [Section 23.1](#) for more information). The specific heat  $c_v$  will also have a different value for electrons and ions. The only boundary condition currently supported for anisotropic diffusion is the homogeneous Neumann condition (see [19.2](#)).

The anisotropic equation ([19.31](#)) is solved within the `Unsplit` scheme of the `Diffuse` unit. Since it requires magnetic fields, it is also necessary for the code to be using the `Unsplit Staggered Mesh (USM)` MHD solver (see [Section 15.3.3](#) for more details). A setup shortcut has been provided (`+supportAnisoCond`), which should be used to enable all functionality available for anisotropic thermal diffusion. Additional runtime parameters control the behavior of the anisotropic thermal conduction package:

- `diff_anisoCondForEle`: Set to `.true.` to activate anisotropic electron thermal conduction in 3T case (or for general anisotropic thermal diffusion in 1T case)
- `diff_anisoCondForIon`: Set to `.true.` to activate anisotropic ion thermal conduction (3T only)
- `diff_useCrossCond`: Set to `.true.` to include the  $K_{\wedge}$  terms in the anisotropic diffusion equation
- `gr_hypresSlopeLimType`: Controls the type of slope limiter used for transverse temperature gradients. The default value is `HYPRESL_MC` and other acceptable values are: `HYPRESL_NONE`, `HYPRESL_MINMOD`

It is possible to have both isotropic and anisotropic thermal diffusion. For example, if `diff_anisoCondForIon` is set to `.false.`, but `diff_useIonCond` is `.true.` and `diff_anisoCondForEle` is `.true.`, then the ion temperature will diffuse isotropically and the electron temperature will diffuse anisotropically. Note: the runtime parameter `diff_anisoCondForEle` also controls anisotropic thermal diffusion in 1T applications.

Slope-limiting for transverse temperature gradients was implemented, because without it, the  $\mathbf{b} \cdot \nabla T$  term in ([19.31](#)) can cause heat to flow from a cold cell to a hotter one (a violation of the 2nd law of thermodynamics). In extreme cases, this can lead to negative temperatures, but `gr_hypreusefloor` should be set to `.true.` with an appropriate value for `gr_hyprefloor` (default values are already set to automatically avoid negative temperatures during diffusion). If using AMR, the slope-limiter is not applied at fine/coarse boundaries, so inaccurate temperature decreases are still technically possible. The default slope-limiter is `HYPRESL_MC`. `HYPRESL_MINMOD` is more diffusive, so it can be used if inaccurate temperature decreases are problematic. To turn off this slope-limiter feature, use `HYPRESL_NONE`. In general, a solver such as `HYPRE_GMRES` should be used when using a slope-limiter, because slope-limiting can lead to an asymmetric matrix during the implicit diffusion solve done by `HYPRE`. The solver type is set with the runtime parameter `gr_hypresolvertype`, and the default value is `HYPRE_GMRES` when `+supportAnisoCond` has been used during setup. If the `HYPRE_PCG` solver is used, slope-limiting will need to be turned off.

### 19.1.6 Magnetic Diffusion

FLASH is able to solve the resistive MHD equations with contributions to the induction equation of the form

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times \mathbf{E} \quad (19.32)$$

$$= \nabla \times \left( \frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J} + \frac{(\eta_{\perp} - \eta_{\parallel})}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \times \mathbf{b} \right). \quad (19.33)$$

FLASH provides two methods to add resistive MHD to a simulation, either as a contribution to the unsplit `Hydro` fluxes, or implicitly as a an operator split update using the `Unified HYPRE` solver. This is controlled with the runtime parameter `resistivitySolver`. When set to `"implicit"` the implicit solver is used.

The exact form of the generalized Ohm's Law used in Eq. 19.32 is determined by the parameter `resistivityForm` and described in Table 19.4. Currently the "implicit" solver only uses the "parallel" form for isotropic resistivity.

Table 19.4: Options for the runtime parameter `resistivityForm` and the resultant contribution to the generalized Ohm's Law.

<code>resistivityForm</code>	$\mathbf{E}$
"parallel"	$\frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J}$
"perpendicular"	$\frac{\eta_{\perp}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \times \mathbf{b}$
"fullAniso"	$\frac{\eta_{\parallel}}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} \mathbf{J} + \frac{(\eta_{\perp} - \eta_{\parallel})}{\epsilon_0 \omega_{pe}^2 \tau_{ei}} (\mathbf{b} \times \mathbf{J}) \times \mathbf{b}$

The "implicit" Magnetic Diffusion solver uses the Unified HYPRE solver to advance Eq. 19.32 implicitly. All the runtime parameters in Table 8.7 that are specific to this solver are prepended with "gr\_uhpreMag\_". Eq. 19.32 can be expressed in the form of Eq. 8.93 with the tensors

$$A_i = 1 \quad (19.34)$$

$$B_{ijkl} = \eta (\delta_{il} \delta_{kj} - \delta_{ij} \delta_{kl}) \quad (19.35)$$

$$C_{ij} = 0 \quad (19.36)$$

$$D_i = 0 \quad (19.37)$$

$$E_{ijk} = 0. \quad (19.38)$$

In order to suppress the generation of spurious  $\nabla \cdot \mathbf{B}$  errors Eq. 19.32 may be augmented with a parabolic divergence cleaning term (Dedner et. al. 2002) that damps the divergence of the magnetic field,

$$\frac{\partial \mathbf{B}}{\partial t} = c_p^2 \nabla (\nabla \cdot \mathbf{B}). \quad (19.39)$$

Here the constant  $c_p$  is taken as the maximum value over the whole domain of  $\max(\eta, c_f)$ , where  $c_f$  is the fast magnetosonic speed. This can be expressed as a  $B_{ijkl}$  tensor

$$B_{ijkl} = c_p^2 \delta_{ik} \delta_{jl}. \quad (19.40)$$

In cylindrical geometries the evolution equations for the radial and axial magnetic field components may still be expressed in conservative form and so we only need the  $B_{ijkl}$  tensor in Eqs 19.34 for  $i = r, z$ . The azimuthal field equation though will introduce some geometric source terms. These terms can be expressed in the general form of Eq. 8.98, but in particular for problems with spatially varying resistivity,  $\eta$ , have been observed to introduce numerical artifacts and instability, especially at fine-coarse boundaries. The conservation form may be restored by evolving  $\frac{B_{\phi}}{r}$ , which will introduce an additional  $E$ -factor term,

$$\partial_t \frac{B_{\phi}}{r} \sim \frac{1}{r} \partial_r \left[ r \eta \left( \partial_r \frac{B_{\phi}}{r} + \frac{2}{r} \frac{B_{\phi}}{r} \right) \right] \quad (19.41)$$

$$E_{\phi\phi r} = 2\eta. \quad (19.42)$$

This form of the equations will be evolved when the grid `geometry` is "cylindrical" and the runtime parameter `diff_conserveAngfield` = .TRUE.. This should not be confused with the Hydro runtime parameter `conserveAngField`, which serves a similar purpose in the MHD solver.

While the above described conservative formulation works well for evolving the magnetic field in cylindrical coordinates, for problems with orders of magnitude difference in resistivity across material interfaces, as is typical in Z-pinch simulations the resultant fields can cause some small scale discrepancies in the calculated ohmic heating at fine-coarse boundaries that can be significantly detrimental to the overall simulation.



To overcome this the finite-volume discretization of Sec. 8.10.4.2 needs to be modified so that rather than the flux of  $\frac{B_\phi}{r}$  being conserved at fine-coarse boundaries the current density  $\mathbf{J} = \nabla \times \mathbf{B}$  is conserved.

$$\partial_t r B_\phi \sim r \left[ \frac{\eta}{r} \partial_r (r B_\phi) \right] \quad (19.43)$$

$$\int_{V_\alpha} \partial_t r B_\phi dV \approx \frac{V_\alpha r_\alpha}{\Delta r} \left[ \frac{\eta_{\alpha+1/2}}{r_{\alpha+1/2}} \partial_r (r B_\phi)|_{\alpha+1/2} - \frac{\eta_{\alpha-1/2}}{r_{\alpha-1/2}} \partial_r (r B_\phi)|_{\alpha-1/2} \right] \quad (19.44)$$

The discretization in Eq. 19.44 is the default for cylindrical geometry.

The boundary conditions imposed on the HYPRE matrix are determined by the the `ab_boundary_type` runtime parameters Table.

Table 19.5: Unified HYPRE solver boundary conditions used for magnetic diffusion

<code>ab_boundary_type</code>	Description
<code>periodic</code>	Periodic ('wrap-around')
<code>reflect,reflecting</code>	the normal vector components change sign
<code>outflow</code>	Zero-gradient boundary condition in normal direction
<code>circuit</code>	Dirichlet boundary condition taken from the <code>Circuit</code> unit
<code>user</code>	Dirichlet boundary condition

#### 19.1.6.1 Ohmic Heating

The Ohmic heating from magnetic diffusion when using the "implicit" solver is computed from the updated magnetic fields using the implicit  $\theta$  parameter,

$$\partial_t \rho_i \epsilon_i = (1 - \theta) Q_i^n + \theta Q_i^{n+1}. \quad (19.45)$$

In the above  $Q_i^n$  is the rate of heat added to the cell  $i$  at time  $t^n$ . The solver supports two methods for computing  $Q$  depending on the runtime parameter `diff_useQohmEdgeMethod`. When set to `.FALSE`,  $Q$  is calculated as a direct centered difference of

$$Q^n = \nabla \times (\eta \nabla \times \mathbf{B}). \quad (19.46)$$

In Z-pinch simulations where an artificially high vacuum resistivity is used this discretization this can bias heating into the vacuum at a conductor-vacuum interface since the model doesn't take into account the relative resistance of neighboring cells. When `diff_useQohmEdgeMethod = .TRUE`, the solver will use a parallel resistor model at cell edges to distribute the current density at an edge to the neighboring cells. A diagram of this model is shown in Fig. 19.2. The equivalent resistance is determined from the resistivity and the geometry of the cell quadrants that meet at the edge. The current density is calculated from finite-differences at the edge center and the total current through the cross-sectional area made up of the joining quadrants. The parallel resistor is solved for the current through each quadrant where the dissipated energy is taken as the ohmic heating in the quadrant. The heating in the entire cell is summed over the contributions from all edges on the cell.

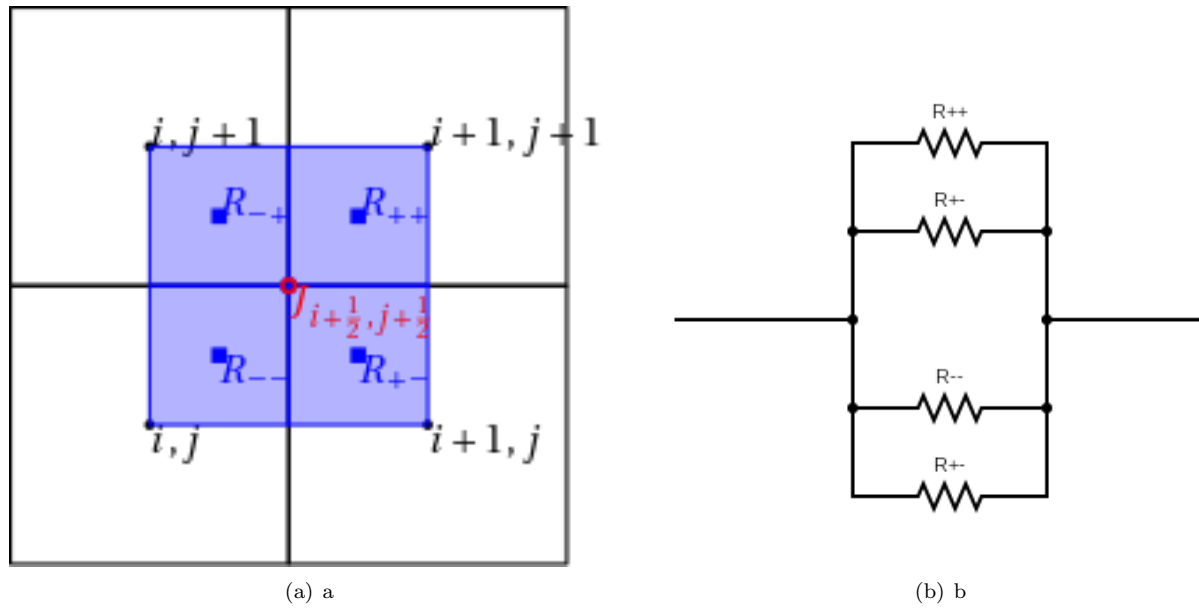


Figure 19.2: **19.2(a)** cell quadrants that share the edge at  $i + \frac{1}{2}, j + \frac{1}{2}$  with current density  $J_{i + 1/2, j + 1/2}$ . **19.2(b)** The parallel resistor model used to calculate the Ohmic heating in the quadrants of **19.2(a)**

## Chapter 20

# Gravity Unit

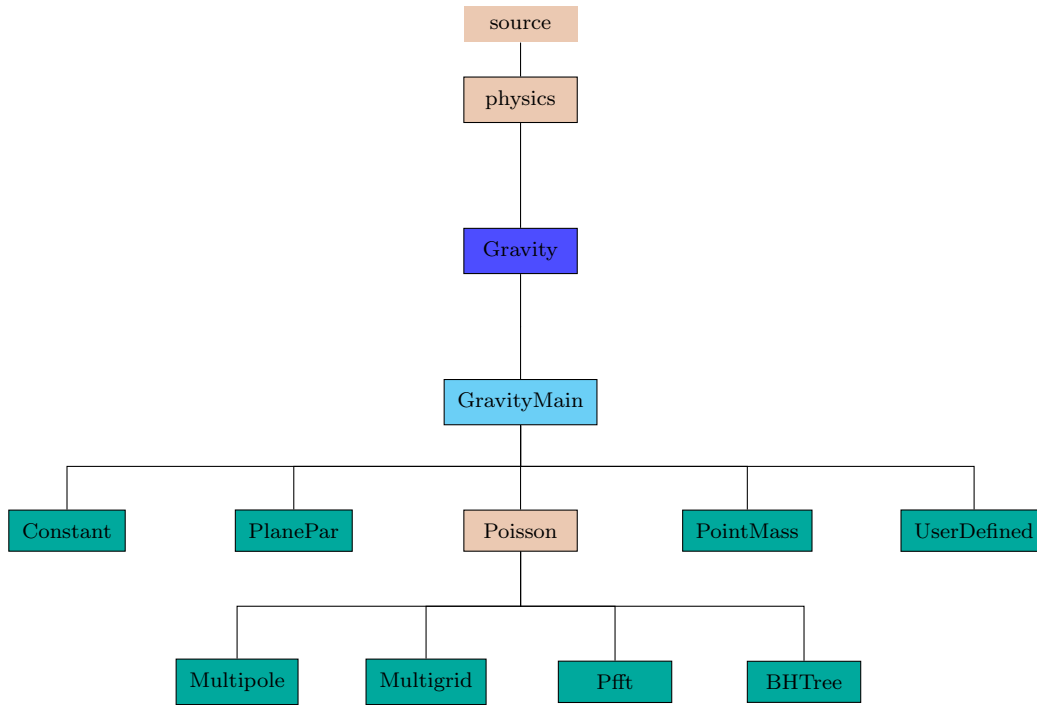


Figure 20.1: The **Gravity** unit directory tree.

### 20.1 Introduction

The **Gravity** unit supplied with FLASH4 computes gravitational source terms for the code. These source terms can take the form of the gravitational potential  $\phi(\mathbf{x})$  or the gravitational acceleration  $\mathbf{g}(\mathbf{x})$ ,

$$\mathbf{g}(\mathbf{x}) = -\nabla\phi(\mathbf{x}) . \quad (20.1)$$

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

$$\nabla^2\phi(\mathbf{x}) = 4\pi G\rho(\mathbf{x}) , \quad (20.2)$$

where  $G$  is Newton's gravitational constant. In the latter case, either periodic or isolated boundary conditions can be applied.

## 20.2 Externally Applied Fields

The FLASH distribution includes three externally applied gravitational fields, along with a placeholder module for you to create your own. Each provides the acceleration vector  $\mathbf{g}(\mathbf{x})$  directly, without using the gravitational potential  $\phi(\mathbf{x})$  (with the exception of `UserDefined`, see below).

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.

### 20.2.1 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`.

### 20.2.2 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.

### 20.2.3 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.

### 20.2.4 User-Defined Gravitational Field

The `UserDefined` implementation is a placeholder module for the user to create their own external gravitational field. All of the subroutines in this module are stubs, and the user may copy these stubs to their setup directory to write their own implementation, either by specifying the gravitational acceleration directly or by specifying the gravitational potential and taking its gradient. If your user-defined gravitational field is time-varying, you may also want to set `PPDEFINE FLASH_GRAVITY_TIMEDEP` in your setup's Config file.

## 20.3 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 (20.2). This unit's implementation can also return the acceleration field  $\mathbf{g}(\mathbf{x})$  computed by finite-differencing the potential using the expressions

$$\begin{aligned} g_{x;ijk} &= \frac{1}{2\Delta x} (\phi_{i-1,j,k} - \phi_{i+1,j,k}) + \mathcal{O}(\Delta x^2) \\ g_{y;ijk} &= \frac{1}{2\Delta y} (\phi_{i,j-1,k} - \phi_{i,j+1,k}) + \mathcal{O}(\Delta y^2) \\ g_{z;ijk} &= \frac{1}{2\Delta z} (\phi_{i,j,k-1} - \phi_{i,j,k+1}) + \mathcal{O}(\Delta z^2) . \end{aligned} \tag{20.3}$$

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 8.10.2.1](#) for self gravity is appropriate for spherical or nearly-spherical mass distributions with isolated boundary conditions. For non-spherical mass distributions higher order moments of the solver must be used. Note that storage and CPU costs scale roughly as the square of number of moments used, so it is best to use this solver only for nearly spherical matter distributions.

The multigrid solver described in [Section 8.10.2.6](#) is appropriate for general mass distributions and can solve problems with more general boundary conditions.

The tree solver described in [8.10.2.4](#) is appropriate for general mass distributions and can solve problems with both isolated and periodic boundary conditions set independently in individual directions.

### 20.3.1 Coupling Gravity with Hydrodynamics

The gravitational field couples to the Euler equations only through the momentum and energy equations. If we define the total energy density as

$$\rho E \equiv \frac{1}{2} \rho v^2 + \rho \epsilon, \quad (20.4)$$

where  $\epsilon$  is the specific internal energy, then the gravitational source terms for the momentum and energy equations are  $\rho \mathbf{g}$  and  $\rho \mathbf{v} \cdot \mathbf{g}$ , respectively. Because of the variety of ways in which different hydrodynamics schemes treat these source terms, the gravity module only supplies the potential  $\phi$  and acceleration  $\mathbf{g}$ , leaving the implementation of the fluid coupling to the hydrodynamics module. Finite-difference and finite-volume hydrodynamic schemes apply the source terms in their advection steps, sometimes at multiple intermediate timesteps and sometimes using staggered meshes for vector quantities like  $\mathbf{v}$  and  $\mathbf{g}$ .

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

$$\begin{aligned} (\rho v)_i^{n+1} &= (\rho v)_i^n + \frac{\Delta t}{2} g_i^{n+1} (\rho_i^n + \rho_i^{n+1}) \\ (\rho E)_i^{n+1} &= (\rho E)_i^n + \frac{\Delta t}{4} g_i^{n+1} (\rho_i^n + \rho_i^{n+1}) (v_i^n + v_i^{n+1}). \end{aligned} \quad (20.5)$$

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

Note that finite-volume schemes do not retain explicit conservation of momentum and energy when gravity source terms are added. Godunov schemes such as PPM, require an additional step in order to preserve second-order time accuracy. The gravitational acceleration component  $g_i$  is fitted by interpolants along with the other state variables, and these interpolants are used to construct characteristic-averaged values of  $\mathbf{g}$  in each cell. The velocity states  $v_{L,i+1/2}$  and  $v_{R,i+1/2}$ , which are used as inputs to the Riemann problem solver, are then corrected to account for the acceleration using the following expressions

$$\begin{aligned} v_{L,i+1/2} &\rightarrow v_{L,i+1/2} + \frac{\Delta t}{4} (g_{L,i+1/2}^+ + g_{L,i+1/2}^-) \\ v_{R,i+1/2} &\rightarrow v_{R,i+1/2} + \frac{\Delta t}{4} (g_{R,i+1/2}^+ + g_{R,i+1/2}^-). \end{aligned} \quad (20.6)$$

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.

### 20.3.2 Tree Gravity

The **Tree** implementation of the gravity unit in `physics/Gravity/GravityMain/Poisson/BHTree` is meant to be used together with the tree solver implementation `Grid/GridSolvers/BHTree/Wunsch`. It either calculates the gravitational potential field which is subsequently differentiated in subroutine `Gravity_accelOneRow` to obtain the gravitational acceleration, or it calculates the gravitational acceleration directly. The latter approach is more accurate, because the error due to numerical differentiation is avoided, however, it consumes more memory for storing three components of the gravitational acceleration. The direct acceleration calculation can be switched on by specifying `bhtreeAcc=1` as a command line argument of the setup script.

The gravity unit provides subroutines for building and walking the tree called by the tree solver. In this version, only monopole moments (node masses) are used for the potential/acceleration calculation. It also defines new multipole acceptance criteria (MACs) that estimate the error in gravitational acceleration of a contribution of a single node to the potential (hereafter partial error) much better than purely geometrical MAC defined in the tree solver. They are: (1) the approximate partial error (APE), and (2) the maximum partial error (MPE). The first one is based on an assumption that the partial error is proportional to the multipole moment of the node. The node is accepted for calculation of the potential if

$$D^{m+2} > \frac{GM S_{\text{node}}^m}{\Delta a_{\text{p,APE}}} \quad (20.7)$$

where  $D$  is distance between the *point-of-calculation* and the node,  $M$  is the node mass,  $S_{\text{node}}$  is the node size,  $m$  is a degree of the multipole approximation and  $\Delta a_{\text{p,APE}}$  is the requested maximum error in acceleration (controlled by runtime parameter `grv_bhAccErr`). Since only monopole moments are used for the potential calculation, the most reasonable choice of  $m$  seems to be  $m = 2$ . This MAC is similar to the one used in Gadget2 (see Springel, 2005, MNRAS, 364, 1105).

The second MAC (maximum partial error, MPE) calculates the error in acceleration of a single node contribution  $\Delta a_{\text{p,MPE}}$  according to formula 9 from Salmon&Warren (1994; see this paper for details):

$$\Delta a_{\text{p,MPE}} \leq \frac{1}{D^2} \frac{1}{(1 - S_{\text{node}}/D)^2} \left( \frac{3[B_2]}{D^2} - \frac{2[B_3]}{D^3} \right) \quad (20.8)$$

where  $B_n = \sum_i m_i |\mathbf{r}_i - \mathbf{r}_0|^n$  where  $m_i$  and  $\mathbf{r}_i$  are masses and positions of individual grid cells within the node and  $\mathbf{r}_0$  is the node mass center position. Moment  $B_2$  can be easily determined during the tree build, moment  $B_3$  can be estimated as  $B_3^2 \geq B_2^3/M$ . The maximum allowed partial error in gravitational acceleration is controlled by runtime parameters `grv_bhAccErr` and `grv_bhUseRelAccErr` (see 20.4.1).

During the tree walk, subroutine `Gravity_bhNodeContrib` adds contributions of tree nodes to the gravitational potential or acceleration fields. In case of the potential, the contribution is

$$\Phi = -\frac{GM}{|\vec{r}|} \quad (20.9)$$

if **isolated** boundary conditions are used, or

$$\Phi = -GM f_{\text{EF},\Phi}(\vec{r}) \quad (20.10)$$

if periodic **periodic** or **mixed** boundary conditions are used. In case of the acceleration, the contributions are

$$\vec{a}_g = \frac{GM \vec{r}}{|\vec{r}|^3} \quad (20.11)$$

for **isolated** boundary conditions, or

$$\vec{a}_g = GM f_{\text{EF},a}(\vec{r}) \quad (20.12)$$

for **periodic** or **mixed** boundary conditions. In the above formulae,  $G$  is the constant of gravity,  $M$  is the node mass,  $\vec{r}$  is the position vector between *point-of-calculation* and the node mass center and  $f_{\text{EF},\Phi}$  and  $f_{\text{EF},a}$  are the Ewald fields for the potential and the acceleration (see below).

Boundary conditions can be isolated or periodic, set independently for each direction. If they are periodic at least in one direction, the Ewald method is used for the potential calculation (Ewald, P. P., 1921,

Ann. Phys. 64, 253). The original Ewald method is an efficient method for computing gravitational field for problems with periodic boundary conditions in three directions. Ewald speeded up evaluation of the gravitational potential by splitting it into two parts,  $Gm/r = Gm \operatorname{erf}(\alpha r)/r + Gm \operatorname{erfc}(\alpha r)/r$  ( $\alpha$  is an arbitrary constant) and then by applying Poisson summation formula on  $\operatorname{erfc}$  terms, gravitational field at position  $\vec{r}$  can be written in the form

$$\phi(\vec{r}) = -G \sum_{a=1}^N m_a \left( \sum_{i_1, i_2, i_3} A_S(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3}) + A_L(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3}) \right) = -G \sum_{a=1}^N m_a f_{\text{EF}, \Phi}(\vec{r}_a - \vec{r}), \quad (20.13)$$

the first sum runs over whole computational domain, where at position  $\vec{r}_a$  is mass  $m_a$ . Second sum runs over all neighbouring computational domains, which are at positions  $\vec{l}_{i_1, i_2, i_3}$  and  $A_S(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3})$  and  $A_L(\vec{r}, \vec{r}_a, \vec{l}_{i_1, i_2, i_3})$  are short and long-range contributions, respectively. It is sufficient to take into account only few terms in eq. 20.13. The Ewald field for the acceleration,  $f_{\text{EF}, a}$ , is obtained using a similar decomposition. We modified Ewald method for problems with periodic boundary conditions in two directions and isolated boundary conditions in the third direction and for problems with periodic boundary conditions in one direction and isolated in two directions.

The gravity unit allows also to use a static external gravitational field read from file. In this version, the field can be either spherically symmetric or planar being only a function of the z-coordinate. The external field file is a text file containing three columns of numbers representing the coordinate, the potential and the acceleration. The coordinate is the radial distance or z-distance from the center of the external field given by runtime parameters. The external field is mapped to a grid using a linear interpolation each time the gravitational acceleration is calculated (in subroutine `Gravity_accelOneRow`).

## 20.4 Usage

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

`-with-unit=physics/Gravity`

on your command line when you configure the code with `setup`. The default implementation is `Constant`, which can be overridden by including the entire path to the specific implementation in the command line or `Config` file. The other available implementations are `Gravity/GravityMain/Planepar`, `Gravity/GravityMain/Pointmass` and `Gravity/GravityMain/Poisson`. The `Gravity` unit provides accessor functions to get gravitational acceleration and potential. However, none of the external field implementations of Section 20.2 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 Poisson equation for the gravitational potential, additional storage is needed in `unk` for holding the last, as well as (usually) the previous, gravitational potential field; and, depending on the Poisson solver used, additional variables may be needed. The variables `GPOT_VAR` and `GPOT_VAR`, and others as needed, will be automatically defined in `Flash.h` in those cases. See `Gravity_potentialListOfBlocks` for more information.

### 20.4.1 Tree Gravity Unit Usage

Calculation of gravitational potential can be enabled by compiling in this unit and setting the runtime parameter `useGravity` true. The constant of gravity can be set independently by runtime parameter `grv_bhNewton`; if it is not positive, the constant `Newton` from the FLASH `PhysicalConstants` database is used. If parameters `gr_bhPhysMACTW` or `gr_bhPhysMACComm` are set, the gravity unit MAC is used and it can be chosen by setting `grv_bhMAC` to either `ApproxPartialErr` or `MaxPartialErr`. If the first one is used, the order of the multipole approximation is given by `grv_bhMPDegree`.

The maximum allowed partial error in gravitational acceleration is set with the runtime parameter `grv_bhAccErr`. It has either the meaning of an error in absolute acceleration or in relative acceleration normalized by the acceleration from the previous time-step. The latter is used if `grv_bhUseRelAccErr` is

set to True, and in this case the first call of the tree solver calculates the potential using purely geometrical MAC (because the acceleration from the previous time-step does not exist).

Boundary conditions are set by the runtime parameter `grav_boundary_type` and they can be `isolated`, `periodic` or `mixed`. In the case of mixed boundary conditions, runtime parameters `grav_boundary_type_x`, `grav_boundary_type_y` and `grav_boundary_type_z` specify along which coordinate boundary conditions are periodic and isolated (possible values are `periodic` or `isolated`). Arbitrary combination of these values is permitted, thus suitable for problems with planar resp. linear symmetry. It should work for computational domain with arbitrary dimensions. The highest accuracy is reached with blocks of cubic physical dimensions.

If runtime parameter `grav_boundary_type` is `periodic` or `mixed`, then the Ewald field for appropriate symmetry is calculated at the beginning of the simulation. Parameter `grv_bhEwaldSeriesN` controls the range of indices  $i_1, i_2, i_3$  in (eq. 20.13). There are two implementations of the Ewald method: the new one (default) requires less memory and it should be faster and of comparable accuracy as the old one. The default implementation computes Ewald field minus the singular  $1/r$  term and its partial derivatives on a single cubic grid, and the Ewald field is then approximated by the first order Taylor formula. Parameter `grv_bhEwaldNPer` controls number of grid points in the  $x$  direction in the case of `periodic` or in periodic direction(s) in the case of `mixed` boundary conditions. Since an elongated computational domain is often desired when `grav_boundary_type` is `mixed`, the cubic grid would lead to a huge field of data. In this case, the amount of necessary grid points is reduced by using an analytical estimate to the Ewald field sufficiently far away of the symmetry plane or axis.

The old implementation (from Flash4.2) is still present and is enabled by adding `bhtreeEwaldV42=1` on the setup command line. The Ewald field is then stored in a nested set of grids, the first of them corresponds in size to full computational domain, and each following grid is half the size (in each direction) of the previous grid. Number of nested grids is controlled by runtime parameter `grv_bhEwaldNRefV42`. If `grv_bhEwaldNRefV42` is too low to cover origin (where is the Ewald field discontinuous), then the run is terminated. Each grid is composed of `grv_bhEwaldFieldNxV42`  $\times$  `grv_bhEwaldFieldNyV42`  $\times$  `grv_bhEwaldFieldNzV42` points. When evaluation of the Ewald Field at particular point is needed at any time during a run, the field value is found by interpolation in a suitable level of the grid. Linear or semi-quadratic interpolation can be chosen by runtime parameter `grv_bhLinearInterpolOnlyV42` (option `true` corresponds to linear interpolation). Semi-quadratic interpolation is recommended only in the case when there are periodic boundary conditions in two directions.

The external gravitational field can be switched on by setting `grv_useExternalPotential` true. The parameter `grv_bhExtrnPotFile` gives the name of the file with the external potential and `grv_bhExtrnPotType` specifies the field symmetry: `spherical` for the spherical symmetry and `planez` for the planar symmetry with field being a function of the  $z$ -coordinate. Parameters `grv_bhExtrnPotCenterY`, `grv_bhExtrnPotCenterX` and `grv_bhExtrnPotCenterZ` specify the position (in the simulation coordinate system) of the external field origin (the point where the radial or  $z$ -coordinate is zero).

Table 20.1: Tree gravity unit parameters controlling the accuracy of calculation.

Variable	Type	Default	Description
<code>grv_bhNewton</code>	real	-1.0	constant of gravity; if $< 0$ , it is obtained from internal physical constants database
<code>grv_bhMAC</code>	string	"ApproxPartialErr"	MAC, other option: "MaxPartialErr"
<code>grv_bhMPDegree</code>	integer	2	degree of multipole in error estimate in APE MAC
<code>grv_bhUseRelAccErr</code>	logical	.false.	if .true., <code>grv_bhAccErr</code> has meaning of relative error, otherwise absolute
<code>grv_bhAccErr</code>	real	0.1	maximum allowed error in gravitational acceleration

Tree gravity unit parameters controlling the external gravitational field.



Table 20.2: Tree gravity unit parameters controlling the boundary conditions.

Variable	Type	Default	Description
<code>grav_boundary_type</code>	string	"isolated"	or "periodic" or "mixed"
<code>grav_boundary_type_x</code>	string	"isolated"	or "periodic"
<code>grav_boundary_type_y</code>	string	"isolated"	or "periodic"
<code>grav_boundary_type_z</code>	string	"isolated"	or "periodic"
<code>grv_bhEwaldAlwaysGenerate</code>	boolean	true	whether Ewald field should be regenerated
<code>grv_bhEwaldSeriesN</code>	integer	10	number of terms in the Ewald series
<code>grv_bhEwaldNPer</code>	integer	32	number of points+1 of the Taylor expansion
<code>grv_bhEwaldFName</code>	string	"ewald_coeffs"	file with coefficients of the Ewald field Taylor expansion
<code>grv_bhEwaldFieldNxV42</code>	integer	32	size of the Ewald field grid in x-direction
<code>grv_bhEwaldFieldNyV42</code>	integer	32	size of the Ewald field grid in y-direction
<code>grv_bhEwaldFieldNzV42</code>	integer	32	size of the Ewald field grid in z-direction
<code>grv_bhEwaldNRefV42</code>	integer	-1	number of refinement levels (nested grids) for the Ewald field; if < 0, determined automatically
<code>grv_bhLinearInterpolOnlyV42</code>	logical	.true.	if .false., semi-quadratic interpolation is used for interpolation in the Ewald field
<code>grv_bhEwaldFNameAccV42</code>	string	"ewald_field_acc"	file with the Ewald field for acceleration
<code>grv_bhEwaldFNamePotV42</code>	string	"ewald_field_pot"	file with coefficients of the Ewald field for potential

Variable	Type	Default	Description
<code>grv_bhUseExternalPotential</code>	logical	.false.	whether to use external field
<code>grv_bhUsePoissonPotential</code>	logical	.true.	whether to use gravitational field calculated by the tree solver
<code>grv_bhExtrnPotFile</code>	string	"external_potential.dat"	file containing the external gravitational field
<code>grv_bhExtrnPotType</code>	string	"planez"	type of the external field: planar or spherical symmetry
<code>grv_bhExtrnPotCenterX</code>	real	0.0	x-coordinate of the center of the external field
<code>grv_bhExtrnPotCenterY</code>	real	0.0	y-coordinate of the center of the external field
<code>grv_bhExtrnPotCenterZ</code>	real	0.0	z-coordinate of the center of the external field

## 20.5 Unit Tests

There are two unit tests for the gravity unit. `Poisson3` is essentially the Maclaurin spheroid problem described in [Section 35.3.4](#). Because an analytical solution exists, the accuracy of the gravitational solver can be quantified. The second test, `Poisson3_active` is a modification of `Poisson3` to test the mapping of particles in `Grid_mapParticlesToMesh`. Some of the mesh density is redistributed onto particles, and the particles are then mapped back to the mesh, using the analytical solution to verify completeness. This test is similar to the simulation `PoisParticles` discussed in [Section 35.4.3](#). `PoisParticles` is based on the Huang-Greengard Poisson gravity test described in [Section 35.3.3](#).



## Chapter 21

# Particles Unit

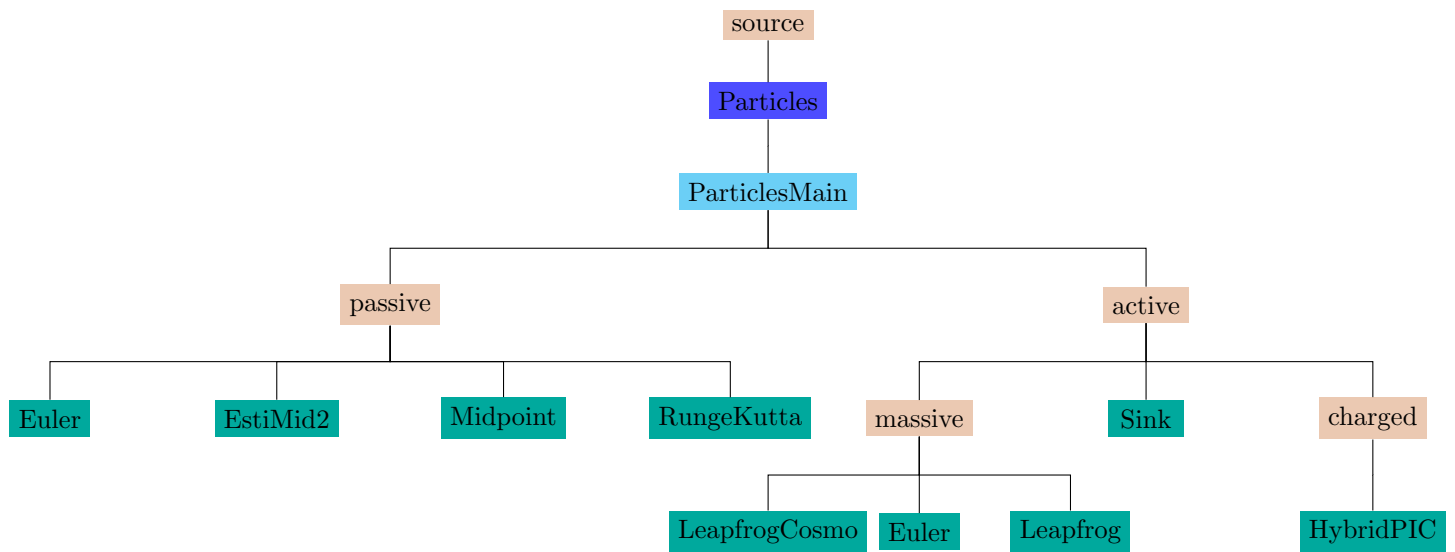


Figure 21.1: The `Particles` unit main subunit.

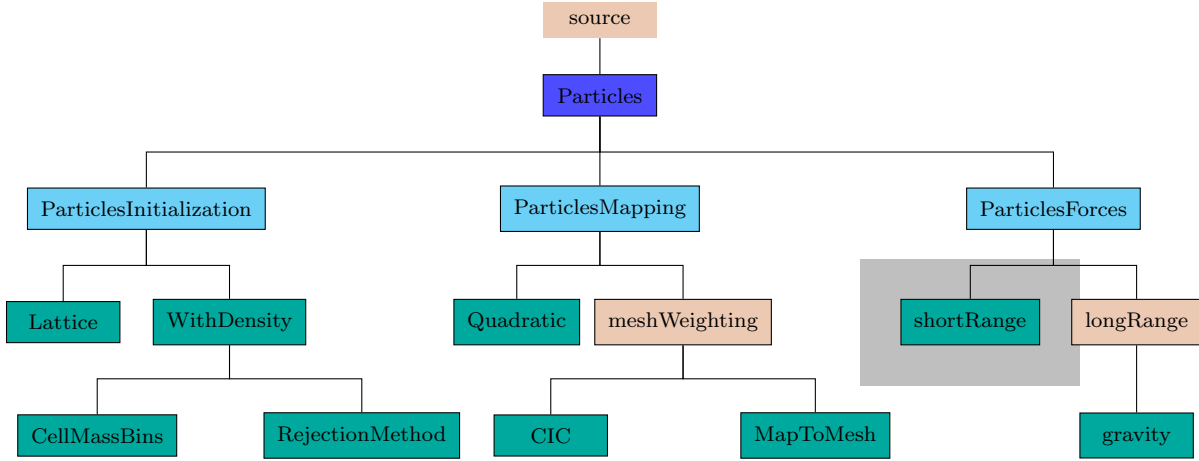


Figure 21.2: The `Particles` unit with `ParticlesInitialization` and `ParticlesMapping` subunits.

The support for particles in FLASH4 comes in two flavors, *active* and *passive*. Active particles are further classified into two categories; *massive* and *charged*. The active particles contribute to the dynamics of the simulation, while passive particles follow the motion of Lagrangian tracers and make no contribution to the dynamics. Particles are dimensionless objects characterized by positions  $\mathbf{x}_i$ , 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.

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i \quad (21.1)$$

Active particles experience forces and may themselves contribute to the problem dynamics (*e.g.*, through long-range forces or through collisions). They may additionally have their own motion independent of the grid, so an additional motion equation of

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n. \quad (21.2)$$

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 (21.1) and another describing the effects of forces.

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{F}_{\text{lr},i} + \mathbf{F}_{\text{sr},i}, \quad (21.3)$$

Here,  $\mathbf{F}_{\text{lr},i}$  represents the sum of all long-range forces (coupling all particles, except possibly those handled by the short-range term) acting on the  $i$ th particle and  $\mathbf{F}_{\text{sr},i}$  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 (21.1) forward through time. Many alternative integration methods are described in Section 21.1 below. Additional information about the mesh to particle mapping is described in Section 21.2. 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).

### FLASH Transition

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

### New since FLASH3.1

Since release 3.1 of FLASH, a single simulation can have both active and passive particles defined. FLASH3 and FLASH2 allowed only active *or* passive particles in a simulation. Because of the added complexity, new **Config** syntax and new **setup** script syntax is necessary for Particles. See [Section 5.2](#) for command line options, [Section 6.6](#) for **Config** syntax, and [Section 21.3](#) below for more details.

FLASH4 includes support for **sink particles**. These are a special kind of (massive) active particles, with special rules for creation, mass accretion, and interaction with fluid variables and other particles. See [Section 21.4](#) below for information specific to sink particles.

## 21.1 Time Integration

The active and passive particles have many different time integration schemes available. The subroutine **Particles.advance** handles the movement of particles through space and time. Because FLASH4 has support for including different types of both active and passive particles in a single simulation, the implementation of **Particles.advance** may call several helper routines of the form **pt\_advanceMETHOD** (e.g., **pt\_advanceLeapfrog**, **pt\_advanceEuler\_active**, **pt\_advanceCustom**), each acting on an appropriate subset of existing particles. The *METHOD* here is determined by the **ADVMETHOD** part of the **PARTICLETYPE** **Config** statement (or the **ADV** par of a **-particlemethods setup** option) for the type of particle. See the **Particles.advance** source code for the mapping from **ADVMETHOD** keyword to **pt\_advanceMETHOD** subroutine call.

### 21.1.1 Active Particles (Massive)

The **active** particles implementation includes different time integration schemes, long-range force laws (coupling all particles), and short-range force laws (coupling nearby particles). The attributes listed in [Table 21.1](#) are provided by this subunit. A specific implementation of the active portion of **Particles.advance** is selected by a setup option such as **-with-unit=Particles/ParticlesMain/active/massive/Leapfrog**, or by specifying something like **REQUIRES Particles/ParticlesMain/active/massive/Leapfrog** in a simulation's **Config** file (or by listing the path in the **Units** file if not using the **-auto** configuration option). Further details are given in [Section 21.3](#) below.

Available time integration schemes for active particles include

- **Forward Euler.** Particles are advanced in time from  $t^n$  to  $t^{n+1} = t^n + \Delta t^n$  using the following difference equations:

$$\begin{aligned} \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \\ \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n . \end{aligned} \quad (21.4)$$

Here  $\mathbf{a}_i$  is the particle acceleration. Within FLASH4, this scheme is implemented in **Particles/-ParticlesMain/active/massive/Euler**. This Euler scheme (as well as the Euler scheme for the passive particles) is first-order accurate and is included for testing purposes only. It should not be used in a production run.

- **Variable-timestep leapfrog.** Particles are advanced using the following difference equations

$$\begin{aligned}
\mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\
\mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \frac{1}{2} \mathbf{a}_i^0 \Delta t^0 \\
\mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2} + C_n \mathbf{a}_i^n + D_n \mathbf{a}_i^{n-1} \\
\mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \Delta t^n .
\end{aligned} \tag{21.5}$$

The coefficients  $C_n$  and  $D_n$  are given by

$$\begin{aligned}
C_n &= \frac{1}{2} \Delta t^n + \frac{1}{3} \Delta t^{n-1} + \frac{1}{6} \left( \frac{\Delta t^{n2}}{\Delta t^{n-1}} \right) \\
D_n &= \frac{1}{6} \left( \Delta t^{n-1} - \frac{\Delta t^{n2}}{\Delta t^{n-1}} \right) .
\end{aligned} \tag{21.6}$$

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

- **Cosmological variable-timestep leapfrog.** (`Particles/ParticlesMain/active/massive/LeapfrogCosmo`)

The coefficients in the leapfrog update are modified to take into account the effect of cosmological redshift on the particles. The particle positions  $\mathbf{x}$  are interpreted as comoving positions, and the particle velocities  $\mathbf{v}$  are interpreted as comoving peculiar velocities ( $\mathbf{v} = \dot{\mathbf{x}}$ ). The resulting update steps are

$$\begin{aligned}
\mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\
\mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \frac{1}{2} \mathbf{a}_i^0 \Delta t^0 \\
\mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2} \left[ 1 - \frac{A^n}{2} \Delta t^n + \frac{1}{3!} \Delta t^{n2} (A^{n2} - \dot{A}^n) \right] \left[ 1 - \Delta t^{n-1} \frac{A^n}{2} + \Delta t^{n-12} \frac{A^{n2} + 2\dot{A}^n}{12} \right] \\
&\quad + \mathbf{a}_i^n \left[ \frac{\Delta t^{n-1}}{2} + \frac{\Delta t^{n2}}{6\Delta t^{n-1}} + \frac{\Delta t^{n-1}}{3} - \frac{A^n \Delta t^n}{6} (\Delta t^n + \Delta t^{n-1}) \right] \\
&\quad + \mathbf{a}_i^{n-1} \left[ \frac{\Delta t^{n-12} - \Delta t^{n2}}{6\Delta t^{n-1}} - \frac{A^n \Delta t^{n-1}}{12} (\Delta t^n + \Delta t^{n-1}) \right] \\
\mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \Delta t^n .
\end{aligned}$$

Here we define  $A \equiv -2\dot{a}/a$ , where  $a$  is the scale factor. Note that the acceleration  $\mathbf{a}_i^{n-1}$  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  $\dot{a} \equiv 0$ .

- **Sink particles** have their own implementation of several advancement methods (with time subcycling), implemented under `Particles/ParticlesMain/active/Sink`, see description below in [Section 21.4](#).

The leapfrog-based integrators implemented under `Particles/ParticlesMain/active/massive` supply the additional particle attributes listed in [Table 21.2](#).

Table 21.1: Particle attributes provided by active particles.

Attribute	Type	Description
MASS_PART_PROP	REAL	Particle mass
ACCX_PART_PROP	REAL	$x$ -component of particle acceleration
ACCY_PART_PROP	REAL	$y$ -component of particle acceleration
ACCZ_PART_PROP	REAL	$z$ -component of particle acceleration

Table 21.2: Particle attributes provided by leapfrog time integration.

Attribute	Type	Description
OACX_PART_PROP	REAL	$x$ -component of particle acceleration at previous timestep
OACY_PART_PROP	REAL	$y$ -component of particle acceleration at previous timestep
OACZ_PART_PROP	REAL	$z$ -component of particle acceleration at previous timestep

### 21.1.2 Charged Particles - Hybrid PIC

Collisionless plasmas are often modeled using fluid magnetohydrodynamics (MHD) models. However, the MHD fluid approximation is questionable when the gyroradius of the ions is large compared to the spatial region that is studied. On the other hand, kinetic models that discretize the full velocity space, or full particle in cell (PIC) models that treat ions and electrons as particles, are computationally very expensive. For problems where the time scales and spatial scales of the ions are of interest, hybrid models provide a compromise. In such models, the ions are treated as discrete particles, while the electrons are treated as a (often massless) fluid. This means that the time scales and spatial scales of the electrons do not need to be resolved, and enables applications such as modeling of the solar wind interaction with planets. For a detailed discussion of different plasma models, see Ledvina et al. (2008).

#### 21.1.2.1 The hybrid equations

In the hybrid approximation, ions are treated as particles, and electrons as a massless fluid. In what follows we use SI units. We have  $N_I$  ions at positions  $\mathbf{r}_i(t)$  [m] with velocities  $\mathbf{v}_i(t)$  [m/s], mass  $m_i$  [kg] and charge  $q_i$  [C],  $i = 1, \dots, N_I$ . By spatial averaging we can define the charge density  $\rho_I(\mathbf{r}, t)$  [Cm<sup>-3</sup>] of the ions, their average velocity  $\mathbf{u}_I(\mathbf{r}, t)$  [m/s], and the corresponding current density  $\mathbf{J}_I(\mathbf{r}, t) = \rho_I \mathbf{u}_I$  [Cm<sup>-2</sup>s<sup>-1</sup>]. Electrons are modelled as a fluid with charge density  $\rho_e(\mathbf{r}, t)$ , average velocity  $\mathbf{u}_e(\mathbf{r}, t)$ , and current density  $\mathbf{J}_e(\mathbf{r}, t) = \rho_e \mathbf{u}_e$ . The electron number density is  $n_e = -\rho_e/e$ , where  $e$  is the elementary charge. If we assume that the electrons are an ideal gas, then  $p_e = n_e k T_e$ , so the pressure is directly related to temperature ( $k$  is Boltzmann's constant).

The trajectories of the ions are computed from the Lorentz force,

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i, \quad \frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i} (\mathbf{E} + \mathbf{v}_i \times \mathbf{B}), \quad i = 1, \dots, N_I$$

where  $\mathbf{E} = \mathbf{E}(\mathbf{r}, t)$  is the electric field, and  $\mathbf{B} = \mathbf{B}(\mathbf{r}, t)$  is the magnetic field. The electric field is given by

$$\mathbf{E} = \frac{1}{\rho_I} (-\mathbf{J}_I \times \mathbf{B} + \mu_0^{-1} (\nabla \times \mathbf{B}) \times \mathbf{B} - \nabla p_e) + \eta \mathbf{J} - \eta_h \nabla^2 \mathbf{J},$$

where  $\rho_I$  is the ion charge density,  $\mathbf{J}_I$  is the ion current,  $p_e$  is the electron pressure,  $\mu_0 = 4\pi \cdot 10^{-7}$  is the magnetic constant,  $\mathbf{J} = \mu_0^{-1} \nabla \times \mathbf{B}$  is the current, and  $\eta_h$  is a hyperresistivity. Here we assume that  $p_e$  is adiabatic. Then the relative change in electron pressure is related to the relative change in electron density by

$$\frac{p_e}{p_{e0}} = \left( \frac{n_e}{n_{e0}} \right)^\gamma,$$

where the zero subscript denote reference values (here the initial values at  $t = 0$ ). Then Faraday's law is used to advance the magnetic field in time,

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}.$$

### 21.1.2.2 A cell-centered finite difference hybrid PIC solver

We use a cell-centered representation of the magnetic field on a uniform grid. All spatial derivatives are discretized using standard second order finite difference stencils. Time advancement is done by a predictor-corrector leapfrog method with subcycling of the field update, denoted cyclic leapfrog (CL) by Matthew (1994). An advantage of the discretization is that the divergence of the magnetic field is zero, down to round off errors. The ion macroparticles (each representing a large number of real particles) are deposited on the grid by a cloud-in-cell method (linear weighting), and interpolation of the fields to the particle positions are done by the corresponding linear interpolation. Initial particle positions are drawn from a uniform distribution, and initial particle velocities from a Maxwellian distribution. Further details of the algorithm can be found in Holmström, M. (2012,2013) and references therein, where an extension of the solver that include inflow and outflow boundary conditions was used to model the interaction between the solar wind and the Moon. In what follows we describe the FLASH implementation of the hybrid solver with periodic boundary conditions.

### 21.1.2.3 Hybrid solver implementation

The two basic operations needed for a PIC code are provided as standard operations in FLASH:

- Deposit charges and currents onto the grid: `call Grid_mapParticlesToMesh()`
- Interpolate fields to particle positions: `call Grid_mapMeshToParticles()`

At present the solver is restricted to a Cartesian uniform grid, since running an electromagnetic particle code on an adaptive grid is not straightforward. Grid refinement/coarsening must be accompanied by particle split/join operations. Also, jumps in the cell size can lead to reflected electromagnetic waves.

The equations are stated in SI units, so all parameters should be given in SI units, and all output will be in SI units. The initial configuration is a spatial uniform plasma consisting of two species. The first species, named `pt_picPname_1` consists of particles of mass `pt_picPmass_1` and charge `pt_picPcharge_1`. The initial (at  $t = 0$ ) uniform number density is `pt_picPdensity_1`, and the velocity distribution is Maxwellian with a drift velocity of (`pt_picPvelx_1`, `pt_picPvely_1`, `pt_picPvelz_1`), and a temperature of `pt_picPtemp_1`. Each model macro-particle represents many real particles. The number of macro-particles per grid cell at the start of the simulation is set by `pt_picPpc_1`. So this parameter will control the total number of macro-particles of species 1 in the simulation.

All the above parameters are also available for a second species, e.g., `pt_picPmass_2`, which is initialized in the same way as the first species. The grid is initialized with the uniform magnetic field (`sim_bx`, `sim_by`, `sim_bz`).

Now for grid quantities. The cell averaged mass density is stored in `pden`, and the charge density in `cdens`. The magnetic field is represented by (`grbx`, `grby`, `grbz`). A background field can be set during initialization in (`gbx1`, `gby1`, `gbz1`). We then solve for the deviation from this background field. Care must be taken so that the background field is divergence free, using the discrete divergence operator. The easiest way to ensure this is to compute the background field as the rotation of a potential. The electric field is stored in (`grex`, `grey`, `grez`), the current density in (`grjx`, `grjy`, `grjz`), and the ion current density in (`gjix`, `gjy`, `gjiz`). A resistivity is stored in `gres`, thus it is possible to have a non-uniform resistivity in the simulation domain, but the default is to set the resistivity to the constant value of `sim_resistivity` everywhere. For post processing, separate fields are stored for charge density and ion current density for species 1 and 2.

Regarding particle properties. Each computational meta-particles is labeled by a positive integer, `specie` and has a `mass` and `charge`. As all FLASH particles they also each have a position  $\mathbf{r}_i = (\text{posx}, \text{posy}, \text{posz})$  and a velocity  $\mathbf{v}_i = (\text{velx}, \text{vely}, \text{velz})$ . To be able to deposit currents onto the grid, each particle stores the current density corresponding to the particle,  $\mathbf{J}_{Ii}$ . For the movement of the particles by the Lorentz force, we also need the electric and magnetic fields at the particle positions,  $\mathbf{E}(\mathbf{r}_i)$  and  $\mathbf{B}(\mathbf{r}_i)$ , respectively.

Regarding the choice of time step. The timestep must be constant,  $\Delta t = \text{dtinit} = \text{dtmin} = \text{dtmax}$ , since the leap frog solver requires this. For the solution to be stable the time step,  $\Delta t$ , must be small enough. We will try and quantify this, and here we assume that the grid cells are cubes,  $\Delta x = \Delta y = \Delta z$ , and that we have a single species plasma.



Table 21.3: Runtime parameters for the hybrid solver. Initial values are at  $t = 0$ . For each parameter for species 1, there is a corresponding parameter for species 2 (named with 2 instead of 1), e.g., `pt_picPvelx_2`.

Variable	Type	Default	Description
<code>pt_picPname_1</code>	STRING	"H+"	Species 1 name
<code>pt_picPmass_1</code>	REAL	1.0	Species 1 mass, $m_i$ [amu]
<code>pt_picPcharge_1</code>	REAL	1.0	Species 1 charge, $q_i$ [e]
<code>pt_picPdensity_1</code>	REAL	1.0	Initial $n_I$ species 1 [ $\text{m}^{-3}$ ]
<code>pt_picPtemp_1</code>	REAL	1.5e5	Initial $T_I$ species 1 [K]
<code>pt_picPvelx_1</code>	REAL	0.0	Initial $\mathbf{u}_I$ species 1 [m/s]
<code>pt_picPvely_1</code>	REAL	0.0	
<code>pt_picPvelz_1</code>	REAL	0.0	
<code>pt_picPpc_1</code>	REAL	1.0	Number of macro-particle of species 1 per cell
<code>sim_bx</code>	REAL	0.0	Initial $\mathbf{B}$ (at $t = 0$ ) [T]
<code>sim_by</code>	REAL	0.0	
<code>sim_bz</code>	REAL	0.0	
<code>sim_te</code>	REAL	0.0	Initial $T_e$ [K]
<code>pt_picResistivity</code>	REAL	0.0	Resistivity, $\eta$ [ $\Omega \text{ m}$ ]
<code>pt_picResistivityHyper</code>	REAL	0.0	Hyperresistivity, $\eta_h$
<code>sim_gam</code>	REAL	-1.0	Adiabatic exponent for electrons
<code>sim_nsub</code>	INTEGER	3	Number of CL B-field update subcycles (must be odd)
<code>sim_rng_seed</code>	INTEGER	0	Seed the random number generator (if $> 0$ )

First of all, since the time advancement is explicit, there is the standard CFL condition that no particle should travel more than one grid cell in one time step, i.e.  $\Delta t \max_i(|\mathbf{v}_i|) < \Delta x$ . This time step is printed to standard output by FLASH (as `dt_Part`), and can thus be monitored.

Secondly, we also have an upper limit on the time step due to Whistler waves (Pritchett 2000),

$$\Delta t < \frac{\Omega_i^{-1}}{\pi} \left( \frac{\Delta x}{\delta_i} \right)^2 \sim \frac{n}{B} (\Delta x)^2, \quad \delta_i = \frac{1}{|q_i|} \sqrt{\frac{m_i}{\mu_0 n}},$$

where  $\delta_i$  is the ion inertial length, and  $\Omega_i = |q_i|B/m_i$  is the ion gyrofrequency.

Finally, we also want to resolve the ion gyro motion by having several time steps per gyration. This will only put a limit on the time step if, approximately,  $\Delta x > 5\delta_i$ , and we then have that  $\Delta t < \Omega_i^{-1}$ .

All this are only estimates that does not take into account, *e.g.*, the initial functions, or the subcycling of the magnetic field update. In practice one can just reduce the time step until the solution is stable. Then for runs with different density and/or magnetic field strength, the time step will need to be scaled by the change in  $n_I/B$ , *e.g.*, if  $n_I$  is doubled,  $\Delta t$  can be doubled. The factor  $(\Delta x)^2$  implies that reducing the cell size by a factor of two will require a reduction in the time step by a factor of four.

### 21.1.3 Passive Particles

Passive particles may be moved using one of several different methods available in FLASH4. With the exception of **Midpoint**, they are all single-step schemes. The methods are either first-order or second-order accurate, and all are explicit, as described below. In all implementations, particle velocities are obtained by mapping grid-based velocities onto particle positions as described in [Section 21.2](#).

Numerically solving Equation (21.1) for passive particles means solving a set of simple ODE initial value problems, separately for each particle, where the velocities  $\mathbf{v}_i$  are given at certain discrete points in time by the state of the hydrodynamic velocity field at those times. The time step is thus externally given and

Table 21.4: The grid variables for the hybrid solver that are most important for a user. For each property for species 1, there is a corresponding variable for species 2 (named with 2 instead of 1), e.g., `cde2`.

Variable	Type	Description
<code>cden</code>	PER.VOLUME	Total charge density, $\rho$ [C/m <sup>3</sup> ]
<code>grbx</code>	GENERIC	Magnetic field, $\mathbf{B}$ [T]
<code>grby</code>	GENERIC	
<code>grbz</code>	GENERIC	Electric field, $\mathbf{E}$ [V/m]
<code>grex</code>	GENERIC	
<code>grey</code>	GENERIC	
<code>grez</code>	GENERIC	Current density, $\mathbf{J}$ [A/m <sup>2</sup> ]
<code>grjx</code>	PER.VOLUME	
<code>grjy</code>	PER.VOLUME	
<code>grjz</code>	PER.VOLUME	
<code>gjix</code>	PER.VOLUME	Ion current density, $\mathbf{J}_I$ [A/m <sup>2</sup> ]
<code>gjiy</code>	PER.VOLUME	
<code>gjiz</code>	PER.VOLUME	
<code>cde1</code>	PER.VOLUME	Species 1 charge density, $\rho_I$ [C/m <sup>3</sup> ]
<code>jix1</code>	PER.VOLUME	Species 1 ion current density, $\mathbf{J}_I$ [A/m <sup>2</sup> ]
<code>jiy1</code>	PER.VOLUME	
<code>jiz1</code>	PER.VOLUME	

cannot be arbitrarily chosen by a particle motion ODE solver<sup>1</sup>. Statements about the order of a method in this context should be understood as referring to the same method if it were applied in a hypothetical simulation where evaluations of velocities  $\mathbf{v}_i$  could be performed at arbitrary times (and with unlimited accuracy). Note that FLASH4 does not attempt to provide a particle motion ODE solver of higher accuracy than second order, since it makes little sense to integrate particle motion to a higher order than the fluid dynamics that provide its inputs.

In all cases, particles are advanced in time from  $t^n$  (or, in the case of `Midpoint`, from  $t^{n-1}$ ) to  $t^{n+1} = t^n + \Delta t^n$  using one of the difference equations described below. The implementations assume that at the time when `Particles.advance` is called, the fluid fields have already been updated to  $t^{n+1}$ , as is the case with the `Driver.evolveFlash` implementations provided with FLASH4. A specific implementation of the passive portion of `Particles.advance` is selected by a setup option such as `-with-unit=Particles/ParticlesMain/passive/Euler`, or by specifying something like `REQUIRES Particles/ParticlesMain/passive/Euler` in a simulation's `Config` file (or by listing the path in the `Units` file if not using the `-auto` configuration option). Further details are given in [Section 21.3](#) below.

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

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \quad . \quad (21.7)$$

Here  $\mathbf{v}_i^n$  is the velocity of the particle, which is obtained using particle-mesh interpolation from the grid at  $t = t^n$ .

Note that this evaluation of  $\mathbf{v}_i^n$  cannot be deferred until the time when it is needed at  $t = t^{n+1}$ , since at that point the fluid variables have been updated and the velocity fields at  $t = t^n$  are not available any more. Particle velocities are therefore interpolated from the mesh at  $t = t^n$  and stored as particle attributes. Similar concerns apply to the remaining methods but will not be explicitly mentioned every time.

<sup>1</sup>Even though it is possible to do so, see `Particles.computeDt`, one does not in general wish to let particles integration dictate the time step of the simulation.

Table 21.5: Important particle properties for the hybrid solver. Note that this is for the computational macro-particles.

Variable	Type	Description
<code>specie</code>	REAL	Particle type (an integer number 1,2,3,...)
<code>mass</code>	REAL	Mass of the particle, $m_i$ [kg]
<code>charge</code>	REAL	Charge of the particle, $q_i$ [C]
<code>jx</code>	REAL	Particle ion current, $\mathbf{J}_{Ii} = q_i \mathbf{v}_i$ [A m]
<code>jy</code>	REAL	
<code>jz</code>	REAL	
<code>bx</code>	REAL	Magnetic field at particle, $\mathbf{B}(\mathbf{r}_i)$ [T]
<code>by</code>	REAL	
<code>bz</code>	REAL	
<code>ex</code>	REAL	Electric field at particle, $\mathbf{E}(\mathbf{r}_i)$ [V/m]
<code>ey</code>	REAL	
<code>ez</code>	REAL	

- **Two-Stage Runge-Kutta (Particles/ParticlesMain/passive/RungeKutta).** This 2-stage Runge-Kutta scheme is the preferred choice in FLASH4. It is also the default which is compiled in if particles are included in the setup but no specific alternative implementation is requested. The scheme is also known as Heun’s Method:

$$\begin{aligned} \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \frac{\Delta t^n}{2} [\mathbf{v}_i^n + \mathbf{v}_i^{*,n+1}] , \\ \text{where } \mathbf{v}_i^{*,n+1} &= \mathbf{v}(\mathbf{x}_i^{*,n+1}, t^{n+1}) , \\ \mathbf{x}_i^{*,n+1} &= \mathbf{x}_i^n + \Delta t^n \mathbf{v}_i^n . \end{aligned} \quad (21.8)$$

Here  $\mathbf{v}(\mathbf{x}, t)$  denotes evaluation (interpolation) of the fluid velocity field at position  $\mathbf{x}$  and time  $t$ ;  $\mathbf{v}_i^{*,n+1}$  and  $\mathbf{x}_i^{*,n+1}$  are intermediate results <sup>2</sup>; and  $\mathbf{v}_i^n = \mathbf{v}(\mathbf{x}_i^n, t^n)$  is the velocity of the particle, obtained using particle-mesh interpolation from the grid at  $t = t^n$  as usual.

- **Midpoint (Particles/ParticlesMain/passive/Midpoint).** This Midpoint scheme is a two-step scheme. Here, the particles are advanced from time  $t^{n-1}$  to  $t^{n+1} = t^{n-1} + \Delta t^{n-1} + \Delta t^n$  by the equation

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^{n-1} + \mathbf{v}_i^n (\Delta t^{n-1} + \Delta t^n) . \quad (21.9)$$

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

To get the scheme started, an Euler step (as described for `passive/Euler`) is taken the first time `Particles/ParticlesMain/passive/Midpoint/pt_advancePassive` is called.

The Midpoint alternative implementation uses the following additional particle attributes:

```
PARTICLEPROP pos2PrevX REAL      # two previous x-coordinate
PARTICLEPROP pos2PrevY REAL      # two previous y-coordinate
PARTICLEPROP pos2PrevZ REAL      # two previous z-coordinate
```

- **Estimated Midpoint with Correction (Particles/ParticlesMain/passive/EstiMidpoint2).** The scheme is second order even if  $\Delta t^n = \Delta t^{n+1}$  is not assumed. It is essentially the `EstiMidpoint` or “Predictor-Corrector” method of previous releases, with a correction for non-constant time steps by using additional evaluations (at times and positions that are easily available, without requiring more particle attributes).

---

<sup>2</sup>They can be considered “predicted” positions and velocities.

Particle advancement follows the equation

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t^n \mathbf{v}_i^{\text{comb}}, \quad (21.10)$$

where

$$\mathbf{v}_i^{\text{comb}} = c_1 \mathbf{v}(\mathbf{x}_i^{*,n+\frac{1}{2}}, t^n) + c_2 \mathbf{v}(\mathbf{x}_i^{*,n+\frac{1}{2}}, t^{n+1}) + c_3 \mathbf{v}(\mathbf{x}_i^n, t^n) + c_4 \mathbf{v}(\mathbf{x}_i^n, t^{n+1}) \quad (21.11)$$

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

$$\mathbf{x}_i^{*,n+\frac{1}{2}} = \mathbf{x}_i^n + \frac{1}{2} \Delta t^{n-1} \mathbf{v}_i^n$$

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

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

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  $t_*^{n+\frac{1}{2}} = t^n + \Delta t_*^n$  the estimated midpoint time, and  $t^{n+\frac{1}{2}} = t^n + \frac{1}{2} \Delta t^n$  the actual midpoint of the  $[t^n, t^{n+1}]$  interval. Also write  $\mathbf{x}_i^{\text{E},n+\frac{1}{2}} = \mathbf{x}_i^n + \frac{1}{2} \Delta t^n \mathbf{v}_i^n$  for first-order (Euler) approximate positions at the actual midpoint time  $t^{n+\frac{1}{2}}$ , and we continue to denote with  $\mathbf{x}_i^{*,n+\frac{1}{2}}$  the estimated positions reached at the estimated midpoint time  $t_*^{n+\frac{1}{2}}$ .

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

$$\mathbf{v}(\mathbf{x}_i^{\text{E},n+\frac{1}{2}}, t^{n+\frac{1}{2}}) = \mathbf{v}(\mathbf{x}_i^n, t^n) + \mathbf{v}_t(\mathbf{x}_i^n, t^n) \frac{1}{2} \Delta t^n + (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) \frac{1}{2} \Delta t^n + O((\frac{1}{2} \Delta t^n)^2) \quad (21.12)$$

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

On the other hand, expansion of (21.11) gives

$$\begin{aligned} \mathbf{v}_i^{\text{comb}} &= (c_1 + c_2 + c_3 + c_4) \mathbf{v}(\mathbf{x}_i^n, t^n) \\ &+ (c_2 + c_4) \mathbf{v}_t(\mathbf{x}_i^n, t^n) \Delta t + (c_1 + c_2) (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) \Delta t_*^n \\ &+ \text{higher order terms in } \Delta t \text{ and } \Delta t_*^n. \end{aligned}$$

After introducing a time step factor  $f$  defined by  $\Delta t_*^n = f \Delta t^n$ , this becomes

$$\begin{aligned} \mathbf{v}_i^{\text{comb}} &= (c_1 + c_2 + c_3 + c_4) \mathbf{v}(\mathbf{x}_i^n, t^n) \\ &+ (c_2 + c_4) \mathbf{v}_t(\mathbf{x}_i^n, t^n) \Delta t + (c_1 + c_2) (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}}) \mathbf{v}(\mathbf{x}_i^n, t^n) f \Delta t \\ &+ O((\Delta t)^2) \quad . \end{aligned} \quad (21.13)$$

One can derive conditions for second order accuracy by comparing (21.13) with (21.12) and requiring that

$$\mathbf{v}_i^{\text{comb}} = \mathbf{v}(\mathbf{x}_i^{\text{E},n+\frac{1}{2}}, t^{n+\frac{1}{2}}) + O((\Delta t)^2) \quad . \quad (21.14)$$

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:

$$\begin{aligned} c_1 + c_2 + c_3 + c_4 &= 1 \quad (\text{otherwise the scheme will not even be of first order}) , \\ c_2 + c_4 &= \frac{1}{2} \quad (\text{and thus also } c_1 + c_3 = \frac{1}{2}) , \\ c_1 + c_2 &= \frac{\Delta t^n}{\Delta t^{n-1}} . \end{aligned}$$

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:

$$\begin{aligned} c_1 &= \frac{\Delta t^n}{\Delta t^{n-1}} - \frac{1}{2} , \\ c_2 &= \frac{1}{2} , \\ c_3 &= 1 - \frac{\Delta t^n}{\Delta t^{n-1}} , \\ c_4 &= 0 . \end{aligned}$$

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 (21.10) simplifies significantly.

An Euler step, as described for `passive/Euler` in (21.7), is taken the first time when `Particles/ParticlesMain/passive/EstiMidpoint2/pt_advancePassive` is called and when the time step has changed too much. Since the integration scheme is tolerant of time step changes, it should usually not be necessary to apply the second criterion; even when it is to be employed, the criteria should be less strict than for an uncorrected `EstiMidpoint` scheme. For `EstiMidPoint2` the timestep is considered to have changed too much if either of the following is true:

$$\Delta t^n > \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq \text{pt\_dtChangeToleranceUp} \times \Delta t^{n-1}$$

or

$$\Delta t^n < \Delta t^{n-1} \quad \text{and} \quad |\Delta t^n - \Delta t^{n-1}| \geq \text{pt\_dtChangeToleranceDown} \times \Delta t^{n-1},$$

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  $\mathbf{x}_i^{*,n+\frac{1}{2}}$  and  $\mathbf{v}_i^{*,n+\frac{1}{2}}$  between the `Particles_advance` calls at  $t^n$  and  $t^{n+1}$ :

```
PARTICLEPROP velPredX REAL
PARTICLEPROP velPredY REAL
PARTICLEPROP velPredZ REAL
PARTICLEPROP posPredX REAL
PARTICLEPROP posPredY REAL
PARTICLEPROP posPredZ REAL
```

The time integration of passive particles is tested in the `ParticlesAdvance` unit test, which can be used to examine the convergence behavior, see Section 21.3.4.

## 21.2 Mesh/Particle Mapping

Particles behave in a fundamentally different way than grid-based quantities. Lagrangian, or passive particles are essentially independent of the grid mesh and move along with the velocity field. Active particles may be located independently of mesh refinement. In either case, there is a need to convert grid-based quantities into similar attributes defined on particles, or vice versa. The method for interpolating mesh quantities

to tracer particle positions must be consistent with the numerical method to avoid introducing systematic error. In the case of a finite-volume methods such as those used in FLASH4, the mesh quantities have cell-averaged rather than point values, which requires that the interpolation function for the particles also represent cell-averaged values. Cell averaged quantities are defined as

$$f_i(x) \equiv \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} f(x') dx' \quad (21.15)$$

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.

### 21.2.1 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

$$f(x) = A + B(x - x_i) + C(x - x_i)^2. \quad (21.16)$$

Then integrating gives

$$\begin{aligned} f_{i-1} &= \frac{1}{\Delta x} \left[ A + \frac{1}{2}B(x - x_i)^2 \right]_{x_{i-3/2}}^{x_{i-1/2}} + \frac{1}{3}C(x - x_i)^3 \Big|_{x_{i-3/2}}^{x_{i-1/2}} \\ &= A - B\Delta x + \frac{13}{12}C\Delta x^2, \end{aligned} \quad (21.17)$$

$$\begin{aligned} f_i &= \frac{1}{\Delta x} \left[ A + \frac{1}{2}B(x - x_i)^2 \right]_{x_{i-1/2}}^{x_{i+1/2}} + \frac{1}{3}C(x - x_i)^3 \Big|_{x_{i-1/2}}^{x_{i+1/2}} \\ &= A + \frac{1}{12}C\Delta x^2, \end{aligned} \quad (21.18)$$

and

$$\begin{aligned} f_{i+1} &= \frac{1}{\Delta x} \left[ A + \frac{1}{2}B(x - x_i)^2 \right]_{x_{i+1/2}}^{x_{i+3/2}} + \frac{1}{3}C(x - x_i)^3 \Big|_{x_{i+1/2}}^{x_{i+3/2}} \\ &= A + B\Delta x + \frac{13}{12}C\Delta x^2, \end{aligned} \quad (21.19)$$

We may write these as

$$\begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & -1 & \frac{13}{12} \\ 1 & 0 & \frac{1}{12} \\ 1 & 1 & \frac{13}{12} \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \end{bmatrix}. \quad (21.20)$$

Inverting this gives expressions for  $A$ ,  $B$ , and  $C$ ,

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & \frac{13}{12} & -\frac{1}{24} \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix}. \quad (21.21)$$

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

$$f(x, y) = A + B(x - x_i) + C(x - x_i)^2 + D(y - y_j) + E(y - y_j)^2 + F(x - x_i)(y - y_j). \quad (21.22)$$

In this case, the cell averaged quantities are given by

$$f_{i,j}(x, y) \equiv \frac{1}{\Delta y} \Delta x \int_{x_{i-1/2}}^{x_{i+1/2}} dx' \int_{y_{j-1/2}}^{y_{j+1/2}} dy' f(x', y') . \quad (21.23)$$

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

$$\begin{bmatrix} f_{i-1,j-1} \\ f_{i,j-1} \\ f_{i+1,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i-1,j+1} \\ f_{i,j+1} \\ f_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} 1 & -1 & \frac{13}{12} & -1 & \frac{13}{12} & 1 \\ 1 & 0 & \frac{13}{12} & -1 & \frac{13}{12} & 0 \\ 1 & 1 & \frac{13}{12} & -1 & \frac{13}{12} & -1 \\ 1 & -1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\ 1 & 0 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\ 1 & 1 & \frac{13}{12} & 0 & \frac{13}{12} & 0 \\ 1 & -1 & \frac{13}{12} & 1 & \frac{13}{12} & -1 \\ 1 & 0 & \frac{13}{12} & 1 & \frac{13}{12} & 0 \\ 1 & 1 & \frac{13}{12} & 1 & \frac{13}{12} & 1 \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \\ F\Delta x\Delta y \end{bmatrix} . \quad (21.24)$$

At this point we note that there are more constraints than unknowns, and we must make a choice of the constraints. We chose to ignore the cross terms and take only the face-centered cells next to the cell containing the particle, giving

$$\begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{1}{12} & -1 & \frac{13}{12} \\ 1 & -1 & \frac{13}{12} & 0 & \frac{1}{12} \\ 1 & 0 & \frac{1}{12} & 0 & \frac{1}{12} \\ 1 & 1 & \frac{13}{12} & 0 & \frac{1}{12} \\ 1 & 0 & \frac{1}{12} & 1 & \frac{13}{12} \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \end{bmatrix} . \quad (21.25)$$

Inverting gives

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & -\frac{1}{24} & \frac{7}{6} & -\frac{1}{24} & -\frac{1}{24} \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix} . \quad (21.26)$$

Similarly, in three dimensions, the interpolation function is

$$f(x, y, z) = A + B(x - x_i) + C(x - x_i)^2 + D(y - y_j) + E(y - y_j)^2 + F(z - z_k) + G(z - z_k)^2 . \quad (21.27)$$

and we have

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \\ F\Delta z \\ G\Delta z^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} & \frac{5}{4} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} \\ 0 & 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & -1 & 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i,j,k-1} \\ f_{i,j-1,k} \\ f_{-i,j,k} \\ f_{i,j,k} \\ f_{i+1,j,k} \\ f_{i,j+1,k} \\ f_{i,j,k+1} \end{bmatrix} . \quad (21.28)$$

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

$$\begin{aligned} f(r, z) = & A + B(r - r_i) + C(r - r_i)^2 + D(z - z_j) \\ & + E(z - z_j)^2 + F(r - r_i)(z - z_j) . \end{aligned} \quad (21.29)$$

and

$$\begin{bmatrix} A \\ B\Delta r \\ C\Delta r^{\frac{2}{6}} \\ D\Delta z \\ E\Delta z^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & -\frac{h_1-1}{24h_1} & \frac{7}{6} & -\frac{h_1-1}{24h_1} & -\frac{1}{24} \\ 0 & -\frac{(7+6h_1)(h_1-1)}{3h_2} & \frac{2h_1}{3h_2} & \frac{(7+6h_1)(h_1-1)}{3h_2} & 0 \\ 0 & \frac{(12h_1^2+12h_1-1)(h_1-1)}{h_1h_2} & -2\frac{12h_1^2-13}{h_2} & -\frac{(12h_1^2+12h_1-1)(h_1-1)}{h_1h_2} & 0 \\ -\frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix}. \quad (21.30)$$

### 21.2.2 Cloud in Cell Mapping

Other interpolation routines can be defined that take into account the actual quantities defined on the grid. These “mesh-based” algorithms are represented in FLASH4 by the Cloud-in-Cell mapping, where the interpolation to/from the particles is defined as a simple linear weighting from nearby grid points. The weights are defined by considering only the region of one “cell” size around each particle location; the proportional volume of the particle “cloud” corresponds to the amount allocated to/from the mesh. The CIC method can be used with both types of particles. When using it with active particles the MapToMesh methods should also be selected. In order to include the CIC method with passive particles, the `setup` command line option is `-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.

## 21.3 Using the Particles Unit

The Particles unit encompasses nearly all aspects of Lagrangian particles. The exceptions are input/output the movement of related data structures between different blocks as the particles move from one block to another, and mapping the particle attributes to and from the grid.

Beginning with release of version 4 it is possible to add particles to a simulation during evolution, a new function `Particles_addNew` has been added to the unit’s API for this purpose. It has been possible to include multiple different types of particles in the same simulation since release FLASH3.1. Particle types must be specified in the `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 subunits; a brief description of each subunit is included below and further expanded in subsections in this chapter.

- The `ParticlesInitialization` subunit distributes a given set of particles through the spatial domain at the simulation startup. Some type of spatial initialization is always required; the functionality is provided by `Particles_initPositions`. The users of active particles typically have their own custom initialization. The following two implementations of initialization techniques are included in the FLASH4 distribution (they are more likely to be used with the passive tracer particles):



**Lattice** distributes particles regularly along the axes directions throughout a subsection of the physical grid.

**WithDensity** distributes particles randomly, with particle density being proportional to the grid gas density.

Users have two options for implementing custom initialization methods. The two files involved in the process are: **Particles\_initPositions** and **pt\_initPositions**. The former does some housekeeping such as allowing for inclusion of one of the available methods along with the user specified one, and assigning tags at the end. A user wishing to add one custom method with no constraints on tags etc is advised to implement a custom version of the latter. This approach allows the user to focus the implementation on the placement of particles only. Users desirous of refining the grid based on particles count during initialiation should see the setup **PoisParticles** for an example implementation of the **Particles\_initPositions** routine. If more than one implementation of **pt\_initPositions** is desired in the same simulation then it is necessary to implement each one separately with different names (as we do for tracer particles: **pt\_initPositionsLattice** and **pt\_initPositionsWithDensity**) in their simulation setup directory. In addition, a modified copy of **Particles\_initPostions**, which calls these new routines in the loop over types, must also be placed in the same directory.

- The **ParticlesMain** subunit contains the various time-integration options for both active and passive particles. A detailed overview of the different schemes is given in [Section 21.1](#).
- The **ParticlesMapping** subunit controls the mapping of particle properties to and from the grid. FLASH currently supplies the following mapping schemes:

**Cloud-in-cell** (**ParticlesMapping/meshWeighting/CIC**), which weights values at nearby grid cells; and

**Quadratic** (**ParticlesMapping/Quadratic**), which performs quadratic interpolation.

Some form of mapping must always be included when running a simulation with particles. As mentioned in [Section 21.2](#) the quadratic mapping scheme is only available to map *from* the grid quantities to the corresponding particle attributes. Since active particles require the same mapping scheme to be used in mapping to and from the mesh, they cannot use the quadratic mapping scheme as currently implemented in FLASH4. The CIC scheme may be used by both the active and passive particles.

For active particles, we use the mapping routines to assign particles' mass to the particle density grid-based solution variable (**PDEN\_VAR**). This mapping is the initial step in the particle-mesh (PM) technique for evaluating the long range gravitational force between all particles. Here, we use the particle mapping routine **Particles\_mapToMeshOneBlk** to “smear” the particles' attribute over the cells of a temporary array. The temporary array is an input argument which is passed from the grid mapping routine **Grid\_mapParticlesToMesh**. This encapsulation means that the particle mapping routine is independent of the current state of the grid, and is not tied to a particular **Grid** implementation. For details about the task of mapping the temporary array values to the cells of the appropriate block(s), please see [Section 8.9.2](#). New schemes can be created that “smear” the particle across many more cells to give a more accurate **PDEN\_VAR** distribution, and thus a higher quality force approximation between particles. Any new scheme should implement a customized version of the **pt\_assignWeights** routine, so that it can be used by the **Particles\_mapToMeshOneBlk** routine during the map.

- The **ParticlesForces** subunit implements the long and short range forces described in Equation (21.3) in the following directories:
  - **longRange** collects different long-range force laws (requiring elliptic solvers or the like and dependent upon all other particles);
  - **shortRange** collects different short-range force laws (directly summed or dependent upon nearest neighbors only).

Currently, only one long-range force law (gravitation) with one force method (particle-mesh) is included with FLASH. Long-range force laws are contained in the `Particles/ParticlesForces/longRange`, which requires that the `Gravity` unit be included in the code. In the current release, no `shortRange` implementation of `ParticlesForces` is supplied with FLASH. However, note that the sink particle implementation described below in [Section 21.4](#) includes directly computed particle-particle forces.

After particles are moved during time integration or by forces, they may end up on other blocks within or without the current processor. The redistribution of particles among processors is handled by the `GridParticles` subunit, as the algorithms required vary considerably between the grid implementations. The boundary conditions are also implemented by the `GridParticles` unit. See [Section 8.9](#) for more details of these redistribution algorithms. The user should include the option `-with-unit=Grid/GridParticles` on the setup line, or `REQUIRES Grid/GridParticles` in the Config file.

In addition, the input-output routines for the Particles unit are contained in a subunit `IOParticles`. Particles are written to the main checkpoint files. If the user desires, a separate output file can be created which contains only the particle information. See [Section 21.3.3](#) below as well as [Section 9.2.3](#) for more details. The user should include the option `-with-unit=IO/IOParticles` on the setup line, or `REQUIRES IO/IOParticles` in the Config file.

In FLASH4, the initial particle positions can be used to construct an appropriately refined grid, i.e. more refined in places where there is a clustering of particles. To use this feature the `flash.par` file must include: `refine_on_particle_count=.true.` and `max_particles_per_blk=[some value]`. Please be aware that FLASH will abort if the criterion is too demanding. To overcome the abort, specify a less demanding criterion, or increase the value of `lrefine_max`.

### 21.3.1 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. If the variable `initializeParticleAtRestart` is `.true.`, then particle data is initilized from scratch instead of a checkpoint file when restarting a simulation. 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.

### 21.3.2 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 21.1](#)). 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.

### 21.3.3 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`.

### 21.3.4 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. To 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 21.1.3. Add other options as desired.

For `unitTest/ParticlesAdvance/HomologousPassive`,

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

is recommended to get started.

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

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

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

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

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

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

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

where `maxError` is the maximum absolute error between numerical and analytic solution for any particle that was encountered during a simulation run, and `maxActualDt` is the maximum time step  $\Delta t$  used in the run.

The appropriate runtime parameters of various units, in particular `Driver`, `Particles`, and `Grid`, should be used to control the desired simulation run. In particular, it is recommended to vary `dtmax` by several orders of magnitude (over a range where it directly determines `maxActualDt`) for a given test in order to examine convergence behavior.

## 21.4 Sink Particles

### 21.4.1 Basics of Sink Particles

Sink particles are required in collapse simulations to model dense core, star, or black hole formation and accretion. Using sink particles solves two main problems in collapse calculations:

1. The physical length scale associated with the collapse, the Jeans length,

$$\lambda_J = \sqrt{\frac{\pi c_s^2}{G\rho}} \quad (21.31)$$

decreases with increasing gas density  $\rho$  (here,  $G$  is the gravitational constant and  $c_s$  the sound speed). To avoid artificial fragmentation, the Jeans length must be resolved with at least 4 grid cells (Truelove et al. 1997, ApJ 489, L179). More recent calculations show that a minimum of 32 grid cells is required to resolve the kinetic energy in rotational motions inside the Jeans volume and to resolve minimum turbulent MHD dynamo action (Sur et al. 2010, ApJ 721, L134; Federrath et al. 2011, ApJ 731, 62). The FLASH sink particle module automatically refines the AMR grid based on the Jeans length (controlled by runtime parameters `refineOnJeansLength`, `jeans_ncells_ref`, and `jeans_ncells_deref`). However, once the highest level of the AMR hierarchy is reached, sink particles must be created to avoid artificial fragmentation when the Jeans length drops further during the collapse.

2. The typical timescale for collapsing objects is the freefall time,  $t_{\text{ff}} = \sqrt{3\pi/(32G\rho)}$ . With increasing gas density, this timescale becomes shorter and shorter, which means that the code will literally grind to a halt during runaway collapse, as time steps become shorter and shorter. The first object going into freefall collapse thus determines the time step of the whole calculation, such that following the formation of a star cluster would be impossible. To avoid these problems, the gas in regions that are in a state of runaway collapse are replaced by sink particles.

### 21.4.2 Using the Sink Particle Unit

To include sink particles, specify `REQUIRES Particles/ParticlesMain/active/Sink` in the simulation `Config` file. This automatically includes all required subunits for the sink particles. Currently, sink particles require a Paramesh 4 `Grid` implementation and Poisson gravity. Currently, various subunits and implementation directories of the `Particles` unit and are also included automatically. Table 21.6 lists all runtime parameters of the sink particle module. The default values of these parameters should be ok in most simulations, except `sink_density_thresh`, `sink_accretion_radius`, and `sink_softening_radius`. Those three parameters have to be adopted to the resolution and physics of a given simulation.

First, the sink particle accretion radius,  $r_{\text{acc}}$  (`sink_accretion_radius`) should be calculated, based on the minimum cell size  $\Delta x$  for a given simulation. The recommended value is  $r_{\text{acc}} = 2.5\Delta x$ . If sink particles are included, the code will print a message to the standard output, stating the number of cells that a chosen sink accretion radius corresponds to. The recommended value for the sink particle softening radius (`sink_softening_radius`) is to set it equal to the accretion radius.

In order to avoid artificial fragmentation, the gas density on the AMR grid must not exceed a critical density  $\rho_{\text{thresh}}$  in regions of gravitational collapse (Truelove et al. 1997). This density is related to the smallest resolvable Jeans length  $\lambda_J$  on the highest level of the AMR hierarchy. The density threshold  $\rho_{\text{thresh}}$  (`sink_density_thresh`) is obtained by solving the definition of the Jeans length, Equation (21.31) for  $\rho$ ,

$$\rho_{\text{thresh}} = \frac{\pi c_s^2}{G\lambda_J^2} = \frac{\pi c_s^2}{4Gr_{\text{acc}}^2}. \quad (21.32)$$

This equation relates the sink particle accretion radius to the sink particle density threshold and depends on the thermodynamics (sound speed) of a given simulation. Since the Jeans length should be resolved with at least 4 grid cells (Truelove et al. 1997), the sink particle accretion radius  $r_{\text{acc}}$  must not be smaller than 2 grid cells. The accretion radius is thus determined by the smallest linear cell size  $\Delta x$  of the AMR grid. As recommended above, setting  $r_{\text{acc}} \simeq 2.5\Delta x$  satisfies the Truelove criterion, because the Jeans length  $\lambda_J = 2r_{\text{acc}}$  is thus resolved with 5 grid cells on the top level of the AMR hierarchy.

Please cite Federrath et al. (2010, ApJ 713, 269) if you are using this unit.

### 21.4.3 The Sink Particle Method

For technical details and tests of the FLASH sink particle unit, see Federrath et al. (2010, ApJ 713, 269) and Federrath et al. (2011, IAUS 270, 425). We only summarize here the most important aspects of the implementation.

Before a sink particle is created, we define a spherical control volume with the accretion radius  $r_{\text{acc}}$  around each cell that exceeds the resolution-dependent density threshold,  $\rho_{\text{thresh}}$  given by Equation (21.32), and check whether the gas in this control volume

- is on the highest level of refinement,
- is converging ( $\nabla \cdot \mathbf{v} < 0$ ; i.e., has negative radial velocity),
- has a central gravitational potential minimum,
- is gravitationally bound ( $|E_{\text{grav}}| > E_{\text{int}} + E_{\text{kin}} + E_{\text{mag}}$ ),
- is gravitationally unstable (Jeans-unstable),
- and is not within the accretion radius of an existing sink particle.

These checks are designed to avoid spurious sink particle formation and to trace only truly collapsing and star-forming gas. As soon as a sink particle is created, it can gain further mass by accretion from the AMR grid, if this gas is inside the sink particle accretion radius and is bound and collapsing towards the sink particle. If the accretion radii of multiple sink particles overlap, the gas of a given cell is accreted onto the sink particle to which the gas is most strongly bound.

We take all contributions to the gravitational interactions between the gas on the grid and the sink particles into account. Those interactions are

1. GAS–GAS (handled by either the multigrid or the tree Poisson solver)
2. GAS–SINKS (computed by direct summation over all particles and grid cells)
3. SINKS–GAS (computed by direct summation over all particles and grid cells)
4. SINKS–SINKS (computed by direct N-body summation over all sink particles).

We note that all interactions including sink particles (GAS–SINKS, SINKS–GAS, SINKS–SINKS) do not require the Poisson solver or any particle–grid mapping procedure. Since these are all computed by direct summation, computational costs can become quite expensive once the number of sink particles reaches thousands and more, but is significantly more accurate than mapping procedures. In fact, a previous implementation used interpolation of sink particle masses back onto the grid and employing the Poisson solver to compute all interactions. This resulted in very smooth gravitational interactions that made it impossible to follow close orbits and highly eccentric encounters of multiple (two and more) sink particles, in particular for the sink–sink interactions. Thus, the current implementation of sink particles is designed to follow a maximum of a few thousand particles. Linear and spline softening of the gravitational forces for extremely close encounters of multiple sink particles are implemented. By default, linear softening is used for gas–sinks and sinks–gas interactions (`sink_softening_type_gas`), motivated by an almost uniform gas density inside the sink particle radius. Spline softening is used for sink–sink interactions (`sink_softening_type_sinks`), as it is more suitable to follow N-body dynamics. Softening is only applied inside the sink softening radius (`sink_softening_radius`). A second-order accurate leapfrog integrator is used to advance the sink

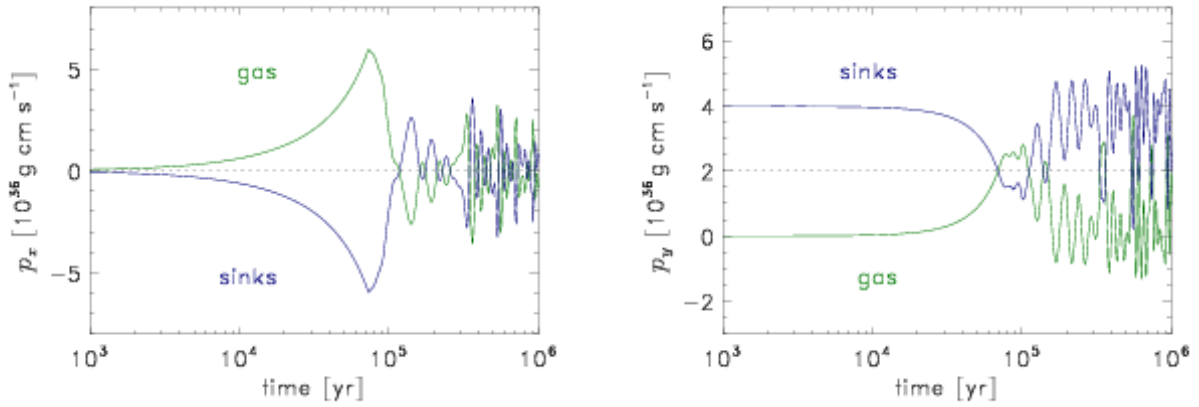


Figure 21.3: Sink particle unit test to check momentum conservation during sink particle creation, accretion, and eccentric orbits.

particles. For cosmological simulations, a cosmological version of the leapfrog integrator is available (see `sink_integrator`). The sink particles are fully integrated into the MPI parallelization of the FLASH code. Any split or unsplit solver for hydro or MHD (in which case the bound state check includes the contribution of the magnetic energy) is supported.

#### 21.4.4 Sink Particle Unit Test

To invoke the sink particle unit test, use `./setup unitTest/SinkMomTest -auto -3d`. This initializes a momentum conservation test. An initially uniform cloud with radius  $0.025 \text{ pc}$  and density  $10^{-18} \text{ g cm}^{-3}$  at rest starts collapsing and forms a sink particle. An initial sink particle with  $0.1 M_{\odot}$  and initial momentum  $p_y = 4 \times 10^{36} \text{ g cm s}^{-1}$  in positive  $y$ -direction is also present. The purpose of this test is to see how well the total  $x$ -momentum  $p_x$  and the total  $y$ -momentum  $p_y$  are conserved. This test is particularly suitable, because it involves gas collapse, sink particle creation, accretion, more than one sink particle, and thus all gravitational interactions (gas–gas, gas–sinks, sinks–gas, sinks–sinks), as well as close eccentric orbits of the two sink particles. Figure 21.3 shows  $p_x$  and  $p_y$  as a function of time for both sink particles and gas separately. The symmetry between sink and gas momenta shows that momentum is well conserved for several orbits. Figure 21.3 can be reproduced with the IDL and Python tools in `/source/Simulation/SimulationMain/unitTest/SinkMomTest/utils/` by running the IDL script `plot_mom_sinks.pro`.

Table 21.6: Runtime parameters for the sink particle module.

Variable	Type	Default	Description
<code>useSinkParticles</code>	BOOLEAN	<code>.false.</code>	switch sinks on/off
<code>sink_density_thresh</code>	REAL	1.0e-14	density threshold for sink creation and accretion
<code>sink_accretion_radius</code>	REAL	1.0e14	creation and accretion radius
<code>sink_softening_radius</code>	REAL	1.0e14	gravitational softening radius
<code>sink_softening_type_gas</code>	STRING	<code>"linear"</code>	sink-gas softening type (options: <code>"linear"</code> , <code>"spline"</code> )
<code>sink_softening_type_sinks</code>	STRING	<code>"spline"</code>	sink-sink softening type (options: <code>"linear"</code> , <code>"spline"</code> )
<code>sink_integrator</code>	STRING	<code>"leapfrog"</code>	sink particle time integrator (options: <code>"euler"</code> , <code>"leapfrog"</code> , <code>"leapfrog_cosmo"</code> )
<code>sink_dt_factor</code>	REAL	0.5	time step safety factor ( $\leq 0.5$ )
<code>sink_subdt_factor</code>	REAL	0.01	time step safety factor for sink-sink subcycling ( $\leq 0.5$ )
<code>sink_convergingFlowCheck</code>	BOOLEAN	<code>.true.</code>	creation check for converging gas flow
<code>sink_potentialMinCheck</code>	BOOLEAN	<code>.true.</code>	creation check for gravitational potential minimum
<code>sink_jeansCheck</code>	BOOLEAN	<code>.true.</code>	creation check for Jeans instability
<code>sink_negativeEtotCheck</code>	BOOLEAN	<code>.true.</code>	creation check for gravitationally bound gas
<code>sink_GasAccretionChecks</code>	BOOLEAN	<code>.true.</code>	check for bound and converging state before gas accretion
<code>sink_merging</code>	BOOLEAN	<code>.false.</code>	switch for sink particle merging
<code>sink_offDomainSupport</code>	BOOLEAN	<code>.false.</code>	support for sink particles to remain active when leaving the grid domain (in case of outflow boundary conditions)
<code>sink_AdvanceSerialComputation</code>	BOOLEAN	<code>.true.</code>	use the global sink particle array for time advancement (to greatly speed up computation of sink-sink interaction)
<code>pt_maxSinksPerProc</code>	INTEGER	100	number of sinks per processor
<code>refineOnSinkParticles</code>	BOOLEAN	<code>.true.</code>	sinks must be on highest AMR level
<code>refineOnJeansLength</code>	BOOLEAN	<code>.true.</code>	switch for refinement on Jeans length
<code>jeans_ncells_ref</code>	REAL	32.0	number of cells for Jeans length refinement
<code>jeans_ncells_deref</code>	REAL	64.0	number of cells for Jeans length de-refinement





## Chapter 22

# Cosmology Unit

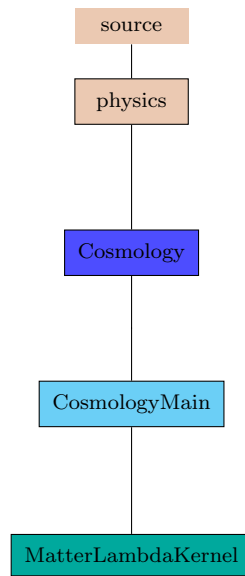


Figure 22.1: The `Cosmology` unit tree.

The `Cosmology` unit solves the Friedmann equation for the scale factor in an expanding universe, applies a cosmological redshift to the hydrodynamical quantities, and supplies library functions for various routine cosmological calculations needed by the rest of the code for initializing, performing, and analyzing cosmological simulations.

### 22.1 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  $\dot{\mathbf{x}}$ . The comoving gas

density, pressure, temperature, and internal energy are defined to be

$$\begin{aligned}\rho &\equiv a^3 \tilde{\rho} \\ p &\equiv a \tilde{p} \\ T &\equiv \frac{\tilde{T}}{a^2} \\ \rho\epsilon &\equiv a \tilde{\rho} \tilde{\epsilon} .\end{aligned}\tag{22.1}$$

The quantities marked with a tilde, such as  $\tilde{\rho}$ , are the corresponding “proper” or physical quantities. Note that, in terms of comoving quantities, the equation of state has the same form as for the proper quantities in noncomoving coordinates. For example, the perfect-gas equation of state is

$$\rho\epsilon = \frac{p}{\gamma - 1} = \frac{\rho k T}{(\gamma - 1)\mu} .\tag{22.2}$$

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

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0\tag{22.3}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \mathbf{v}) + \nabla p + 2 \frac{\dot{a}}{a} \rho \mathbf{v} + \rho \nabla \phi = 0\tag{22.4}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p) \mathbf{v}] + \frac{\dot{a}}{a} [(3\gamma - 1) \rho\epsilon + 2\rho v^2] + \rho \mathbf{v} \cdot \nabla \phi = 0\tag{22.5}$$

$$\frac{\partial \rho\epsilon}{\partial t} + \nabla \cdot [(\rho\epsilon + p) \mathbf{v}] - \mathbf{v} \cdot \nabla p + \frac{\dot{a}}{a} (3\gamma - 1) \rho\epsilon = 0 .\tag{22.6}$$

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

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

$$\nabla^2 \phi = \frac{4\pi G}{a^3} (\rho - \bar{\rho}) ,\tag{22.7}$$

where  $\bar{\rho}$  is the comoving mean matter density. Note that, because of the presence of  $a$  in (22.7), 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 (22.7). Accounting for particle masses in density is handled by the `Gravity` unit.

The comoving mean matter density is defined in terms of the critical density  $\rho_{\text{crit}}$  by

$$\begin{aligned}\bar{\rho} &\equiv \Omega_m \rho_{\text{crit}} \\ \rho_{\text{crit}} &\equiv \frac{3H^2}{8\pi G} .\end{aligned}\tag{22.8}$$

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

$$H^2(t) \equiv \left(\frac{\dot{a}}{a}\right)^2 = H_0^2 \left( \frac{\Omega_m}{a^3} + \frac{\Omega_r}{a^4} + \Omega_\Lambda - \frac{\Omega_c}{a^2} \right) .\tag{22.9}$$

Here  $\Omega_m$ ,  $\Omega_r$ , and  $\Omega_\Lambda$  are the present-day densities, respectively, of matter, radiation, and cosmological constant, divided by  $\rho_{\text{crit}}$ . The contribution of the overall spatial curvature of the universe is given by

$$\Omega_c \equiv \Omega_m + \Omega_r + \Omega_\Lambda - 1 .\tag{22.10}$$

The `Cosmology_solveFriedmannEqn` routine numerically solves the Friedmann equation to obtain the scale factor and its rate of change as functions of time. In principle, any good ODE integrator can be used; the `csm_integrateFriedman` subroutine uses a fourth-order Runge-Kutta method to integrate the Friedmann equation under the assumption that  $\Omega_r = 0$ . Subunits can also use analytic solutions where appropriate.

Redshift terms for particles are handled separately by the appropriate time integration subunits of the `Particles` unit. For an example, see the `LeapfrogCosmo` implementation of the `ParticlesMain` subunit in [Section 21.1.1](#).

## 22.2 Using the Cosmology unit

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

```
REQUESTS physics/Cosmology/
```

in your setup's `Config` file. At present the Cosmology unit in FLASH4 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 22.1](#). 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 22.1: Runtime parameters used with the `Cosmology` unit.

Parameter	Type	Default	Description
<code>useCosmology</code>	BOOLEAN	<code>.true.</code>	True if cosmology is to be used in this simulation
<code>OmegaMatter</code>	REAL	0.3	Ratio of total mass density to critical density at the present epoch ( $\Omega_m$ )
<code>OmegaBaryon</code>	REAL	0.05	Ratio of baryonic (gas) mass density to critical density at the present epoch; must be $\leq \text{OmegaMatter}$ ( $\Omega_b$ )
<code>CosmologicalConstant</code>	REAL	0.7	Ratio of the mass density equivalent in the cosmological constant to the critical density at the present epoch ( $\Omega_\Lambda$ )
<code>OmegaRadiation</code>	REAL	$5 \times 10^{-5}$	Ratio of the mass density equivalent in radiation to the critical density at the present epoch ( $\Omega_r$ )
<code>HubbleConstant</code>	REAL	$2.1065 \times 10^{-18}$	Value of the Hubble constant $H_0$ in $\text{sec}^{-1}$
<code>MaxScaleChange</code>	REAL	<code>HUGE(1.)</code>	Maximum permitted fractional change in the scale factor during each timestep

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.

## 22.3 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 × 10-18sec-1 (50 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 23

# Material Properties Units

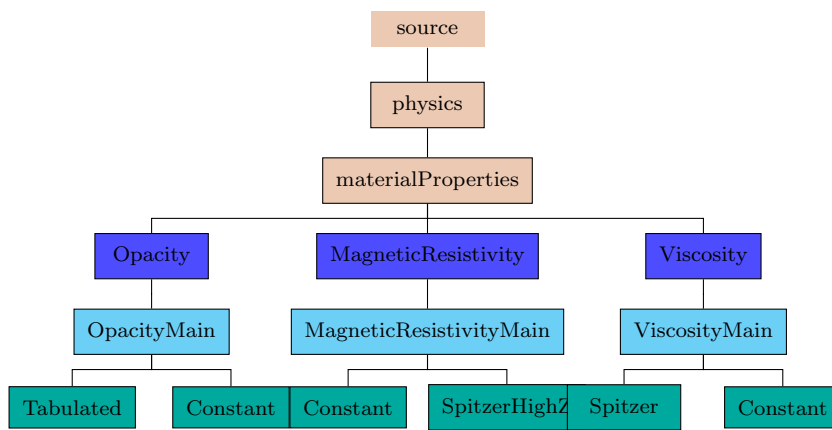


Figure 23.1: The `materialProperties` directory with `Opacity`, `MagneticResistivity` and `Viscosity` subunits.

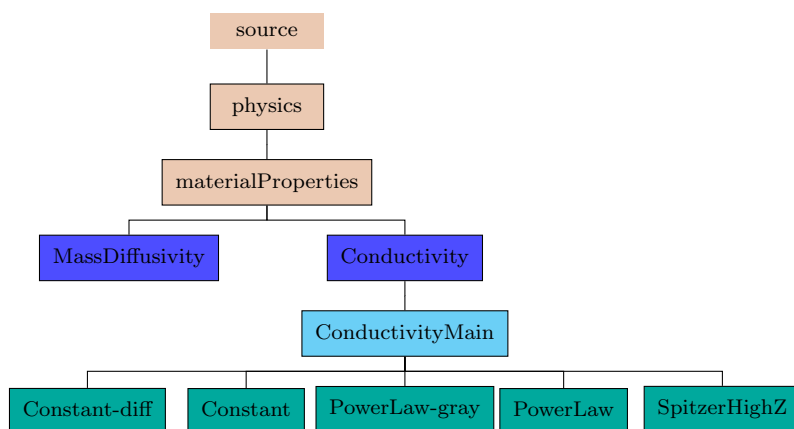


Figure 23.2: The `materialProperties` directory with `MassDiffusivity` and `Conductivity` subunits.

### FLASH Transition

The set of implementations of the material properties units provided with FLASH is not comprehensive. For **Heat Conductivity** and **Viscosity**, we provide implementations for effects with constant coefficients; these can be used as models for implementing effects that follow other laws. For **MassDiffusivity**, only no-operation stubs are provided. A routine that calculates constant magnetic resistivity and viscosity is provided in the **MagneticResistivity** unit and can be used in non-ideal magnetohydrodynamics simulations. Several add-on capabilities are being made available to the users from the [Download Page on Flash Website](#).

## 23.1 Thermal Conductivity

The **Conductivity** unit implements a prescription for computing thermal conductivity coefficients used by the Hydro PPM, the unsplit hydro and MHD solvers. The FLASH4 release provides four implementations:

- **Constant** for heat conduction with a constant isochoric conductivity.
- **Constant-diff** for heat conduction with a constant coefficient of diffusion.
- **SpitzerHighZ** which is used for electron thermal conduction. Note that this model can be used with any material.
- **LeeMore** is another model for electron thermal conduction. Like **SpitzerHighZ**, it can also be used with any material. The **LeeMore** model is based on Lee & More (Phys. Fluids, 1984) and should be more accurate than **SpitzerHighZ**.

To use thermal conductivity in a FLASH4 simulation, the runtime parameter **useConductivity** must be set to **.true**. **SpitzerHighZ** and **LeeMore** are useful options for realistic HEDP simulations. See [Section 35.7.5](#) for an example of how the **SpitzerHighZ** implementation is used in a realistic simulation.

The Spitzer conductivity implemented here is shown in (23.1). It is consistent with the value given in (Atzeni, 2004).

$$K_{\text{ele}} = \left(\frac{8}{\pi}\right)^{3/2} \frac{k_B^{7/2}}{e^4 \sqrt{m_{\text{ele}}}} \left(\frac{1}{1 + 3.3/\bar{z}}\right) \frac{T_{\text{ele}}^{5/2}}{\bar{z} \ln \Lambda_{ei}} \quad (23.1)$$

where:

- $K_{\text{ele}}$  is the electron conductivity
- $k_B$  is the Boltzmann constant
- $e$  is the electron charge
- $m_{\text{ele}}$  is the mass of an electron
- $\bar{z}$  is the average ionization as computed by the EOS
- $T_{\text{ele}}$  is the electron temperature
- $\ln \Lambda_{ei}$  is the Coulomb logarithm associated with electron-ion collisions and is discussed in [Section 18.5](#).

At high temperatures, **LeeMore** and **SpitzerHighZ** are nearly identical aside from differences in the treatments of the Coulomb logarithm ([Section 18.5.2](#)). At lower temperatures there can be substantial differences between **LeeMore** and **SpitzerHighZ** and generally the **LeeMore** model should be much more accurate. This is because **SpitzerHighZ** incorrectly assumes that the material remains a classical plasma even at low temperatures. In practice, however, it is often the case in HEDP simulations that the laser

heating will rapidly bring the material to the temperatures where the classical plasma approximation is valid and the differences between `SpitzerHighZ` and `LeeMore` will be minimal.

Users are encouraged to experiment with using both the `SpitzerHighZ` and `LeeMore` models in HEDP simulations in order to determine the sensitivity of the results to the electron thermal conductivity model. Although `LeeMore` should always be much more accurate than `SpitzerHighZ`, there may be cases where the `LeeMore` model may still not be accurate enough for a specific application. It is well known that some materials do not agree well with the `LeeMore` prediction in certain density-temperature regimes (e.g. Desjarlais et al. 2002).

### 23.1.1 Anisotropic Thermal Conductivity

There are now implementations within the `Conductivity` unit which calculate and use anisotropic thermal conductivities. For full functionality, users should include the setup shortcut `+supportAnisoCond` in their setup command. This will enable the code to calculate and store the thermal conductivities  $K_{\parallel}$ ,  $K_{\perp}$ , and  $K_{\wedge}$  (Righi-Leduc). If the user chooses to use one of the anisotropic thermal conductivity implementations without this setup shortcut, and without turning on relevant logical switches in their `flash.par` file, the code will do isotropic thermal diffusion with the  $K_{\perp}$  coefficient. This is most suitable for 1D and 2D problems in which the B-field is only into or out of the page. Otherwise, it only makes sense to use these conductivities in conjunction with the anisotropic thermal diffusion solver (see [Section 19.1.5](#) for more information).

There are two implementations that calculate anisotropic coefficients:

- `EppHain` gives coefficients defined by Epperlein & Haines (Phys. Fluids, 1986).
- `JiHeld` gives coefficients defined by Ji & Held (Phys. Plasmas, 2013).

These are currently two of the most accurate implementations in the code (for MHD simulations), and should be strongly considered for use in magnetic HEDP applications. Alternatively, the user may choose to use constant coefficients by using the `ConductivityMain/Constant` implementation. In this case, the following runtime parameters should be defined:

- `cond_constantParallel`: thermal conductivity  $K_{\parallel}$
- `cond_constantPerpendicular`: thermal conductivity  $K_{\perp}$
- `cond_constantCross`: thermal conductivity  $K_{\wedge}$

Note that the current implementation does not allow for different constant values for electrons and ions, thus the `Constant` implementation should not be used for physically realistic 3T applications.

For a more detailed description of how to control anisotropic thermal diffusion, see [Section 19.1.5](#).

## 23.2 Magnetic Resistivity

The magnetic resistivity unit `source/physics/materialProperties/MagneticResistivity` provides routines that compute magnetic resistivity  $\eta$  for a mixture of ionized gases used by the MHD solvers. The default top level routines return zero values for resistivity (a stub functionality). Specific routines for constant and variable resistivity are provided in the low level subdirectories within `/MagneticResistivityMain`. By default, all routines return results in non-dimensional units (hence without  $4\pi$  or  $\mu_0$  coefficients). In other words, it is magnetic diffusivity in  $\text{cm}^2 \text{s}^{-1}$  that is actually returned. However they provide an option to return results either in CGS or SI unit.

This unit now supports fully anisotropic magnetic resistivity. The resistivity components  $\eta_{\parallel}$ ,  $\eta_{\perp}$ , and  $\eta_{\wedge}$  may be used depending on the value of the runtime parameter `resistivityForm`. Valid values for this parameter are: “parallel”, “perpendicular”, or “fullaniso”. This not only controls which resistivity coefficients are computed, but also how/if they are used when solving the induction equation. The default value for `resistivityForm` is “perpendicular”. See [Section 19.1.6](#) for more details on how resistivity is implemented with implicit magnetic diffusion, and [Section 15.3.5](#) for how the  $\eta_{\wedge}$  term is implemented.

### 23.2.1 Constant resistivity

This implementation 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.

Different constants may also be set for anisotropic coefficients using runtime parameters: `res_constantParallel`, `res_constantPerpendicular`, and `res_constantCross`. Note that all of these values will be overwritten by `resistivity` if `resistivity` is non-zero.

#### FLASH 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.

### 23.2.2 Spitzer HighZ resistivity

This implementation returns the Spitzer resistivity for the induction equation and is a useful option for HEDP simulations. The implementation follows the prescription of Braginskii (1965), defining a perpendicular and parallel magnetic resistivity, with respect to the field line, as

$$\eta_{\perp} = \frac{m_{ele}}{e^2 n_{ele} \tau_{ele}}, \quad (23.2)$$

$$\eta_{\parallel} = \eta_{\perp}/1.96. \quad (23.3)$$

Here we denote with  $m_{ele}$ ,  $e$ ,  $n_{ele}$ ,  $\tau_{ele}$  the mass, charge, number density and the collision time of electrons, respectively. In its current form, the `/MagneticResistivityMain/SpitzerHighZ/MagneticResistivity_fullState` routine is used and will return parallel or both parallel and perpendicular components depending on the value of the runtime parameter `resistivityForm`. Note that this implementation will always return zero for the  $\eta_{\wedge}$  component. To activate the Spitzer HighZ resistivity implementation, simply add the appropriate path in the `Config` file of your simulation directory.

### 23.2.3 DaviesWen resistivity

This implementation returns magnetic resistivity coefficients based on the work by Davies, et al. (Phys. Plasmas, 2021). All three resistivity components  $\eta_{\parallel}$ ,  $\eta_{\perp}$ , and  $\eta_{\wedge}$  are computed with dependencies on temperature, ionization, atomic number, and Hall parameter making this the most complete implementation for anisotropic resistivity suitable for HEDP applications.

To activate the DaviesWen resistivity implementation, simply add the appropriate path in the `Config` file of your simulation directory.

### 23.2.4 Vacuum resistivity

Both the Spitzer HighZ and DaviesWen implementations described above have the capability for the user to define an additional constant resistivity for vacuum regions. The runtime parameter for this constant vacuum resistivity is `res_vacRes`, which has a default value of zero. The value of `res_vacRes` will overwrite all previously calculated resistivity values when the mass density in a cell is less than or equal to the runtime parameter `res_vacDens`. The default value for `res_vacDens` is zero. The use of `res_vacRes` can be further



controlled by the runtime parameters `res_vacSpec` and `res_vacFrac`. The default value of `res_vacSpec` is “vacu” and defines the name of the species (specified during setup) that is to be considered vacuum. A vacuum species mass fraction greater than `res_vacFrac` will be considered vacuum, and the default value of `res_vacFrac` is 0.5. If a species with the name specified by `res_vacSpec` does not exist, then only the density condition with `res_vacDens` is considered.

These runtime parameters `res_vacDens`, `res_vacSpec`, and `res_vacFrac` are also used to determine whether Ohmic heating should be calculated in a given cell. In other words, when these parameters are set such that a given cell is considered to be vacuum, then the cell’s resistivity will be `res_vacRes` and Ohmic heating will not be calculated. Note that this functionality to ignore Ohmic heating in vacuum regions is only implemented in the implicit magnetic diffusion solver.

## 23.3 Viscosity

The viscosity unit `physics/materialProperties/Viscosity` 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 two implementations. The `Viscosity/ViscosityMain/Constant` provides for either a constant dynamic viscosity or constant kinematic viscosity, where the choice between the two is made with the runtime parameter `visc_whichCoefficientIsConst`. The `Viscosity/ViscosityMain/Spitzer` unit implements an isotropic Spitzer type viscosity.

### 23.3.1 Constant Viscosity

To use viscosity in a FLASH4 simulation, the runtime parameter `useViscosity` must be set to `.true`. See ?? in the hydro module documentation for details on how the solver uses these modules. The `viscosity` sub-module implements a prescription for computing viscosity coefficients used by the hydro solver. To use viscosity in a FLASH simulation, the runtime parameter `diffus_visc` must be set to `.true`. See ?? in the hydro module documentation for details on how the solver uses these modules.

### 23.3.2 Spitzer Viscosity

This module implements a coefficient of plasma ion viscosity according to the classical Spitzer(1962) prescription:

$$\mu_{ion} = \frac{0.406 \cdot m_{ion}^{1/2} (kT_{ion})^{5/2}}{Z^4 e^4 \ln \Lambda} \quad (23.4)$$

$$= 2.21 \cdot 10^{-15} \frac{A_{ion}^{1/2} T_{ion}^{5/2}}{Z^4 \log \Lambda} \quad (23.5)$$

and the usual units of a dynamic viscosity [gm cm<sup>-1</sup> sec<sup>-1</sup>]. Here  $m_{ion}$  is the ion mass,  $T_{ion}$  is the ion temperature in Kelvin,  $A$  is the dimensionless average atomic number,  $Z$  is the dimensionless average ionization state, and  $\log \Lambda$  is the ion-ion coulomb logarithm.

The kinematic viscosity is also computed as:

$$\nu_{ion} = \frac{\mu_{ion}}{\rho} \quad (23.6)$$

where  $\rho$  is the fluid density, and the kinematic viscosity has units [cm<sup>2</sup> s<sup>-1</sup>].

## 23.4 Thermoelectric Coefficients

The thermoelectric unit `physics/materialProperties/Thermoelectric` provides routines for computing thermoelectric coefficients which govern the thermoelectric transport of both magnetic fields and heat. The coefficients included are:  $\beta_{\parallel}$ ,  $\beta_{\perp}$ , and  $\beta_{\wedge}$ . These appear in Ohm’s Law as follows:

$$e\mathbf{E}/k = -\beta_{\parallel} \nabla T - (\beta_{\perp} - \beta_{\parallel})(\mathbf{b} \times \nabla T) \times \mathbf{b} - \beta_{\wedge}(\mathbf{b} \times \nabla T), \quad (23.7)$$

where  $e$  is electron charge,  $k$  is Boltzmann's constant,  $\mathbf{b}$  is a unit vector in the direction of the B-field and  $T$  is the electron temperature. The nomenclature in the literature is confusing, but in FLASH, we refer to these terms as Seebeck, CrossField, and Nernst, respectively. Each of these coefficients also appears in the electron heat flux equation, with transport driven by current rather than temperature gradients. There is also a fourth coefficient  $\beta_{new}$  that replaces the combination of  $(\beta_{\parallel} - \beta_{\perp})/\chi$  in the CrossField term when the effective atomic number is greater than or equal to 1. All four coefficients are returned as dimensionless. See [Section 15.3.5](#) for more details on how these coefficients are implemented.

### 23.4.1 Constant thermoelectric coefficients

This implementation returns constant thermoelectric coefficients as set by the following runtime parameters:

- `thel_constantParallel`:  $\beta_{\parallel}$
- `thel_constantPerpendicular`:  $\beta_{\perp}$
- `thel_constantCross`:  $\beta_{\wedge}$
- `thel_constantNew`:  $\beta_{new}$

The default value for all of these is zero. To activate the Constant thermoelectric implementation, simply add the appropriate path in the `Config` file of your simulation directory.

### 23.4.2 DaviesWen thermoelectric coefficients

This implementation returns thermoelectric coefficients based on the work by Davies, et al. (Phys. Plasmas, 2021). These coefficients are dependent on temperature, ionization, atomic number, and electron Hall parameter. They represent the only physically realistic implementation in FLASH, thus should be used in MHD HEDP applications where thermoelectric transport is important.

To activate the DaviesWen thermoelectric implementation, simply add the appropriate path in the `Config` file of your simulation directory.

## 23.5 Opacity

The `Opacity` unit, which resides in `physics/materialProperties/Opacity`, exists to provide opacities for multigroup radiation diffusion to the `RadTrans` unit. Thus, the `Opacity` unit does not have an API for providing continuous opacities. The group structure is specified using parameters which are described in [Chapter 25](#). Opacities are accessed by the `RadTrans` unit using the `Opacity` subroutine.

```
call Opacity(soln, ngrp, opacityAbsorption, opacityEmission, opacityTransport)
```

The first argument is an input and provides the complete state within a cell as provided by the routine `Grid_getBlPtr`. The second input argument is the group number. The last three arguments are outputs which return the absorption, emission, and transport opacities in units of  $1/cm$  using the conditions provided by the `soln` argument. See [Chapter 25](#) for a description of these three opacities. There are currently two `Opacity` implementations: Constant and Multispecies which are described below.

### 23.5.1 Constant Implementation

The Constant implementation is very simple and is useful for running test problems. It sets the absorption, emission, and transport opacities to constant values that are specified using the runtime parameters `op_absorbConst`, `op_emissConst`, and `op_transConst`. Users can provide custom implementations of the Constant `Opacity` subroutine to test different formulas for opacities in FLASH.

The Constant opacity implementation can be included with the following setup option:

```
-with-unit=physics/materialProperties/Opacity/OpacityMain/Constant
```

### 23.5.2 Constcm2g Implementation

The Constcm2g implementation is also very simple and is useful for running test problems. It sets the absorption, emission, and transport opacities, in units of  $\text{cm}^2/\text{g}$ , to constant values that are specified using the runtime parameters `op_absorbConst`, `op_emiConst`, and `op_transConst`. Users can provide custom implementations of the Constcm2g Opacity subroutine to test different formulas for opacities in FLASH.

The Constcm2g opacity implementation can be included with the following setup option:

```
-with-unit=physics/materialProperties/Opacity/OpacityMain/Constcm2g
```

### 23.5.3 BremsstrahlungAndThomson Implementation

The BremsstrahlungAndThomson Opacity implementation assumes that the free-free Bremsstrahlung process is responsible for the coupling between radiation and matter (the `emit_opac` and `absorb_opac` terms) and that simple Thomson scattering dominates the diffusion of radiation through matter (the `trans_opac` term). Therefore, the emission and absorption opacities are given by the following formula, in units of  $\text{cm}^{-1}$ :

$$\kappa_{\text{emit,absorb}} = 3.68 \times 10^{22} g_{ff} (1 - Z)(1 + X) T^{-3.5} \rho^2, \quad (23.8)$$

where  $g_{ff}$  is the Gaunt factor ( $\simeq 1$ ),  $Z$  the fractional abundance of elements heavier than hydrogen,  $X$  the fractional abundance of hydrogen,  $T$  the temperature and  $\rho$  the temperature.

For the transport opacity, dominated by Thompson scattering, we have, in units of  $\text{cm}^{-1}$ :

$$\kappa_{\text{trans}} = 0.2(1 + X)\rho. \quad (23.9)$$

The parameters `op_absorbScale`, `op_emiScale`, `op_transScale` are also introduced allowing the user to scale the three opacities accordingly.

This is a gray opacity implementation. The `ngrp` argument in `Opacity` calls has to be supplied but will be ignored.

The BremsstrahlungAndThomson opacity implementation can be included with the following setup option:

```
-with-unit=physics/materialProperties/Opacity/OpacityMain/BremsstrahlungAndThomson
```

### 23.5.4 OPAL Implementation

The OPAL Opacity implementation in `Opacity/OpacityMain/OPAL` uses a table lookup scheme to read realistic stellar opacities from the OPAL database of Lawrence Livermore National Lab (LLNL). OPAL opacities are functions of density, temperature and composition and are stored in two different datasets: one for high temperature ( $3.75 < \log T < 8.7$ ) based on Grevesse & Sauval (1998) and one for low temperature ( $2.7 < \log T < 4.5$ ) based on Ferguson, Alexander, et al (2005), where  $\log T = \log_{10}(T/\text{K})$ . Each dataset includes files corresponding to different hydrogen mass fractions ( $X$ ). Both datasets assume solar metallicity ( $Z = 0.02$ ). The database we use is exactly the same used by the Modules for Experiments in Stellar Astrophysics (MESA) code (Paxton et al. 2011; 2013; 2015) and the SuperNova Explosion Code (SNEC; Morozova et al. 2015). The `radflaHD/SupernovaRad1D` setup is utilizing the new OPAL implementation to compute the light curve of a supernova (SN) explosion using a progenitor model computed with MESA.

This is a gray opacity implementation. The `ngrp` argument in `Opacity` calls has to be supplied but will be ignored.

### 23.5.5 Multispecies Implementation

In general, the opacity is a strong function of the material (or species in FLASH parlance), frequency (or group number), and state of the fluid. The Multispecies Opacity unit implementation provides the flexibility to specify different opacities to use as a function of species and opacity *type* (either absorption, emission, or transport). It can be included in the simulation using the following setup option:

```
-with-unit=physics/materialProperties/Opacity/OpacityMain/Multispecies
```

The opacities for each species are averaged together based on the relative number densities to produce an average opacity of each type for a given cell. For each species and type, the Multispecies implementation allows the user must specify an opacity model to use. There are currently three commonly used opacity models in FLASH:

- **constant**: This model simply returns a constant value for the opacity in units of  $1/\text{cm}$
- **constcm2g**: This model also returns a constant opacity but uses units of  $\text{cm}^2/\text{g}$
- **tabulated**: This model allows the user to specify the names of files which store opacity tables. These tables store opacities as functions electron temperature and density

The first two opacity models, **constant** and **constcm2g**, are fairly self-explanatory. The **tabulated** opacity model is useful for modeling realistic opacities in a general way. Each table is tabulates an opacity at  $N_D$  discrete density points and  $N_T$  discrete electron temperature points. The table for each species and opacity type can have different a temperature/density grid. However, each table must use the same energy group structure which is consistent with the group structure used by the **RadTrans** unit.

Extraction (interpolation) of opacities from the stored tables is currently done using the bilinear form. For a temperature/density pair  $(t, d)$  inside the grid, the code determines the quadrant corners  $T_1 \leq t \leq T_2$  and  $D_1 \leq d \leq D_2$  and the corresponding four opacities  $\kappa_{xy} = \kappa(T_x, D_y); x, y = 1, 2$  and performs the bilinear interpolation:

$$\tau_1 = \frac{T_2 - t}{T_2 - T_1} \quad (23.10)$$

$$\tau_2 = \frac{t - T_1}{T_2 - T_1} \quad (23.11)$$

$$\delta_1 = \frac{D_2 - d}{D_2 - D_1} \quad (23.12)$$

$$\delta_2 = \frac{d - D_1}{D_2 - D_1} \quad (23.13)$$

$$\kappa(t, d) = \sum_{i=1}^2 \sum_{j=1}^2 \tau_i \delta_j \kappa_{ij}. \quad (23.14)$$

In case the target  $(t, d)$  lays outside the grid boundaries, the corresponding boundary value(s) is(are) taken. This is a temporary solution and will be supplemented with options for calculating more accurate opacities, especially for lower cell temperatures. Two possible options are provided for performing the interpolation: i) on the original tabulated opacities or ii) on the logarithmically (base 10) transformed tabulated opacities.

Currently, the tables must be stored in either the IONMIX or IONMIX4 formats. The exact specification of this format can be found in [Section 23.5.6](#). The IONMIX4 format is the most general since it allows for arbitrary temperature/density points. Users who wish to use their own tabulated opacities may convert their files into the IONMIX4 format for use in FLASH. As [Section 23.5.6](#) describes, IONMIX4 files contain group opacities in units of  $\text{cm}^2/\text{g}$ . Each file contains three different group opacities: a Planck Absorption, a Planck Emission, and a Rosseland opacity. The user must specify which of these opacities to use for each species/opacity type in FLASH.

### 23.5.5.1 Runtime Parameters for the Multispecies Opacity

Use of the Multispecies opacity implementation requires the **Multispecies** unit. At least one species must be defined in the simulation to use the Multispecies opacity. Furthermore, the species must all be specified using the **species** setup variable and cannot be defined using the **SPECIES** keyword in a **Config** file. Please see [Section 11.4](#) for more details.

Once the species are specified, the Multispecies opacity implementation will automatically create a set of runtime parameters to allow users to control how the opacity is computed for each species and opacity type. These runtime parameters are shown in [Table 23.1](#). The symbol **<spec>** in the table should be replaced by the species names as specified using the species setup variable.

Table 23.1: Multispecies Opacity Runtime Parameters

Runtime Parameter	Description
op_<spec>Absorb	Model to use for the absorption opacity
op_<spec>Emiss	Model to use for the emission opacity
op_<spec>Trans	Model to use for the transport opacity
op_<spec>AbsorbConstant	Constant value to use for absorption opacity
op_<spec>EmissConstant	Constant value to use for emission opacity
op_<spec>TransConstant	Constant value to use for transport opacity
op_<spec>FileName	Name of IONMIX file for tabulated opacity
op_<spec>FileType	Tabulated opacity file type, either “IONMIX” or “IONMIX4”

The opacity models for each species are specified using the `op.<spec>[Absorb,Emiss,Trans]` runtime parameters. These parameters can be set to the following string values:

- “op.constant” Use a constant opacity in units of  $1/\text{cm}$ . The constant values are specified using the appropriate `op_<spec><type>Constant` runtime parameter.
- “op.constcm2g” Use a constant opacity in units of  $\text{cm}^2/g$ . The constant values are specified using the appropriate `op_<spec><type>Constant` runtime parameter.
- “op.tabpa” Use the tabulated Planck Absorption opacity from the IONMIX or IONMIX4 file with name `op_<spec>FileName`.
- “op.tabpe” Use the tabulated Planck Emission opacity from the IONMIX or IONMIX4 file with name `op_<spec>FileName`.
- “op.tabro” Use the tabulated Rosseland opacity from the IONMIX or IONMIX4 file with name `op_<spec>FileName`.

When any of the models for a species is set to “op\_tabpa”, “op\_tabpe”, or “op\_tabro”, FLASH will attempt to read a file which contains the tabulated opacities. The file name, for each species, is given by the runtime parameter `op_<spec>FileName`. The type of the file is set using the parameter `op_<spec>FileType`. Currently, FLASH only supports reading opacity files in the IONMIX or IONMIX4 format, but this will likely change in the future.

The code segment below shows an example of how one would specify the runtime parameters for the Multispecies opacity in a simulation with two species named `cham` and `targ` (this simulation would have been set up using the `species=cham,targ` setup argument):

[illegible]

```

op_targTransConst = 1.0e+06      # Set targ species transport opacity to a large
                                # value Radiation will diffuse very slowly through
                                # this material

```

In this example species **cham** uses a tabulated opacity while species **targ** uses constant opacities. The opacities for **targ** are chosen so that the material will not emit any radiation, and to suppress the transport of radiation. The **LaserSlab** simulation shows a comprehensive example of how the Multispecies opacity is used in a realistic HEDP simulation.

### 23.5.6 The IONMIX EOS/Opacity Format

FLASH reads tabulated opacity and Equation Of State (EOS) files in the IONMIX, IONMIX4, and IONMIX6 formats. The IONMIX4 and IONMIX6 formats are very similar and are more flexible than the IONMIX format. Thus, only the IONMIX4 and IONMIX6 formats will be documented here. These three formats are not particularly user friendly and future FLASH releases will likely include support for EOS and opacity file formats which are easier to handle. Nevertheless, by manipulating their data into the IONMIX4 or IONMIX6 formats, users can use their own EOS/opacity data in FLASH simulations. Each file contains information for a single material. All EOS/opacity information is defined on a temperature/density grid. The densities are actually ion number densities.

Below, the IONMIX4 and IONMIX6 formats are defined. They are very similar: the latter is identical to the former but also includes electron specific entropy information. Certain information is ignored by FLASH, meaning that it is read in, but not used for any calculations (right now). Other information is used by the EOS unit alone, or by the opacity unit alone.

1. Number of temperature points
2. Number of ion number density points
3. Line containing information about the atomic number of each element in this material (ignored by FLASH)
4. Line containing information about the relative fraction of each element in this material (ignored by FLASH)
5. Number of radiation energy groups
6. List of temperatures
7. List of ion number densities
8. Average ionization ( $\bar{z}$ ) for each temperature/density (only used for tabulated EOS in FLASH)
9.  $d\bar{z}/dT_e$  [1/eV] for each temperature/density (ignored by FLASH)
10. Ion pressure ( $P_i$ ) [ $J/cm^3$ ] for each temperature/density (only used for tabulated EOS in FLASH)
11. Electron pressure ( $P_e$ ) [ $J/cm^3$ ] for each temperature/density (only used for tabulated EOS in FLASH)
12.  $dP_i/dT_i$  [ $J/cm^3/eV$ ] for each temperature/density (ignored by FLASH)
13.  $dP_e/dT_e$  [ $J/cm^3/eV$ ] for each temperature/density (ignored by FLASH)
14. Ion specific internal energy  $e_i$  [ $J/g$ ] for each temperature/density (only used for tabulated EOS in FLASH)
15. Electron specific internal energy  $e_e$  [ $J/g$ ] for each temperature/density (only used for tabulated EOS in FLASH)
16.  $de_i/dT_i$  [ $J/g/eV$ ] for each temperature/density point (ignored by FLASH)
17.  $de_e/dT_e$  [ $J/g/eV$ ] for each temperature/density point (ignored by FLASH)

18.  $de_i/dn_i$  [ $J/g/cm^3$ ] for each temperature/density point (ignored by FLASH)
19.  $de_e/dn_i$  [ $J/g/cm^3$ ] for each temperature/density point (ignored by FLASH)
20. **ONLY INCLUDE FOR IONMIX6 FORMAT**  
Electron specific entropy [ $J/g/eV$ ] for each temperature/density point.
21. Each energy group boundary [eV]. There are  $g + 1$  boundaries for a simulation with  $g$  groups.
22. Rosseland opacity [ $cm^2/g$ ] for each density/temperature/energy group (only used for tabulated opacity in FLASH)
23. Planck absorption opacity [ $cm^2/g$ ] for each density/temperature/energy group (only used for tabulated opacity in FLASH)
24. Planck emission opacity [ $cm^2/g$ ] for each density/temperature/energy group (only used for tabulated opacity in FLASH)

Below, the exact format of the IONMIX4 file is defined. This is done by providing code listing which shows how the data can be read in using FORTRAN read statements. Users can use this information to convert their own data into the IONMIX4 format so that it can be used by FLASH.

```
! *** Write the header ***
! Write the number of temperature and density points:
write (header,923)  ntemp, ndens

! Write the atomic number of each element in this material:
write (headr2,921) (izgas(1),l=1,ngases)

! Write the fraction (by number of ions) of each element in this material:
write (headr3,922) (fracsp(1),l=1,ngases)

write (10,980) header
write (10,980) headr2
write (10,980) headr3

! Write the number of radiation energy groups:
write (10,982) ngroups

! Write the temperature points (in eV):
write (10,991) (tplsma(it),it=1,ntemp)

! Write the number density points (in cm^-3):
write (10,991) (densn(id),id=1,ndens)

! Write out the zbar at each temperature/density point (nele/nion):
write (10,991) ((densne(it,id)/densn(id),it=1,ntemp),id=1,ndens)

! Write d(zbar)/dT:
write (10,991) ((dzdt(it,id),it=1,ntemp),id=1,ndens)

! Write the ion pressure (in Joules/cm^3):
write (10,991) ((densn(id)*tplsma(it)*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write the electron pressure (in Joules/cm^3):
write (10,991) ((tplsma(it)*densne(it,id)*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write out d(pion)/dT (in Joules/cm^3/eV):
write (10,991) ((densn(id)*1.602E-19_idk,it=1,ntemp),id=1,ndens)
```

```

! Write out d(pele)/dT (in Joules/cm^3/eV):
write (10,991) &
  (((tplsma(it)*densnn(id)*dzdt(it,id) + densne(it,id))*1.602E-19_idk,it=1,ntemp),id=1,ndens)

! Write out the ion specific internal energy (in Joules/gram):
write (10,991) ((enrgyion(it,id),it=1,ntemp),id=1,ndens)

! Write out the electron specific internal energy (in Joules/gram):
write (10,991) ((enrgy(it,id)-enrgyion(it,id),it=1,ntemp),id=1,ndens)

! Write out the ion specific heat (in Joules/gram/eV):
write (10,991) ((heatcpion(it,id),it=1,ntemp),id=1,ndens)

! Write out the electron specific heat (in Joules/gram/eV):
write (10,991) ((heatcp(it,id)-heatcpion(it,id),it=1,ntemp),id=1,ndens)

! Write out d(eion)/d(nion) (I think, but I'm not sure...)
write (10,991) ((dedden_ion(it,id)*condd*densnn(id)/tplsma(it)**2, it=1,ntemp),id=1,ndens)

! Write out d(eele)/d(nele) (I think, but I'm not sure...)
write (10,991) &
  (((dedden(it,id)-dedden_ion(it,id))*condd*densnn(id)/tplsma(it)**2, it=1,ntemp),id=1,ndens)

! ONLY FOR IONMIX6 FORMAT, IONMIX4 FORMAT DOES NOT INCLUDE ELECTRON
! SPECIFIC ENTROPY:
!
! Write out electron specific entropy
write (10,991) ((entrele(it,id),it=1,ntemp),id=1,ndens)

! Write out the energy group boundaries (in eV):
write (10,991) (engrup(ig),ig=1,ngrups+1)

! Write the Rosseland group opacities
write (10,991) (((orgp(it,id,ig),it=1,ntemp),id=1,ndens),ig=1,ngrups)

! Write the Planck absorption opacities
write (10,991) (((opgpa(it,id,ig),it=1,ntemp),id=1,ndens),ig=1,ngrups)

! Write the Planck emission opacities
write (10,991) (((opgpe(it,id,ig),it=1,ntemp),id=1,ndens),ig=1,ngrups)

921 format (' atomic #s of gases: ',5i10)
922 format (' relative fractions: ',1p5e10.2)
923 format (2i10)
980 format (a80)
981 format (4e12.6,i12)
982 format (i12)
991 format (4e12.6)

```

## 23.6 Mass Diffusivity

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



# Chapter 24

## Physics Utilities

### 24.1 PlasmaState

The `PlasmaState` unit was introduced in FLASH4.4 as a place to implement auxiliary routines that have knowledge of the behavior of plasmas. Currently, the `PlasmaState` unit contains routines for calculating the plasma parameter (Coulomb logarithm)  $\ln \Lambda$  (`PlasmaState_logLambda`), the collision time  $\tau$  (`PlasmaState_tau`), and the electron-ion equilibration time  $\tau_{ei}$  (`PlasmaState_ieEquilTime`). The Coulomb logarithm and collision time subroutines have options for computing the quantity for electron-electron, electron-ion, or ion-ion collisions. The value of  $\tau$  for electron-ion collisions is shown in (24.1).

$$\tau = \frac{3\sqrt{m_{\text{ele}}}(k_B T_{\text{ele}})^{3/2}}{4\sqrt{2}\pi e^4 \bar{z}^2 n_{\text{ion}} \ln \Lambda_{ei}}. \quad (24.1)$$

where:

- $k_B$  is the Boltzmann constant
- $e$  is the electron charge
- $m_{\text{ele}}$  is the mass of an electron
- $\bar{z}$  is the average ionization as computed by the EOS and subject to a floor value of 1.0
- $n_{\text{ion}}$  is the ion number density in the plasma
- $\ln \Lambda_{ei}$  is the Coulomb Logarithm associated with electron-ion collision, further description below.

Additional details on the electron-ion equilibration time and Coulomb logarithm can be found in Sec. 18.5.

These `PlasmaState` subroutines are used by various `materialProperties` units in calculating transport coefficients, by the `Heatexchange` unit, and in some cases by the `Hydro` unit. The `PlasmaState` unit may eventually include routines to compute other quantities of interest such as plasma frequencies, Hall parameters, etc.

There is also a routine named `PlasmaState_getComposition`, for computing the elemental composition of the mixture of materials in a cell (expressed as number fractions of elements). It is intended to be used with the `Multispecies` unit. Note that to get correct results, a simulation has to define the elemental compositions of the different species (materials) that may be present.



## Chapter 25

# Radiative Transfer Unit

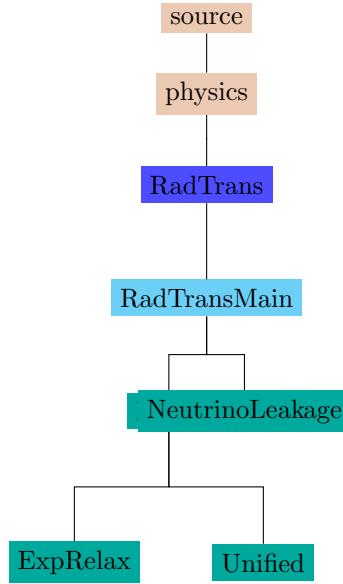


Figure 25.1: The organizational structure of the **RadTrans** unit.

The **RadTrans** unit is responsible for solving the radiative transfer equation

$$\frac{1}{c} \frac{\partial I}{\partial t} + \hat{\Omega} \cdot \nabla I + \rho \kappa I = \eta, \quad (25.1)$$

where,  $I(\mathbf{x}, \hat{\Omega}, \nu, t)$  is the radiation intensity,  $c$  is the speed of light,  $\rho$  is the mass density,  $\kappa(\mathbf{x}, \nu, t)$  is the opacity in units of  $\text{cm}^2/\text{g}$ ,  $\eta(\mathbf{x}, \nu, t)$  is the emissivity,  $\nu$  is the radiation frequency, and  $\hat{\Omega}$  is the unit direction vector. This equation is coupled to the electron internal energy through

$$\frac{\partial u_e}{\partial t} = \int_0^\infty d\nu \int_{4\pi} d\hat{\Omega} (\rho \kappa I - \eta), \quad (25.2)$$

where  $u_e$  is the electron internal energy density.

The **RadTrans** unit is responsible for solving the radiative transfer equation and updating the electron energy using (25.2) over a single time step. This ensures that the total system energy is conserved. Radiation-hydrodynamics effects, such as work, are operator-split and handled by the hydrodynamics unit. Currently, there is only a single **RadTrans** solver, **MGD**, which uses a simple multigroup diffusion approximation and is described in [Section 25.1](#).

## 25.1 Multigroup Diffusion

The radiative transfer (25.1) and electron energy (25.2) equations can be simplified using a multigroup diffusion approximation. The frequency space is divided into  $N_g$  groups with  $N_{g+1}$ . Group  $g$  is defined by the frequency range from  $\nu_g$  to  $\nu_{g+1}$ . The plasma is assumed to emit radiation in a Planck spectrum with a given emission opacity. The multigroup diffusion equations solved by FLASH are

$$\frac{1}{c} \frac{\partial u_g}{\partial t} - \nabla \cdot \left( \frac{1}{3\sigma_{t,g}} \nabla u_g \right) + \sigma_{a,g} u_g = \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} [P(x_{g+1}) - P(x_g)] \quad (25.3)$$

$$\frac{1}{c} \frac{\partial u_e}{\partial t} = \sum_g \left\{ \sigma_{a,g} u_g - \sigma_{e,g} a T_e^4 \frac{15}{\pi^4} [P(x_{g+1}) - P(x_g)] \right\} \quad (25.4)$$

where  $u_g$  is the radiation energy density,  $\sigma_{t,g}$  is the transport opacity,  $\sigma_{a,g}$  is the absorption opacity,  $\sigma_{e,g}$  is the emission opacity,  $a$  is the radiation constant,  $T_e$  is the electron temperature, and  $P(x)$  is the Planck integral, defined below. The argument to the Planck integral, is  $x = h\nu/k_B T_e$  where  $h$  is Planck's constant and  $k_B$  is the Boltzmann constant. The opacities in (25.3) and (25.4) are in units of  $1/\text{cm}$  and  $\sigma = \rho\kappa$ . The Planck integral,  $P(x)$  is defined as

$$P(x) = \int_0^x dx' \frac{(x')^3}{\exp(x') - 1} \quad (25.5)$$

The multigroup diffusion equation and electron internal energy equation are operator-split using an explicit treatment. Thus, on each time step (25.4) is solved for each energy group using coefficients evaluated at time level  $n$  using an implicit treatment for the diffusion solution itself. The equation below shows how the multigroup equations are discretized in time

$$\frac{1}{c} \frac{u_g^{n+1} - u_g^n}{\Delta t} - \nabla \cdot (D_g^n \nabla u_g^{n+1}) + \sigma_{a,g}^n u_g^{n+1} = \sigma_{e,g}^n a (T_e^n)^4 \frac{15}{\pi^4} [P(x_{g+1}^n) - P(x_g^n)] \quad (25.6)$$

$$\frac{u_e^{n+1} - u_e^n}{\Delta t} = \sum_g \left\{ \sigma_{a,g}^n u_g^{n+1} - \sigma_{e,g}^n a (T_e^n)^4 \frac{15}{\pi^4} [P(x_{g+1}^n) - P(x_g^n)] \right\} \quad (25.7)$$

where the  $n$  superscript denotes the time level, and  $\Delta t$  is the time step length. These equations are solved on each time step. The diffusion coefficient,  $D_g^n$ , has been introduced. When no flux-limiter is employed,  $D_g^n = 1/3\sigma_{t,g}^n$ . The flux-limiter limits the radiation flux in each group to the free streaming limit,  $c u_g^n$ . Several flux limiter options are available in FLASH. These are described in section 19.1.3. The string runtime parameter `rt.mgdFlMode` controls the flux limiter mode. The maximum flux is set to  $q_{\max} = \alpha_r c u_g^n$  for each energy group. The coefficient  $\alpha_r$  is set to one by default, which is what is physically most realistic. However, it can be controlled using the runtime parameter `rt.mgdFlCoef`.

The diffusion equation, (25.6), is an implicit equation and is solved using the general implicit diffusion solver described in section 19.1.2. It is recommended that users familiarize themselves with the general implicit diffusion solver to learn more about using MGD in FLASH.

The planck integral in previous versions has significant error and can be negative for  $x < 4 \times 10^{-4}$ . Set `rt.planckIntMethod` to 0 (by default) to use the method in old versions. Set `rt.planckIntMethod` to a positive number to use the new method: minimum of the  $m$ -th order expansion as  $\exp^{-x} \rightarrow 0$  and the  $n$ -th order expansion as  $x \rightarrow 0$ . Current available options: set to 213 for  $m = 2$  and  $n = 13$ , set to 321 for  $m = 3$  and  $n = 21$ , set to 621 for  $m = 6$  and  $n = 21$ .

### 25.1.1 Using Multigroup Radiation Diffusion

This section describes how to use MGD in FLASH. Section 35.7.5 shows an example of how to use MGD in a realistic HEDP simulation and is extremely informative.

Several setup options are needed to use MGD. The `+mgd` setup shortcut will include the MGD unit in the simulation. In addition, storage needs to be created for storing the specific energy for each group. This is done using the `mgd_meshgroups` setup variable. The user must set `mgd_meshgroups` such that:

$$(\text{mgd\_meshgroups}) N_{\text{mesh}} \geq N_g \quad (25.8)$$

where  $N_{\text{mesh}}$  represents the number of meshes (please see section 25.1.2 for more information). If you don't know what this is, you can assume that  $N_{\text{mesh}} = 1$ . In this case, the constraint is that the `mgd_meshgroups` setup variable must be greater than or equal to the number of energy groups. The actual number of energy groups,  $N_g$ , is set at run time using the runtime parameter `rt.mgdNumGroups`. Thus, as an example, the setup options for a 4-group simulation could be:

```
+mgd mgd_meshgroups=6
```

and the runtime parameter file must contain `rt.mgdNumGroups = 4`. No changes are needed to the `Config` file of the simulation.

The FLASH checkpoint files will contain the specific energy in each radiation energy group for each cell. The variables are named `r###` where `###` is a three digit number from one to  $N_g$ . Users can also add these variables to the list of plot variables so they will be included in plot files.

Several runtime parameters are used to control the MGD unit. The runtime parameter `rt.useMGD` must be set to `.true.` for MGD simulations. The energy group structure (values of the group boundaries) are specified, manually, using runtime parameters. A simulation with  $N_g$  groups will have  $N_g + 1$  energy group boundaries. These are set using the `rt.mgdBounds_###` runtime parameters, where `###` is a number between 1 and  $N_g + 1$ . By default, enough runtime parameters exist for specifying 101 energy group boundaries. In the case that more groups are needed, the `mgd_maxgroups` setup variable sets the number of group boundary runtime parameters. For example, if 200 energy groups are needed, then `mgd_maxgroups=200` should be specified on the setup line. If  $N_g \leq 100$ , no action is needed.

When MGD is in use, the `Opacity` unit provides opacities for each cell. Thus, the `useOpacity` runtime parameter must be set to true and the appropriate opacity model must be specified. Please see section 23.5 for a detailed description on how to use the `Opacity` unit in FLASH.

Finally, the radiation boundary conditions must be specified using runtime parameters. The parameters:

```
rt_mgdXlBoundaryType
rt_mgdXrBoundaryType
rt_mgdYlBoundaryType
rt_mgdYrBoundaryType
rt_mgdZlBoundaryType
rt_mgdZrBoundaryType
```

control the boundary condition on each logical face of the domain. The allowed values are:

- **reflecting, neumann**: This is a zero-flux, or Neumann boundary condition and should be used to model reflecting boundaries
- **dirichlet**: This (poorly named) value represents a fixed radiation temperature boundary condition
- **vacuum**: This represents a vacuum boundary condition

Please see section 19.1.2.1 for more information on boundary conditions for the diffusion solver.

As was mentioned earlier, FLASH uses the generalized implicit diffusion solver, described in section 19.1.2, to solve (25.6) for each energy group. Thus, users must make sure to include an appropriate diffusion solver in their simulation.

### 25.1.2 Using Mesh Replication with MGD

When many energy groups are needed, domain decomposition alone is not an effective method for speeding up calculations. The computation time scales roughly linearly with the number of energy groups. As the number of groups increases, users can divide the mesh among larger number of processors to compensate. This is an effective strategy until the strong scaling limit for the simulation is reached. At this point, mesh replication becomes the only way to speed up the simulation further.

Mesh replication allows FLASH to perform the diffusion solves for each energy group in parallel. By default, mesh replication is deactivated and each diffusion equation is solved serially. When mesh replication is in use the total pool of  $N_p$  processes is divided among  $N_m$  identical copies of the computational mesh.

The radiation solver uses these copies of the mesh to perform  $N_m$  parallel radiation solves. The runtime parameter `meshCopyCount` is used to set  $N_m$  and has a value of 1 by default.

This is best illustrated with an example. Suppose  $N_g$ , the total number of energy groups, is set to 100, and there are  $N_p = 6$  processes available to run the simulation. If `meshCopyCount=1`, then  $N_m = 1$  and the computational mesh will not be replicated. This is FLASH's normal mode of operation: the mesh will be divided into six pieces and each process will be responsible for one of these pieces. Each process will also be involved in the solution of the diffusion equation for all 100 energy groups. If `meshCopyCount=2`, then the computational domain will be divided into three pieces. Two processes will be assigned to each piece. In other words two groups of three processes will be created. The first set of three processes will be responsible for solving the diffusion equations for the odd numbered energy groups. The second set of processes will solve the diffusion equations for the even numbered energy groups. Each process only knows about 50 of the 100 groups. In some cases, this may be substantially faster than the `meshCopyCount=1` approach.

Another benefit of mesh replication is that it reduces the amount of memory needed per process. In the example above, when `meshCopyCount=1`, each process must store the energy density for all 100 energy groups. Thus, storage must exist for 100 cell centered variables on each process. When `meshCopyCount=2`, storage is only needed for 50 cell centered variables. The setup variable `mgd_meshgroups` controls the number of cell centered variables created on process. Thus, when `meshCopyCount=2`, we can set `mgd_meshgroups` to 50 instead of 100. As a result, far less memory is used per block.

Whether or not mesh replication is helpful for a particular simulation can only be determined through extensive testing. It depends on the particular simulation and the hardware.

### 25.1.3 Specifying Initial Conditions

The initial conditions must be specified throughout the domain for each radiation energy group. This involves setting the specific energy,  $e_g = u_g/\rho$ , in units of erg/g for each group. Because of the mesh replication capability, the user is discouraged from manually setting the value of the underlying radiation mass scalars. Instead, several routines have been added to the **RadTrans** unit to simplify this process dramatically. Specifying the initial conditions involves two steps. First, the value of  $e_g$  must be set for each group in each cell. Second, either the total specific radiation energy,  $e_r = \sum_g e_g$ , or the radiation temperature,  $T_r = (u_r/a)^{1/4}$ , must be set - depending on the value of `eosModeInit`. Below, two use cases are described.

#### 25.1.3.1 Initializing using a Radiation Temperature

The easiest way to set the initial condition is to use a radiation temperature in each cell and to assume that the radiation field is described by a Planck spectrum. The subroutine:

```
RadTrans_mgdEFromT(blockId, axis, trad, tradActual)
```

will automatically set the value of  $e_g$  for each group using a given radiation temperature and assuming a black body spectrum. The arguments are:

1. `blockId`: The block ID
2. `axis`: A 3 dimensional array of integers indicating the cell within the block
3. `trad`: An input, the desired radiation temperature
4. `tradActual`: An output, The actual radiation temperature (described further below)

When the simulation is initialized, it is important that  $T_r = (\sum_g u_g/a)^{1/4}$ . This will not always be the case. The reason is that a black body spectrum has a non zero energy density for all frequencies. But FLASH simulations require finite energy group boundaries. For example, if `trad` is set to 11604 K (this corresponds to 1 eV), but the upper energy group boundary is only set to 10 eV, then we are effectively cutting off the part of the spectrum above 10 eV. The *actual* radiation temperature, which is consistent with the sum of the energies in each group, will be slightly less than 1 eV. The argument `tradActual` contains this new radiation temperature.

Once `RadTrans_mgdEFromT` has been used to initialize  $e_g$ , either the total radiation specific energy,  $e_r$ , must be set or the radiation temperature for the cell must be specified. When the runtime parameter `eosModeInit` is set to “dens\_temp\_gather”, the initial radiation temperature must be specified in the variable `TRAD_VAR`. When `eosModeInit` is set to “dens\_ie\_gather”,  $e_r$  must be specified in the variable `ERAD_VAR`.

The example below shows a segment from a `Simulation_initBlock` routine from a typical simulation which uses MGD:

```
do k = blkLimits(LOW,KAXIS),blkLimits(HIGH,KAXIS)
  do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
    do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)

      axis(IAXIS) = i
      axis(JAXIS) = j
      axis(KAXIS) = k

      ...

      ! Set the secific energy in each radiation group using a
      ! radiation temperature of 1~eV (11604.55~K):
      call RadTrans_mgdEFromT(blockId, axis, 11604.55, tradActual)

      ! Set the radiation temperature:
      call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, tradActual)

      ! Alternatively, we could have set ERAD_VAR using a*(tradActual)**4

    enddo
  enddo
enddo
```

Note that `tradActual` is used to specify the value of `TRAD_VAR`.

### 25.1.3.2 Manually setting the radiation spectrum

Specifying the radiation temperature is sufficient for many cases. Alternatively, users can manually specify the radiation spectrum. The subroutine

`RadTrans_mgdSetEnergy(blockId, axis, group, eg)`

has been created for this purpose. The arguments are:

- `blockId`: The block ID
- `axis`: A 3 dimensional array of integers indicating the cell within the block
- `group`: The group number, between 1 and  $N_g$
- `eg`: The specific radiation energy (erg/g) for the group

This subroutine sets the value of  $e_g$  for a particular cell and should be called from the `Simulation_initBlock` subroutine.

The example below assumes that the user would like to initialize the radiation field so that  $e_1 = aT_r^4/\rho$  and all other groups are initialized to zero energy. Note, that for this example, the user should obtain the value of  $a$ , the radiation constant, from the `PhysicalConstants` unit. This simulations used  $N_g = 4$ .

```
do k = blkLimits(LOW,KAXIS),blkLimits(HIGH,KAXIS)
  do j = blkLimits(LOW,JAXIS),blkLimits(HIGH,JAXIS)
```

```

do i = blkLimits(LOW,IAXIS),blkLimits(HIGH,IAXIS)

    axis(IAXIS) = i
    axis(JAXIS) = j
    axis(KAXIS) = k

    ...

    ! Set the secific energy in each radiation group:
    call RadTrans_mgdSetEnergy(blockId, axis, 1, a*sim_trad**4/sim_rho)
    call RadTrans_mgdSetEnergy(blockId, axis, 2, 0.0)
    call RadTrans_mgdSetEnergy(blockId, axis, 3, 0.0)
    call RadTrans_mgdSetEnergy(blockId, axis, 4, 0.0)

    ! Set the radiation temperature:
    call Grid_putPointData(blockId, CENTER, TRAD_VAR, EXTERIOR, axis, sim_trad)

    ! Alternatively, we could have set ERAD_VAR using a*sim_trad**4/sim_rho

enddo
enddo
enddo

```

### 25.1.4 Altering the Radiation Spectrum

In some cases, it is necessary to alter the radiation spectrum manually, for example, to add a radiation energy source. In these cases, it is useful to manually alter the mesh variables which store the *specific* radiation energy within each group:  $e_g = u_g/\rho$ . However, because of mesh replication, altering these variables requires a different procedure than modifying other mesh variables (such as the density) in FLASH. The reason is that each process does not, in general, have access to each  $e_g$ . The procedure for modifying  $e_g$  directly is as follows:

1. Loop over the energy groups that each process sees
2. Modify  $e_g$
3. Modify the total specific radiation energy,  $e_{\text{rad}}$ , so that it stores the sum over all  $e_g$

Note, that in step 1, looping over the energy groups does not simply involve looping from 1 to  $N_g$ . The reason is that each process must loop over only those groups that it is responsible for. Again, because of mesh replication, this may be less than  $N_g$ . Below, an example is shown where the specific energy in each group is manually set to  $10^{12}$  erg/g. An ideal place for this type of code to exist is in the [Simulation-adjustEvolution](#) subroutine which gets called every time step.

```

! Loop over all cells, set the specific radiation energy to zero:
do lb = 1, nblk
    call Grid_getBlkIndexLimits(blk1st(lb),blkLimits,blkLimitsGC)
    call Grid_getBlkPtr(blk1st(lb), blkPtr)

    do k = blkLimits(LOW,KAXIS), blkLimits(HIGH,KAXIS)
        do j = blkLimits(LOW,JAXIS), blkLimits(HIGH,JAXIS)
            do i = blkLimits(LOW,IAXIS), blkLimits(HIGH,IAXIS)
                blkPtr(ERAD_VAR,i,j,k) = 0.0
            end do
        end do
    end do
end do

```



```

    call Grid_releaseBlkPtr(blklst(lb), blkPtr)
end do

! Loop over radiation energy groups:
do gloc = 1, NONREP_NLOCS(rt_acrossMe, rt_meshCopyCount, rt_mgdNumGroups)

    ! g represents the global energy group number:
    g = NONREP_LOC2GLOB(gloc, rt_acrossMe, rt_meshCopyCount)

    ! gvar represents the index in the block pointer for group g:
    gvar = MGDR_NONREP_LOC2UNK(gloc)

    ! Loop over blocks and set the energy density in each group:
    do lb = 1, nblk
        call Grid_getBlkIndexLimits(blklst(lb), blkLimits, blkLimitsGC)
        call Grid_getBlkPtr(blklst(lb), blkPtr)

        do k = blkLimits(LOW, KAXIS), blkLimits(HIGH, KAXIS)
            do j = blkLimits(LOW, JAXIS), blkLimits(HIGH, JAXIS)
                do i = blkLimits(LOW, IAXIS), blkLimits(HIGH, IAXIS)

                    ! Set the specific energy in group g:
                    blkPtr(gvar, i, j, k) = 1.0e+12

                    ! Make sure to track the total radiation energy in the cell:
                    blkPtr(ERAD_VAR, i, j, k) = blkPtr(ERAD_VAR, i, j, k) + &
                        blkPtr(gvar, i, j, k)

                end do
            end do
        end do
        call Grid_releaseBlkPtr(blklst(lb), blkPtr)
    end do
end do

! Finally, every process needs to know the total specific radiation
! energy. RadTrans_sumEnergy adds up ERAD_VAR across all of the
! meshes:
call RadTrans_sumEnergy(ERAD_VAR, nblk, blklst)

! Call EOS to get a consistent state:
do lb = 1, nblk
    call Grid_getBlkIndexLimits(blklst(lb), blkLimits, blkLimitsGC)
    call Eos_wrapped(MODE_DENS_EI_GATHER, blkLimits, blklst(lb))
end do

```

In the first part of the example, the value of `ERAD_VAR` is set to zero. `ERAD_VAR` simply stores the total specific radiation energy in each cell. After we are done modifying the energy for each group, it is important that `ERAD_VAR` be updated to contain the total specific radiation energy.

The second part of the example loops over the groups. Notice the strange loop ending index. The `NONREP_NLOCS` macro computes the number of groups represented on a given process. To find the actual group number,  $g$ , a call is made to `NONREP_LOC2GLOB`. Next, `MGDR_NONREP_LOC2UNK` is called to get the actual index into the block pointer which corresponds to group  $g$ . Following this is a fairly standard loop over blocks and cells where the block pointer is used to modify the specific energy in each group. Notice that we are keeping a running total of the total specific energy in the cell which is stored in `ERAD_VAR`.

Finally, we must call `RadTrans_sumEnergy` to add up `ERAD_VAR` across all of the meshes. After this

call, every process will contain the same total specific energy in each cell. This is followed by a call to the equation of state. This is needed to update the radiation temperature and pressure, `TRAD_VAR` and `ERAD_VAR`, respectively.

**Part VI**

**Monitor Units**



## Chapter 26

# Logfile Unit

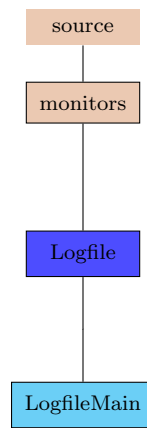


Figure 26.1: The **Logfile** unit directory structure.

FLASH supplies the **Logfile** unit to manage an output log during a FLASH simulation. The logfile contains various types of useful information, warnings, and error messages produced by a FLASH run. Other units can add information to the logfile through the **Logfile** unit interface. The **Logfile** routines enable a program to open and close a log file, write time or date stamps to the file, and write arbitrary messages to the file. The file is kept closed and is only opened for appending when information is to be written, thus avoiding problems with unflushed buffers. For this reason, **Logfile** routines should not be called within time-sensitive loops, as system calls are generated. Even when starting from scratch, the logfile is opened in append mode to avoid deleting important logfiles. Two kinds of Logfiles are supported. The first kind is similar to that in FLASH2 and early releases of FLASH3, where the master processor has exclusive access to the logfile and writes global information to it. The newer kind gives all processors access to their own private logfiles if they need to have one. Similar to the traditional logfile, the private logfiles are opened in append mode, and they are created the first time a processor writes to one. The private logfiles are extremely useful to gather information about failures caused by a small fraction of processors; something that cannot be done in the traditional logfile.

The **Logfile** unit is included by default in all the provided FLASH simulations because it is required by the **Driver/DriverMain Config**. As with all the other units in FLASH, the data specific to the Logfile unit is stored in the module **Logfile\_data.F90**. Logfile unit scope data variables begin with the prefix **log\_variableName** and they are initialized in the routine **Logfile\_init**.

By default, the logfile is named **flash.log** and found in the output directory. The user may change the name of the logfile by altering the runtime parameter **log\_file** in the **flash.par**.

# names of files

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

## 26.1 Meta Data

The `logfile` stores meta data about a given run including the time and date of the run, the number of MPI tasks, dimensionality, compiler flags and other information about the run. The snippet below is an example from a `logfile` showing the basic setup and compilation information:

```
=====
Number of MPI tasks:           2
MPI version:                   1
MPI subversion:                2
Dimensionality:                2
Max Number of Blocks/Proc:    1000
Number x zones:                8
Number y zones:                8
Number z zones:                1
Setup stamp:      Wed Apr 19 13:49:36 2006
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
=====
```

## 26.2 Runtime Parameters, Physical Constants, and Multispecies Data

The `logfile` also records which units were included in a simulation, the runtime parameters, physical constants, and any species and their properties from the `Multispecies` unit. The FLASH3 logfile keeps track of whether a runtime parameter is a default value or whether its value has been redefined in the `flash.par` file. The `[CHANGED]` symbol will occur next to a runtime parameter if its value has been redefined in the `flash.par`. Note that the runtime parameters are output in alphabetical order within the Fortran datatype – so integer parameters are shown first, then real, then string, then Boolean. The snippet below shows the this portion of the `logfile`; omitted sections are indicated with “...”.

```
=====
FLASH Units used:
Driver
Driver/DriverMain
Driver/DriverMain/TimeDep
Grid
Grid/GridMain
```

```

Grid/GridMain/paramesh
Grid/GridMain/paramesh/paramesh4
...
Multispecies
Particles
PhysicalConstants
PhysicalConstants/PhysicalConstantsMain
RuntimeParameters
RuntimeParameters/RuntimeParametersMain
...
physics/utilities/solvers/LinearAlgebra
=====
RuntimeParameters:

=====
algebra                =          2 [CHANGED]
bndpriorityone         =          1
bndprioritythree       =          3
...
cfl                    =          0.800E+00
checkpointfileintervaltime =      0.100E-08 [CHANGED]
cvisc                  =          0.100E+00
derefine_cutoff_1     =          0.200E+00
derefine_cutoff_2     =          0.200E+00
...
zmax                   =          0.128E+02 [CHANGED]
zmin                   =          0.000E+00
basenm                 = cellular_          [CHANGED]
eosmode                = dens_ie
eosmodeinit            = dens_ie
geometry               = cartesian
log_file               = cellular.log        [CHANGED]
output_directory       =
pc_unitsbase           = CGS
plot_grid_var_1        = none
plot_grid_var_10       = none
plot_grid_var_11       = none
plot_grid_var_12       = none
plot_grid_var_2        = none
...
yr_boundary_type       = periodic
zl_boundary_type       = periodic
zr_boundary_type       = periodic
bytepack               = F
chkguardcells          = F
convertttoconsvdformeshcalls = F
convertttoconsvdinmeshinterp = F
...
useburn                = T [CHANGED]
useburntable           = F

=====

Known units of measurement:

```

	Unit	CGS Value	Base Unit
1	cm	1.0000	cm
2	s	1.0000	s
3	K	1.0000	K
4	g	1.0000	g
5	esu	1.0000	esu
6	m	100.00	cm
7	km	0.10000E+06	cm
8	pc	0.30857E+19	cm

...

Known physical constants:

	Constant Name	Constant Value	cm	s	g	K	esu
1	Newton	0.66726E-07	3.00	-2.00	-1.00	0.00	0.00
2	speed of light	0.29979E+11	1.00	-1.00	0.00	0.00	0.00
...							
15	Euler	0.57722	0.00	0.00	0.00	0.00	0.00

=====

Multifluid database contents:

Initially defined values of species:

Name	Index	Total	Positive	Neutral	Negative	bind	Ener	Gamma
ar36	12	3.60E+01	1.80E+01	-9.99E+02	-9.99E+02	3.07E+02	-9.99E+02	
c12	13	1.20E+01	6.00E+00	-9.99E+02	-9.99E+02	9.22E+01	-9.99E+02	
ca40	14	4.00E+01	2.00E+01	-9.99E+02	-9.99E+02	3.42E+02	-9.99E+02	
...								
ti44	24	4.40E+01	2.20E+01	-9.99E+02	-9.99E+02	3.75E+02	-9.99E+02	

=====

## 26.3 Accessor Functions and Timestep Data

Other units within FLASH may make calls to write information, or stamp, the logfile. For example, the `Driver` unit calls the API routine `Logfile_stamp` after each timestep. The `Grid` unit calls `Logfile_stamp` whenever refinement occurs in an adaptive grid simulation. If there is an error that is caught in the code the API routine `Driver_abortFlash` stamps the logfile before aborting the code. Any unit can stamp the logfile with one of two routines `Logfile_stamp` which includes a data and time stamp along with a logfile message, or `Logfile_stampMessage` which simply writes a string to the logfile.

The routine `Logfile_stamp` is overloaded so the user must use the interface file `Logfile_interface.F90` in the calling routine. The next 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
...
[GRID amr_refine_derefine] initiating refinement
[GRID amr_refine_derefine] min blks 250    max blks 251    tot blks 501
[GRID amr_refine_derefine] min leaf blks 188    max leaf blks 188    tot leaf blks 376
[GRID amr_refine_derefine] refinement complete
```



```
[ 04-19-2006 16:40.44 ] [GRID gr_expandDomain]: create level=7
[ 04-19-2006 16:40.44 ] [GRID gr_expandDomain]: create level=7
[ 04-19-2006 16:40.44 ] [GRID gr_expandDomain]: create level=7
[ 04-19-2006 16:40.44 ] [IO_writeCheckpoint] open: type=checkpoint name=sod_hdf5_chk_0000
[ 04-19-2006 16:40.44 ] [io_writeData]: wrote      501      blocks
[ 04-19-2006 16:40.44 ] [IO_writeCheckpoint] close: type=checkpoint name=sod_hdf5_chk_0000
[ 04-19-2006 16:40.44 ] [IO_writePlotfile] open: type=plotfile name=sod_hdf5_plt_cnt_0000
[ 04-19-2006 16:40.44 ] [io_writeData]: wrote      501      blocks
[ 04-19-2006 16:40.44 ] [IO_writePlotfile] close: type=plotfile name=sod_hdf5_plt_cnt_0000
[ 04-19-2006 16:40.44 ] [Driver_evolveFlash]: Entering evolution loop
[ 04-19-2006 16:40.44 ] step: n=1 t=0.000000E+00 dt=1.000000E-10
...
[ 04-19-2006 16:41.06 ] [io_writeData]: wrote      501      blocks
[ 04-19-2006 16:41.06 ] [IO_writePlotfile] close: type=plotfile name=sod_hdf5_plt_cnt_0002
[ 04-19-2006 16:41.06 ] [Driver_evolveFlash]: Exiting evolution loop
=====
```

## 26.4 Performance Data

Finally, the logfile records performance data for the simulation. The `Timers` unit (see [Section 27.1](#)) is responsible for storing, collecting and interpreting the performance data. The `Timers` unit calls the API routine `Logfile_writeSummary` to format the performance data and write it to the logfile. The snippet below shows the performance data section of a logfile.

```
=====
perf_summary: code performance summary
              beginning : 04-19-2006 16:40.43
              ending   : 04-19-2006 16:41.06
seconds in monitoring period :      23.188
  number of subintervals :      21
  number of evolved zones :    16064
    zones per second :    692.758
=====
```

accounting unit	time sec	num calls	secs avg	time pct
initialization	1.012	1	1.012	4.366
guardcell internal	0.155	17	0.009	0.669
writeCheckpoint	0.085	1	0.085	0.365
writePlotfile	0.061	1	0.061	0.264
evolution	22.176	1	22.176	95.633
hydro	18.214	40	0.455	78.549
guardcell internal	2.603	80	0.033	11.227
sourceTerms	0.000	40	0.000	0.002
particles	0.000	40	0.000	0.001
Grid_updateRefinement	1.238	20	0.062	5.340
tree	1.126	10	0.113	4.856
guardcell tree	0.338	10	0.034	1.459
guardcell internal	0.338	10	0.034	1.458
markRefineDerefine	0.339	10	0.034	1.460
guardcell internal	0.053	10	0.005	0.230
amr_refine_derefine	0.003	10	0.000	0.011
updateData	0.002	10	0.000	0.009
guardcell	0.337	10	0.034	1.453
guardcell internal	0.337	10	0.034	1.452

eos	0.111	10	0.011	0.481
update particle refinemen	0.000	10	0.000	0.000
io	2.668	20	0.133	11.507
writeCheckpoint	0.201	2	0.101	0.868
writePlotfile	0.079	2	0.039	0.340
diagnostics	0.040	20	0.002	0.173

```
=====
[ 04-19-2006 16:41.06 ] LOGFILE_END: FLASH run complete.
```

## 26.5 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...", "[programtestLogfile]")
    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 27

# Timer and Profiler Units

### 27.1 Timers

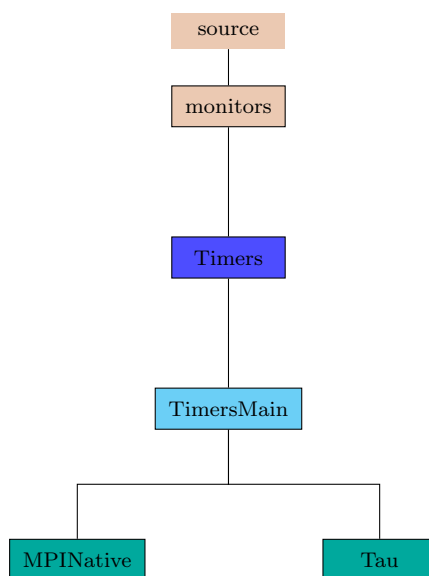


Figure 27.1: The **Timers** unit directory tree.

#### 27.1.1 MPINative

FLASH includes an interface to a set of stopwatch-like timing routines for monitoring performance. The interface is defined in the **monitors/Timers** unit, and an implementation that uses the timing functionality provided by MPI is provided in **monitors/Timers/TimersMain/MPINative**. Future implementations might use the PAPI framework to track hardware counter details.

The performance routines start or stop a timer at the beginning or end of a section of code to be monitored, and accumulate performance information in dynamically assigned accounting segments. The code also has an interface to write the timing summary to the FLASH logfile. These routines are not recommended for timing very short segments of code due to the overhead in accounting.

There are two ways of using the **Timers** routines in your code. One mode is to simply pass timer names as strings to the start and stop routines. In this first way, a timer with the given name will be created if it doesn't exist, or otherwise reference the one already in existence. The second mode of using the timers references them not by name but by an integer key. This technique offers potentially faster access if a timer

is to be started and stopped many times (although still not recommended because of the overhead). The integer key is obtained by calling with a string name `Timers.create` which will only create the timer if it doesn't exist and will return the integer key. This key can then be passed to the start and stop routines.

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

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

Table 27.1: Timer Unit runtime parameters.

Parameter	Type	Default value	Description
<code>eachProcWritesSummary</code>	LOGICAL	TRUE	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>
<code>writeStatSummary</code>	LOGICAL	TRUE	Should timers write the max/min/avg values for timers to the logfile?

`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 26.4](#) 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.

### 27.1.2 Tau

In `FLASH3.1` we add an alternative `Timers` implementation which is designed to be used with the `Tau` framework (<http://acts.nersc.gov/tau/>). Here, we use `Tau` API calls to time the `FLASH` labeled code sections (marked by `Timers.start` and `Timers.stop`). After running the simulation, the `Tau` profile contains timing information for both `FLASH` labeled code sections and all individual subroutines / functions. This is useful because fine grained subroutine / function level data can be overwhelming in a huge code like `FLASH`. Also, the callpaths are preserved, meaning we can see how long is spent in individual subroutines / functions when they are called from within a particular `FLASH` labeled code section. Another reason to use the `Tau` version is that the `MPINative` version (See [Section 27.1.1](#)) 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/TimersMain/Tau` as this `Timers` implementation is not the default.

## 27.2 Profiler

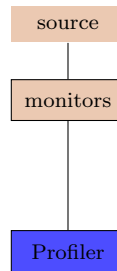


Figure 27.2: The `Profiler` unit directory tree.

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 FLASH4 we created an interface to the IBM profiling libraries `libmpihipm.a` and `libmpihipm_smp.a` and also to HPCToolkit <http://hpctoolkit.org/> (Rice University). We make use of this interface to profile FLASH evolution only, i.e. not initialization. To use this style of profiling add `-unit=monitors/Profiler/ProfilerMain/mpihipm` or `-unit=monitors/Profiler/ProfilerMain/hpctoolkit` to your setup line and also set the FLASH runtime parameter `profileEvolutionOnly = .true`.

For the IBM profiling library (`mpihipm`) you need to add `LIB_MPIHPM` and `LIB_MPIHPM_SMP` macros to your `Makefile.h` to link FLASH to the profiling libraries. The actual macro used in the link line depends on whether you setup FLASH with multithreading support (`LIB_MPIHPM` for MPI-only FLASH and `LIB_MPIHPM_SMP` for multithreaded FLASH). Example values from `sites/miralac1/Makefile.h` follow

```

LIB_MPI =
HPM_COUNTERS = /bgsys/drivers/ppcfloor/bgpm/lib/libbgpm.a
LIB_MPIHPM = -L/soft/perftools/hpctw -lmpihipm $(HPM_COUNTERS) $(LIB_MPI)
LIB_MPIHPM_SMP = -L/soft/perftools/hpctw -lmpihipm_smp $(HPM_COUNTERS) $(LIB_MPI)

```

For HPCToolkit you need to set the environmental variable `HPCRUN_DELAY_SAMPLING=1` at job launch to enable selective profiling (see the HPCToolkit user guide).



**Part VII**

**Diagnostic Units**





## Chapter 28

# Proton Imaging Unit

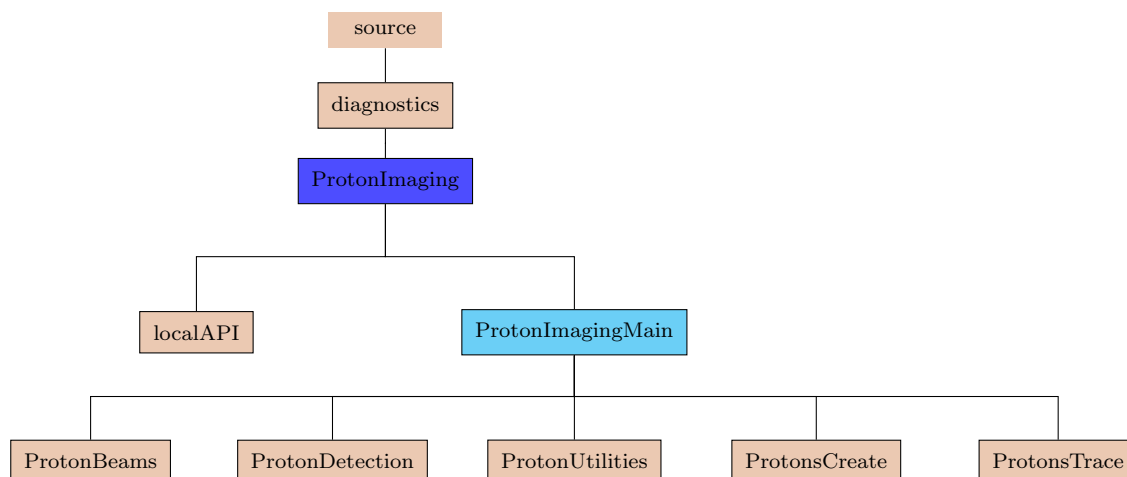


Figure 28.1: The `Proton Imaging` unit directory tree.

The `Proton Imaging` unit fires proton beams onto the simulation domain and records their deflection due to electric and magnetic fields on detector screens. The function `ProtonImaging` is the main driver that orchestrates the proton imaging execution. The deflections of the protons are calculated using the Lorentz force. For each cell in the domain the average electric and magnetic fields are used and the electric and magnetic components do not change within each cell. The code allows for setting up several proton beams with possibly different activation times and several detector screens. Each beam is associated with only one detector, but a detector is allowed to record the protons coming from more than one beam. This way one can simulate multienergetic proton beams by splitting them into several computational beams with identical spatial (isospacial) specifications and identical detector target but using different proton energies. Stacked film detectors for resolving different energy components can be simulated by associating separate isospacial detectors to each energy component of multienergetic proton beams. The mapping 'beam  $\rightarrow$  many detectors' is currently not allowed.

Tracing of the protons through the domain can be done in two ways: 1) crossing the domain during one time step or 2) allowing for time resolved proton tracing, i.e. tracing the protons through the domain during many time steps, if the velocities of the protons are sufficiently small. Although the proton imaging driver can be called during an entire simulation with proton beams being activated at certain simulation times, it is highly recommended to apply the proton imaging diagnostic as a post processing application on existing checkpoint files. Screen protons can be appended to existing detector files, but the proton imaging code does not write currently any data to the checkpoint files. Restarting a FLASH application from a previous proton imaging run is hence not recommended, especially if the time resolved proton imaging version is used. Old

disk protons waiting to be transported though the domain may be out of sync when restarting a previous run.

### 28.0.1 Proton Deflection by Lorentz Force

In the presence of an electric and magnetic field, the (relativistic and non-relativistic) Lorentz force in CGS units on a proton in motion is:

$$\mathbf{F} = Q(\mathbf{E} + \mathbf{v}/c \times \mathbf{B}), \quad (28.1)$$

where  $Q$  is the charge of the proton in esu ( $\text{g}^{1/2} \text{cm}^{3/2} \text{s}^{-1}$ ),  $\mathbf{F}$  the force in dyne ( $\text{g cm s}^{-2}$ ),  $\mathbf{v}$  the proton velocity ( $\text{cm s}^{-1}$ ),  $c$  the speed of light ( $2.998 \times 10^{10} \text{ cm s}^{-1}$ ) and  $\mathbf{E}$  and  $\mathbf{B}$  are the electric and magnetic fields in Gauss ( $\text{g}^{1/2} \text{cm}^{-1/2} \text{s}^{-1}$ ). The non-relativistic acceleration the proton experiences is:

$$\mathbf{a} = Q_m(\mathbf{E} + \mathbf{v}/c \times \mathbf{B}), \quad (28.2)$$

where  $Q_m$  is the mass rescaled charge  $Q/m$  of the proton. For uniform fields in each cell (constant  $(E_x, E_y, E_z)$  and  $(B_x, B_y, B_z)$ ), this leads to a set of 1st order coupled differential equations in time

$$\dot{v}_x(t) = Q_m[E_x + v_y(t)B_z/c - v_z(t)B_y/c] \quad (28.3)$$

$$\dot{v}_y(t) = Q_m[E_y + v_z(t)B_x/c - v_x(t)B_z/c] \quad (28.4)$$

$$\dot{v}_z(t) = Q_m[E_z + v_x(t)B_y/c - v_y(t)B_x/c] \quad (28.5)$$

for the velocity components  $v_x(t)$ ,  $v_y(t)$  and  $v_z(t)$ . Given the initial proton velocity components  $(v_{0x}, v_{0y}, v_{0z})$  at time  $t = 0$ , the analytical solution to this system of coupled differential equations becomes (assuming  $B \neq 0$ ):

$$\begin{aligned} v_x(t) = & (b_x^2 E_x + b_x b_y E_y + b_x b_z E_z) Q_m t \\ & + (b_x^2 + b_y^2 \cos[BQ_m t/c] + b_z^2 \cos[BQ_m t/c]) v_{0x} \\ & + b_z e_y (1 - \cos[BQ_m t/c]) c + b_y e_z (\cos[BQ_m t/c] - 1) c \\ & + b_x b_y (1 - \cos[BQ_m t/c]) v_{0y} + b_x b_z (1 - \cos[BQ_m t/c]) v_{0z} \\ & + (b_y^2 e_x + b_z^2 e_x - b_x b_y e_y - b_x b_z e_z) \sin[BQ_m t/c] c \\ & + (b_x^2 b_z + b_y^2 b_z + b_z^3) \sin[BQ_m t/c] v_{0y} \\ & - (b_x^2 b_y + b_z^2 b_y + b_y^3) \sin[BQ_m t/c] v_{0z} \end{aligned} \quad (28.6)$$

$$v_y(t) = [x \rightarrow y, y \rightarrow z, z \rightarrow x] v_x(t) \quad (28.7)$$

$$v_z(t) = [x \rightarrow z, y \rightarrow x, z \rightarrow y] v_x(t) \quad (28.8)$$

where  $B$  is the magnitude of the magnetic field vector and  $b_x = B_x/B$  and  $e_x = E_x/B$  are the magnetic field magnitude rescaled magnetic and electric field components. The operator  $[x \rightarrow y, y \rightarrow z, z \rightarrow x]$  stands for 'replace x with y, y with z and z with x' in the formula for  $v_x(t)$ . In case  $B = 0$ , these equations reduce to:  $v_x(t) = Q_m E_x t$ ,  $v_y(t) = Q_m E_y t$  and  $v_z(t) = Q_m E_z t$ . Note that the units of  $E_x Q_m t$  are the same as velocity. The quantities  $b_x$ ,  $e_x$  and  $BQ_m t/c$  are dimensionless. Integrating  $v_x(t)$ ,  $v_y(t)$  and  $v_z(t)$  over  $t$ , we get expressions for the positions  $r_x(t)$ ,  $r_y(t)$  and  $r_z(t)$  as a function of time. The resulting equations have terms involving  $t$ ,  $t^2$ ,  $\cos[BQ_m t/c]$  and  $\sin[BQ_m t/c]$ , and cannot be solved analytically. We therefore resort to a Runge-Kutta integration approach with the 6-dimensional ODE vector:

$$\frac{d}{dt} \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ \mathbf{a} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ Q_m(\mathbf{E} + \mathbf{v}/c \times \mathbf{B}) \end{pmatrix}, \quad (28.9)$$

For certain orientations of  $\mathbf{E}$  and  $\mathbf{B}$ , however, analytical solutions for  $\mathbf{r}$  and  $\mathbf{v}$  are possible (see the unit test section).

### 28.0.1.1 Relativistic Proton Equation of Motion

We state here briefly the relativistic proton equation of motion. Starting from the Lorentz equation 28.1, the force  $\mathbf{F}$  is the time derivative of the relativistic momentum  $\mathbf{p}$ :

$$\mathbf{F} = \frac{d\mathbf{p}}{dt} = \frac{d[\gamma(\mathbf{v})m\mathbf{v}]}{dt}, \quad (28.10)$$

where  $m$  is the rest mass of the proton and  $\gamma(\mathbf{v})$  is the Lorentz factor:

$$\gamma(\mathbf{v}) = 1/\sqrt{1 - (\mathbf{v} \cdot \mathbf{v}/c)^2} = 1/\sqrt{1 - (v/c)^2}. \quad (28.11)$$

Performing the differentiation on the r.h.s. of Eq.(28.10) using the chain rule (since both  $\mathbf{v}$  and thus  $\gamma$  depend on time), we arrive at:

$$\mathbf{F} = \gamma(\mathbf{v})m\mathbf{a}_\perp + \gamma(\mathbf{v})^3m\mathbf{a}_\parallel, \quad (28.12)$$

where  $\mathbf{a}_\parallel$  and  $\mathbf{a}_\perp$  are the parallel and perpendicular components of the acceleration  $\mathbf{a} = d\mathbf{v}/dt$  with respect to the velocity vector:

$$\mathbf{a} = \mathbf{a}_\parallel + \mathbf{a}_\perp, \quad \mathbf{v} \cdot \mathbf{a}_\perp = 0, \quad \mathbf{v} \cdot \mathbf{a} = \mathbf{v} \cdot \mathbf{a}_\parallel. \quad (28.13)$$

Inverting Eq.(28.12) to find the acceleration from the force on a moving proton leads to:

$$\mathbf{a} = \frac{1}{\gamma(\mathbf{v})m} \left( \mathbf{F} - \frac{[\mathbf{v} \cdot \mathbf{F}]\mathbf{v}}{c^2} \right) \quad (28.14)$$

with magnitude:

$$a = \frac{1}{\gamma(\mathbf{v})m} \sqrt{F^2 - \frac{1 + \gamma(\mathbf{v})^2}{c^2 \gamma(\mathbf{v})^2} [\mathbf{v} \cdot \mathbf{F}]^2}, \quad (28.15)$$

and inserting the Lorentz force equation 28.1 we get the relativistic expression for the acceleration:

$$\mathbf{a} = Q_{m\gamma} (\mathbf{E} + \mathbf{v}/c \times \mathbf{B} - [\mathbf{v}/c \cdot \mathbf{E}] \mathbf{v}/c), \quad (28.16)$$

where  $Q_{m\gamma}$  is the relativistic mass rescaled charge  $Q/m\gamma(\mathbf{v})$  of the proton. Comparing this expression with its non-relativistic version from Eq.(28.2), the main difference is not only the relativistic increase in mass ( $m\gamma(\mathbf{v})$ ) but also an extra term of order  $c^{-2}$  involving the electric field. The relativistic analytical velocity solutions corresponding to Eqs.(28.6-28.8) are much more complicated and involve extra logarithmic velocity component terms due to the extra relativistic velocity term.

### 28.0.1.2 Approximate Solutions to the Non-Relativistic Proton Equation of Motion

Since Runge-Kutta integration with constraints (due to the cell boundaries) is expensive, we wish to derive conditions under which the proton path through a cell approximates a parabola, for which quadratic solutions to the path positions become available. To this end we expand  $\mathbf{v}(t)$  in terms of  $t$  around the cell entry point  $t = 0$ . We get:

$$\mathbf{v}(t) = \mathbf{v}_0 + \left. \frac{d\mathbf{v}(t)}{dt} \right|_{t=0} t + \frac{1}{2} \left. \frac{d^2\mathbf{v}(t)}{dt^2} \right|_{t=0} t^2 + O[t^3] \quad (28.17)$$

$$= \mathbf{v}_0 + \mathbf{a}_0 t + \frac{\mathbf{j}_0}{2} t^2 + O[t^3], \quad (28.18)$$

where

$$\mathbf{a}_0 = Q_m(\mathbf{E} + \mathbf{v}_0/c \times \mathbf{B}) \quad (28.19)$$

$$\mathbf{j}_0 = Q_m(\mathbf{a}_0/c \times \mathbf{B}) \quad (28.20)$$

are the acceleration and jerk vectors at the cell entry point. Integration of Eq.(28.18) yields the path equation:

$$\mathbf{r}(t) = \mathbf{r}_0 + \mathbf{v}_0 t + \frac{\mathbf{a}_0}{2} t^2 + \frac{\mathbf{j}_0}{6} t^3 + O[t^4]. \quad (28.21)$$

The leading parabolic error term is hence the term involving the jerk vector. The largest parabolic error term  $\epsilon$  can be estimated by:

$$\epsilon = \frac{\max \mathbf{j}_0}{6} t^3. \quad (28.22)$$

If  $\epsilon$  is less than the allowed positional error of the computation, the parabolic approximation is used. To get also the velocity components with the same accuracy, the jerk terms are used to compute the velocities. If the jerk terms are omitted from the velocity calculations, the velocity components would only be accurate to first order in  $t$  and considerable error in velocity directions can accumulate during a proton imaging application.

In order to get a feeling for the conditions under which the parabolic approximation is applied in FLASH, let us first state the expression for the jerk vector  $\mathbf{j}_0$ , which is obtained by noting that  $\mathbf{j}_0 = d\mathbf{a}_0/dt$  and using the acceleration vector expression twice:

$$\mathbf{j}_0 = (Q_m^2/c)\mathbf{E} \times \mathbf{B} + (Q_m^2/c^2)(\mathbf{v}_0 \times \mathbf{B}) \times \mathbf{B}. \quad (28.23)$$

The maximum magnitude that the  $\mathbf{j}_0$  vector can achieve is given by the situation in which the pairs  $\mathbf{B}, \mathbf{v}_0$  and  $\mathbf{E}, \mathbf{B}$  are perpendicular and  $\mathbf{E} \times \mathbf{B}$  is opposite to  $\mathbf{v}_0$ . This leads to

$$|\mathbf{j}_0| \leq (Q_m^2/c)EB + (Q_m^2/c^2)B^2v_0 \quad (28.24)$$

and the inequality will also hold for the individual components of  $\mathbf{j}_0$ . Assuming no electric field, we can estimate the magnitude of  $B$  for which the parabolic approximation should be valid. In FLASH, the positional error is defined as a fraction  $f$  times the minimum cell side size. For a cubic cell with sides  $\Delta$  we can set up the parabolic condition on  $B$  as follows:

$$\frac{Q_m^2 B^2 v_0}{6c^2} t^3 \leq \Delta f. \quad (28.25)$$

The time it takes to cross the cubic cell is of the order of  $t = \Delta/v_0$  and we conservatively extend this time by the factor of  $\sqrt[3]{6}$  to get rid of the 6 in the denominator. The parabolic condition on  $B$  becomes

$$B \leq \frac{cv_0}{Q_m \Delta} \sqrt{f}. \quad (28.26)$$

For typical FLASH runs, we have for 20MeV protons  $v \approx c/5$  and  $\Delta$  is of the order of microns ( $10^{-4}$ cm) for typical simulations. The accuracy fraction  $f$  is typically  $10^{-6}$ . We also have  $Q_m \approx 2.9 \times 10^{14}$  and  $c = 2.99 \times 10^{10}$ . Hence under these conditions we have:

$$B \leq \approx 10^7 \text{ Gauss}, \quad (28.27)$$

i.e., all magnetic fields under  $10^7$  Gauss in a cell can be treated parabolically.

### 28.0.1.3 Approximate Solutions to the Relativistic Proton Equation of Motion

This follows along the lines of the previous section (28.0.1.2). Eq.(28.19) must be replaced by the relativistic expression from Eq.(28.16), replacing  $\mathbf{v}$  by  $\mathbf{v}_0$ , and Eq.(28.20) must be replaced by the time derivative of Eq.(28.16), replacing in the final expression  $\mathbf{v}$  by  $\mathbf{v}_0$  and  $\mathbf{a}$  by  $\mathbf{a}_0$ :

$$\mathbf{a}_0 = Q_{m\gamma} (\mathbf{E} + \mathbf{v}_0/c \times \mathbf{B} - [\mathbf{v}_0/c \cdot \mathbf{E}] \mathbf{v}_0/c), \quad (28.28)$$

$$\mathbf{j}_0 = Q_{m\gamma} (\mathbf{a}_0/c \times \mathbf{B} - [\mathbf{a}_0/c \cdot \mathbf{E}] \mathbf{v}_0/c - 2[\mathbf{v}_0/c \cdot \mathbf{E}] \mathbf{a}_0/c), \quad (28.29)$$

where  $Q_{m\gamma}$  is equal to  $Q/m\gamma(\mathbf{v}_0)$ . Note the extra terms of order  $c^{-2}$  when compared with Eqs.(28.19) and (28.20). The largest parabolic error expression in Eq.(28.22) remains the same. Assuming again no electric field, the same steps leading to the non-relativistic parabolic  $B$  condition in Eq.(28.26), leads to:

$$B \leq \frac{cv_0}{Q_{m\gamma}\Delta} \sqrt{f}, \quad (28.30)$$

the only difference being in using the relativistic mass rescaled proton charge. For the 20MeV protons  $v \approx c/5$  this would lead to a factor of  $\approx 1.02$  larger  $B$  than in Eq.(28.27), due to the relativistically reduced magnitude of  $\mathbf{j}_0$ .

## 28.0.2 Setting up the Proton Beam

The setup of each proton beam follows closely the setup of the laser beams in 18.4.6. The major difference is that the proton imaging beams originate from 3D capsules instead of 2D planar cross-section areas. This allows for simulating proton capsule aberrations in the proton beams. The radius of the capsule as well as its center location is specified for each beam at runtime, as well as each beam's full conical aperture angle. Target coordinates are only needed for directional purposes and the target area is circular and perpendicular to the beam's direction. Additional runtime parameters needed for each beam are: 1) the launching time (the beam fires once the simulation time exceeds the launching time), 2) its target detector screen and 3) the number of protons to be launched and their energies. Once each beam has been set up, it is checked, if its capsule volume is completely outside of the computational domain. A beam capsule (partially) inside the domain is not allowed. A local set of 3D rectangular grid unit vectors  $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$  is constructed for each beam, such that  $\mathbf{u}_3$  points from the capsule center to the target center and  $\mathbf{u}_1 \times \mathbf{u}_2 = \mathbf{u}_3$ . The 3D grid unit vectors serve for generating statistical 3D points inside the spherical capsule as well as statistical 2D points inside the circular target area.

## 28.0.3 Creating the Protons

The protons inside each beam are created by connecting one-to-one the statistical 3D grid points inside the capsule with the statistical 2D grid points inside the circular target area. The proton directions inside each beam are therefore not conically collinear and are allowed to cross. The proton initial positions and velocities on the domain surface are calculated using the same strategy as presented in 18.4.7.6. Protons missing the domain can either be simply ignored or directly recorded on the detector screen.

### 28.0.3.1 Moving Protons through the Domain

As the total number of protons generated can be very large, special techniques have to be used to overcome the computational memory constraints. The Proton Imaging unit has been designed to work with a proton batch and screen proton bucket combination. Both the batch and the bucket are of much smaller size than the total protons and screen protons generated and can be seen as temporary storage devices for both entities. During a time step, the active proton beams start filling the generated protons into the beam proton batch. Once filled, the beam protons in the batch are sent through the domain, where they are either collected as disk protons into the disk proton batch (if time resolved proton imaging was requested) or as screen protons into the screen proton bucket. If the screen proton bucket is full, its contents (screen protons) are written to the corresponding detector files and emptied. The refilling of both the beam/disk proton batch and the screen bucket are independent of each other and both batch and bucket can be of different sizes. The main function `ProtonImaging` processes beam/disk proton batch after batch until no more active beam/disk protons are present. As an example, if at a time step we have 3 active beams with 1,000,000 protons each and the dimension of the proton batch is set to 100,000, a total of 30 beam proton batches will be generated and sent through the domain.

### 28.0.3.2 Proton Specifications

Each beam/disk proton needs to know its position  $x, y, z$  and velocity  $v_x, v_y, v_z$  as it traverses the domain. In addition to these 6 components it needs to know: 1) the time spent travelling in domain, 2) the current

block and processor numbers, 3) a global identification tag, 4) the originating beam and target detector numbers. Screen protons on the other hand are only defined on the detector screen and hence need no specific information for domain travelling. Their information is reduced to a screen position coordinate pair  $x, y$  and a detector number to which detector they are associated with.

Additionally, upon request, the proton imaging code is able to calculate extra magnetic diagnostic quantities  $\mathbf{k}$  and  $J$  for each beam/disk and screen proton:

$$\mathbf{k} = \int \mathbf{B} d\ell, \quad (28.31)$$

$$J = \int \mathbf{v}_n \cdot (\nabla \times \mathbf{B}) d\ell, \quad (28.32)$$

where  $\int d\ell$  denotes path integration,  $\mathbf{B}$  is the magnetic field vector and  $\mathbf{v}_n$  is the unit normal velocity vector along the proton path. Transforming the path integrals into time integrals using the relation  $d\ell = |\mathbf{v}(t)|dt$ , we get the differentials:

$$\frac{d\mathbf{k}}{dt} = \mathbf{B}|\mathbf{v}(t)|, \quad (28.33)$$

$$\frac{dJ}{dt} = \mathbf{v}(t) \cdot (\nabla \times \mathbf{B}). \quad (28.34)$$

leading to extra 4 entries in the Runge-Kutta vector in Eq.(28.9):

$$\frac{d}{dt} \begin{pmatrix} \mathbf{r} \\ \mathbf{v} \\ \mathbf{k} \\ J \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ Q_m(\mathbf{E} + \mathbf{v}/c \times \mathbf{B}) \\ \mathbf{B}v \\ \mathbf{v} \cdot (\nabla \times \mathbf{B}) \end{pmatrix}, \quad (28.35)$$

If the parabolic path approximation is used for a cell, then  $\mathbf{v}(t) = \mathbf{v}_0 + \mathbf{a}_0 t$ , where  $\mathbf{v}_0$  and  $\mathbf{a}_0$  are the velocity and acceleration at cell entry. For a  $J$  cell contribution we have (note this is a general result not depending on the parabolic approximation):

$$J = \int_0^T \mathbf{v}(t) \cdot (\nabla \times \mathbf{B}) dt = \int_0^T d\mathbf{r}(t) \cdot (\nabla \times \mathbf{B}) = [\mathbf{r}(T) - \mathbf{r}_0] \cdot (\nabla \times \mathbf{B}) \quad (28.36)$$

where  $T$  is the cell crossing time and  $\mathbf{r}_0, \mathbf{r}(T)$  are the cell entry and exit locations of the proton. For the parabolic  $\mathbf{k}$  cell contribution we have;

$$\mathbf{k} = \mathbf{B} \int_0^T |\mathbf{v}_0 + \mathbf{a}_0 t| dt. \quad (28.37)$$

The parabolic path length integral can be solved exactly:

$$\begin{aligned} \int_0^T |\mathbf{v}_0 + \mathbf{a}_0 t| dt &= \frac{1}{2} \left[ |\mathbf{a}_0 T + \mathbf{v}_0| T + \frac{\mathbf{a}_0 \cdot \mathbf{v}_0}{|\mathbf{a}_0|^2} (|\mathbf{a}_0 T + \mathbf{v}_0| - |\mathbf{v}_0|) \right. \\ &\quad \left. + \frac{|\mathbf{a}_0 \times \mathbf{v}_0|^2}{|\mathbf{a}_0|^3} \ln \left( \frac{\mathbf{a}_0 \cdot (\mathbf{a}_0 T + \mathbf{v}_0) + |\mathbf{a}_0| |\mathbf{a}_0 T + \mathbf{v}_0|}{\mathbf{a}_0 \cdot \mathbf{v}_0 + |\mathbf{a}_0| |\mathbf{v}_0|} \right) \right]. \end{aligned} \quad (28.38)$$

The second term is the offperpendicular and the third term the offcolinear contribution to the path length. Taylor series expansion of this expression around  $\mathbf{v}_0$  in terms of the components of  $\mathbf{a}_0$  gives the zeroth order term as  $|\mathbf{v}_0|T$  corresponding to the path length for  $\mathbf{a}_0 = \mathbf{0}$ . Likewise the zeroth order term for the expansion around  $\mathbf{a}_0$  in terms of the components of  $\mathbf{v}_0$  is  $|\mathbf{a}_0|T^2/2$  which is the path length for  $\mathbf{v}_0 = \mathbf{0}$ . Application of the above general formula must be done with some care to avoid computational pitfalls. The cases where  $\mathbf{v}_0$  and  $\mathbf{a}_0$  are colinear must be handled separately. It is also easy to show, for example, that for the offcolinear cases the natural logarithm argument is always  $\geq 1$ .

## 28.0.4 Setting up the Detector Screens

Each detector screen is defined as a square planar area with a specific side length  $s$  and an optional circular pinhole. The orientation of the screen in 3D space is specified by three components: 1) the center location of the screen  $\mathbf{C}$ , 2) the normal vector of the screen plane  $\mathbf{n}$  and 3) the orthogonal unit vector pair  $\mathbf{u}_x, \mathbf{u}_y$  with origin at the center and oriented in such a way that each unit vector is parallel to two opposite sides of the detector screen. Location of the pinhole  $\mathbf{H}$  is fixed by giving the distance between the two pinhole and detector centers and is always located on the detector side opposite to  $\mathbf{n}$ . The circular pinhole area is coplanar to the detector screen area. While  $\mathbf{C}$  and  $\mathbf{n}$  are sufficient to characterize the screen plane, the exact location of the detector square area is only specified once the orientation of  $\mathbf{u}_y$  (or  $\mathbf{u}_x$ ) is fixed. For that purpose we define a tilting angle  $\alpha$  of  $\mathbf{u}_y$  wrt one of the global 3D axes along the normal vector  $\mathbf{n}$ . A tilting angle of  $\alpha = 0$  would thus mean that the chosen global axis and  $\mathbf{u}_y$  are coplanar.

### 28.0.4.1 Recording Protons on the Detector Screen

After each proton leaves the domain, it is located on the domain surface at a position  $\mathbf{P}$  and with (outward from domain) velocity  $\mathbf{v}$ . The goal is to see, if the proton actually hits the screen plane and to record the local coordinates  $(x, y)$  on the screen plane with coordinate basis  $[\mathbf{u}_x, \mathbf{u}_y]$ . After some algebra we arrive at the following conditions

$$\mathbf{v} \cdot \mathbf{n} = 0 \quad \text{proton moves parallel to screen plane} \quad (28.39)$$

$$\frac{(\mathbf{C} - \mathbf{P}) \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} > 0 \quad \text{proton hits screen plane} \quad (28.40)$$

and the following expressions for the local coordinates

$$\begin{aligned} x &= \frac{\mathbf{u}_y \cdot [\mathbf{v} \times (\mathbf{C} - \mathbf{P})]}{\mathbf{v} \cdot \mathbf{n}} \\ y &= \frac{\mathbf{u}_x \cdot [(\mathbf{C} - \mathbf{P}) \times \mathbf{v}]}{\mathbf{v} \cdot \mathbf{n}}. \end{aligned} \quad (28.41)$$

For better handling of the screen output data, the local coordinates are shifted and rescaled such that  $(x, y) \in [0, 1]$  when the proton is on the detector screen. Using the detectors side length  $s$ , we achieve this by the following local coordinate transformation:

$$x \longrightarrow (x + s/2)/s \quad (28.42)$$

$$y \longrightarrow (y + s/2)/s. \quad (28.43)$$

If a pinhole is present, the code has to check, if the proton makes it through the hole. Since the detector screen area and the circular pinhole area are coplaner, we can use the local coordinate basis  $[\mathbf{u}_x, \mathbf{u}_y]$  located at the pinhole center  $\mathbf{H}$  to calculate a local pinhole coordinate for each proton, in analogy to Eq.(28.41):

$$h_x = \frac{\mathbf{u}_y \cdot [\mathbf{v} \times (\mathbf{H} - \mathbf{P})]}{\mathbf{v} \cdot \mathbf{n}} \quad (28.44)$$

$$h_y = \frac{\mathbf{u}_x \cdot [(\mathbf{H} - \mathbf{P}) \times \mathbf{v}]}{\mathbf{v} \cdot \mathbf{n}}. \quad (28.45)$$

The proton makes it through the pinhole, if  $h_x^2 + h_y^2 \leq r_h^2$ , where  $r_h$  is the radius of the pinhole.

The user is able to specify via a runtime parameter, if he wants only the detector screen protons  $(x, y) \in [0, 1]$  to be written to the detector file or if he also wants to record the offscreen protons  $(x, y) \notin [0, 1]$ . Each detector file is currently an ascii file (formatted) and named:

$$\langle \text{basename} \rangle \text{ProtonDetectorFile} \langle \text{detectorID} \rangle \_ \langle \text{timestamp} \rangle \quad (28.46)$$

where  $\langle \text{basename} \rangle$  is the name of the simulation,  $\langle \text{detectorID} \rangle$  is the detector number (currently limited to the range [01-99]) and (optionally)  $\langle \text{timestamp} \rangle$  is the simulation time when the file was created. Only the  $(x, y)$  pairs are written (first  $x$  followed by  $y$  on each line) without any headers. Thus the file can be directly incorporated and read by other software for graphical output (see for example the proton imaging ASCII  $\rightarrow$  PGM greyscale converter). Additionally, if requested, the detector file will contain in columns 3 to 6 the proton magnetic diagnostic quantities  $J, k_x, k_y, k_z$  from Eqs.(28.31) and (28.32), in that order.

### 28.0.5 Time Resolved Proton Imaging

For some applications (slow moving protons, fast domain changing) it is desirable to adjust the domain environment as the protons move through it. The proton imaging unit has been given this capability by monitoring the time each proton spends in the domain and compare it to the computational time step. When the proton time exceeds the time step value, it is stored as a disk proton which will be written to disk. All disk protons from a previous time step must be first processed during a new time step before any new beam protons are generated. During a time step, the main driver **ProtonImaging** calls the following two tasks, each of which does the following operations:

#### Transport disk protons:

- Read all disk protons from old disk proton file
- Trace all disk protons through domain:
  - To new disk proton file (if proton time > time step)
  - To screen detector
- Rename new disk proton file → old disk proton file

#### Transport beam protons:

- Create all beam protons
- Trace all beam protons through domain:
  - To new disk proton file (if proton time > time step)
  - To screen detector
- Append new disk proton file → old disk proton file

The time resolved application requires extra memory and disk storage due to the presence of disk protons as well as extra reading/writing time from/to disk. If only a non-time resolved proton imaging run is requested, the code skips all array allocations and operations associated with the disk protons, and only the underlined actions of the transport beam protons task get activated.

### 28.0.6 Usage

To include the use of the Proton Imaging unit, the following should be included into the setup line command:

```
+protonImaging [pi_maxBeams=<number> pi_maxDetectors=<number> threadProtonTrace=True]
```

The +protonImaging shortcut handles all the logistics for properly including the **ProtonImaging** unit. The default settings are: maximum number of beams = 1, maximum number of detectors = 1 and no use of threading during tracing of protons through the domain, which can be changed by the following three setup variables:

- **pi\_maxBeams**: The maximum number proton beams for the simulation.
- **pi\_maxDetectors**: The maximum number of proton imaging detectors for the simulation.
- **threadProtonTrace**: Enables threading during proton domain tracing.

Using proton imaging runtime parameters in the flash.par file, the user can set up the proper proton imaging configuration for his specific needs. The following list of runtime parameters is currently available for the user.



### 28.0.6.1 Proton Imaging Beams Runtime Parameters

The following are the runtime parameters for the proton imaging beams. The `_n` at the end of each runtime parameter characterizes the beam number, hence replace `_n` by `_1` for the first beam, `_n` by `_2` for the second beam, etc.

- `pi_numberOfBeams`: The number of proton imaging beams that are going to be used.
- `pi_beamCapsule[X,Y,Z]_n`: The global 3D components of the capsule center position vector  $\mathbf{C}$ .
- `pi_beamCapsuleRadius_n`: The radius of the spherical capsule.
- `pi_beamTarget[X,Y,Z]_n`: The global 3D components of the target position vector  $\mathbf{T}$  for directional purpose.
- `pi_beamApertureAngle_n`: The conical full aperture angle.
- `pi_beamProtonEnergy_n`: The energy (in MeV) of the protons in the beam.
- `pi_beamTime2Launch_n`: The time during the simulation, when the beam should be fired.
- `pi_beamDetector_n`: The detector number, where the protons of the beam should be recorded.
- `pi_beamNumberOfProtons_n`: Number of protons to be launched from the beam.
- `pi_beamNoBoundaryCondition_n`: Ignore domain boundary conditions (reflective) when protons enter the domain?

### 28.0.6.2 Proton Imaging Detectors Runtime Parameters

The following are the runtime parameters for the proton imaging detectors. As for the beams, the `_n` at the end of each runtime parameter characterizes the detector number.

- `pi_numberOfDetectors`: The number of proton imaging detectors that are going to be used.
- `pi_detectorCenter[X,Y,Z]_n`: The global 3D components of the detector center position vector  $\mathbf{C}$ .
- `pi_detectorNormal[X,Y,Z]_n`: The local 3D components of the detector normal vector  $\mathbf{n}$ .
- `pi_detectorSideLength_n`: The side length (cm) of the detector square screen.
- `pi_detectorSideTiltingAngle_n`: The tilting angle (degrees) of two sides parallel to the detector screen unit axis  $\mathbf{u}_y$  with respect to the tilting axis.
- `pi_detectorSideTiltingAxis_n`: The global tilting axis ('x','y' or 'z') for the detector screen.
- `pi_detectorAlignWRTbeamNr_n`: A useful shortcut to place the detector screen right along a beam path with the screen surface orthogonal to the beam path. This, together with the specified detector distance from the beam capsule center, automatically calculates the center vector  $\mathbf{C}$  and the normal vector  $\mathbf{n}$ . If a beam number  $\leq 0$  is specified, no alignment is performed and the user has to supply center vector and normal vector coordinates.
- `pi_detectorDist2BeamCapsule_n`: The detector distance from the beam capsule center, used only if the detector is aligned with respect to a beam.
- `pi_detectorPinholeRadius_n`: The radius of the pinhole. If  $\leq 0$  no pinhole is used.
- `pi_detectorPinholeDist2Det_n`: The pinhole center distance from the detector center, useful only if a pinhole radius  $> 0$  was specified.

### 28.0.6.3 Proton Imaging General Runtime Parameters

- **pi\_cellStepTolerance**: This factor times the smallest dimension of each cell is taken as the positional error tolerance for the protons during their path (parabolic or Runge-Kutta) tracing through each cell.
- **pi\_cellWallThicknessFactor**: Controls the (imaginary) thickness of the cell walls to ensure computational stability of the proton imaging code. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to  $10^{-6}$  and should only very rarely be changed.
- **pi\_detectorFileNameTimeStamp**: If set to true, a time stamp will be added to each detector file name. This allows for time splitting of detectors.
- **pi\_detectorXYwriteFormat**: Controls formatted ascii output of the proton  $(x, y)$  pairs on the detector screens (default 'es20.10').
- **pi\_detectorDGwriteFormat**: Controls formatted ascii output of the magnetic diagnostic quantities  $k_x, k_y, k_z, J$  on the detector screens (default 'es15.5').
- **pi\_flagDomainMissingProtons**: If true, any proton missing the domain will abort the program. If false, protons missing the domain will be recorded directly on the detector screen.
- **pi\_ignoreElectricalField**: If true, the electrical field in each cell is ignored and the proton movement is only governed by the magnetic field (deflection only).
- **pi\_IOaddDetectorScreens**: If true, the square frame of each detector will be added to the plotfile.
- **pi\_IOaddProtonsCapsule2Domain**: If true, the proton paths from the capsule to the domain entry point will be added to the plotfile.
- **pi\_IOaddProtonsDomain2Screen**: If true, the proton paths from the domain exit point to the detector screen will be added to the plotfile.
- **pi\_IOmaxBlockCrossingNumber**: This is an estimate of the maximum proton path length while traveling through the block. This estimate is given as an integer number in units of block sides (default 5). Only in extremely rare cases, when the magnetic and electric fields are strong and sufficiently warped inside a block and the protons start to circulate inside the block, does one have to increase this number.
- **pi\_IOnumberOfProtons2Plot**: The number of protons that are to be plotted and written to the plotfile.
- **pi\_maxProtonCount**: The maximum number of beam/disk protons that can be created/read-in on one processor per batch in the domain.
- **pi\_opaqueBoundaries**: If true, the protons do not go through cells marked as opaque boundaries.
- **pi\_printBeams**: If true, it prints detailed information about the proton beams to a file with name <basename>ProtonBeamsPrint.txt, where <basename> is the base name of the simulation.
- **pi\_printDetectors**: If true, it prints detailed information about the proton detectors to a file with name <basename>ProtonImagingDetectors.txt, where <basename> is the base name of the simulation.
- **pi\_printMain**: If true, it prints general information regarding the proton imaging setup to a file with name <basename>ProtonImagingMainPrint.txt, where <basename> is the base name of the simulation.
- **pi\_printProtons**: If true, it prints detailed information about the all protons generated during the simulation. Protons are 'generated' on the domain surface and the info is written to several files labeled by batch number BID, processor rank number PID and a time stamp. Each processor writes its own file(s) with name(s): <basename>printProtonsBatch<BID>Proc<PID>.txt, where <basename> is

the base name of the simulation. Use of this feature is reserved ONLY for debugging purposes and is currently limited to 10 batches and 100 processors per time stamp. Usage of a larger number of batches/processors during a simulation does not abort the run, but protons on batches with  $BID > 10$  and processors with  $PID > 99$  are simply ignored and not printed. Users other than code developers should not activate this feature.

- **pi.protonDeterminism**: If true, the Grid Unit will be forced to use the sieve algorithm to move the proton particle data. Forcing this algorithm will result in a slower movement of data, but will fix the order the processors pass data and eliminate round off differences in consecutive runs.
- **pi.recordOffScreenProtons**: If true, protons not hitting their target detector screen are also recorded on the detector files. These protons will have screen coordinate pairs  $(x, y) \notin [0, 1]$ .
- **pi.relativisticTracing**: If true, the motion of the protons through the domain will be treated relativistically using Eq.(28.16) for the relativistic acceleration and Eq.(28.29) for the relativistic jerk terms.
- **pi.RungeKuttaMethod**: Specifies which Runge Kutta method to use for proton tracing. Current options are: 'CashKarp45' (order 4, default), 'EulerHeu12' (order 1), 'BogShamp23' (order 2), 'Fehlberg34' (order 3) and 'Fehlberg45' (order 4).
- **pi.screenProtonBucketSize**: Sets the bucket size for flushing screen protons out to disk.
- **pi.screenProtonDiagnostics**: If true, the magnetic diagnostic quantities  $k_x, k_y, k_z, J$  are evaluated for each proton and recorded on the detector screens.
- **pi.timeResolvedProtonImaging**: If true, it activates the time resolved proton imaging part of the code. Protons might need several time steps to cross the domain.
- **pi.useIOprotonPlot**: If true, protons are plotted to the plotfile for visualization purposes.
- **pi.useParabolicApproximation**: If true, the code traces protons parabolically through cells for low  $B$  / high  $v$  combinations (section 28.0.1.2).
- **useProtonImaging**: If false, no proton imaging will be performed, even if the code was compiled to do so. Bypasses the need to rebuild the code.
- **threadProtonTrace**: If true, proton tracing through a block is threaded. This runtime parameter can only be set during setup of the code.

## 28.0.7 Unit Test

The unit test for the proton imaging unit consists in sending a ring of protons perpendicular onto either a uniform circular  $B$  field or a radially outward uniform  $E$  field and measuring the deflection (radial increase) of the ring as well as the magnitudes of the magnetic diagnostic quantities  $k$  and  $J$  from Eqs.(28.31) and (28.32). Due to the circular  $B$  field or the radial outward  $E$  field, each proton experiences a radial outward force and the radial increase of the protons can be calculated analytically. Exact analytical expressions for  $k$  and  $J$  are also possible.

### 28.0.7.1 Deflection of a Proton Ring by a Uniform Circular Magnetic Field

Consider the case of a uniform circular magnetic field with constant magnetic flux density  $B$  pointing tangentially in clock- or anticlock-wise orientation around an axis, which we chose to be the  $y$  axis. If a proton has only a  $y$  component velocity  $v_{0y} \neq 0$  initially, then it will be deflected radially outward or inward, depending on the clock- or anticlockwise orientation of  $\mathbf{B}$ . To simplify the calculations, we assume that the proton is located on the positive  $x$  axis, such that at this point we have only a  $z$  magnetic component, that is  $B_z = \pm B$  and  $B_x, B_y = 0$ . Further we deal with the non-relativistic situation first, as the extension to the

relativistic case is trivial. We will use Eqs.(28.6-28.8) for that purpose. Since no electric field is present, we have that:

$$E_x, E_y, E_z, e_x, e_y, e_z, b_x, b_y = 0 \quad (28.47)$$

$$b_z = B_z/B = \pm 1 \quad (28.48)$$

$$v_{0x}, v_{0z} = 0. \quad (28.49)$$

Putting these values into Eqs.(28.6-28.8), the proton velocities become:

$$v_x(t) = \pm v_{0y} \sin[BQ_m t/c] \quad (28.50)$$

$$v_y(t) = v_{0y} \cos[BQ_m t/c] \quad (28.51)$$

$$v_z(t) = 0 \quad (28.52)$$

showing that the proton moves within the same  $z$ -plane. The  $+$  sign in  $v_x(t)$  applies, if  $B_z = +B$  (anticlockwise orientation) and the  $-$  sign, if  $B_z = -B$  (clockwise orientation). Integration over time gives the position equations:

$$r_x(t) = r_{0x} \pm \frac{cv_{0y}}{BQ_m} (1 - \cos[BQ_m t/c]) \quad (28.53)$$

$$r_y(t) = r_{0y} + \frac{cv_{0y}}{BQ_m} \sin[BQ_m t/c] \quad (28.54)$$

$$r_z(t) = r_{0z} \quad (28.55)$$

Taking the anticlockwise orientation of  $B$ , looking from above onto the fixed  $z$ -plane, the proton will perform a half-circle of radius  $cv_{0y}/BQ_m$ . We are interested in the (radial) deflection along the  $x$  axis. The proton will travel through the domain in the  $y$  direction, where on exit the detector screen is placed. If we place the detector right at the exit of the  $y$  direction, then we can calculate from the  $r_y(t)$  equation the time it takes for the proton to travel through (exit) the domain:

$$t_{exit} = \frac{c}{BQ_m} \arcsin\left(\frac{D_y BQ_m}{cv_{0y}}\right), \quad (28.56)$$

where  $D_y$  is the length of the domain in  $y$  direction. If  $D_y BQ_m/cv_{0y} > 1$ , then the proton will not exit the domain but rather curve back inside the domain. This happens if either  $D_y$  is too large (the domain extends too far into the  $y$  direction),  $B$  is too large or the initial velocity  $v_{0y}$  is too low. For a given  $D_y$  and  $v_{0y}$ , the limiting  $B$  would be:

$$B_{limit} = \frac{cv_{0y}}{D_y Q_m}. \quad (28.57)$$

Inserting the  $t_{exit}$  expression into the  $r_x(t)$  equation, we get:

$$r_x(t_{exit}) = r_{0x} + \frac{cv_{0y}}{BQ_m} \left(1 - \sqrt{1 - \left(\frac{D_y BQ_m}{cv_{0y}}\right)^2}\right). \quad (28.58)$$

The radial deflection  $\Delta r$  of the proton on the detector screen will be equal to  $r_x(t_{exit}) - r_{0x}$  and therefore:

$$\Delta r = \frac{cv_{0y}}{BQ_m} \left(1 - \sqrt{1 - \left(\frac{D_y BQ_m}{cv_{0y}}\right)^2}\right). \quad (28.59)$$

The limiting radial deflection (i.e., the maximal radial deflection) that can be detected by the screen at the domains exit is obtained by inserting the expression of  $B_{limit}$  into the last equation, which gets:

$$\Delta r_{limit} = D_y, \quad (28.60)$$

and is hence equal to the domains extension in  $y$  direction.

To evaluate the magnitude of the magnetic diagnostic quantities  $k$  and  $J$ , we start from the differential equations derived from Eqs.(28.33) and (28.34):

$$\frac{dk}{dt} = Bv_{0y}, \quad (28.61)$$

$$\frac{dJ}{dt} = v_y(t) \frac{\partial B_x}{\partial z} = -B \frac{v_y(t)}{r_x(t)}, \quad (28.62)$$

where the first equation follows from the fact that the velocity magnitude does not change and the second equation utilized the fact that for a proton positioned on the  $x$  axis only the term  $\partial B_x / \partial z$  from  $\nabla \times \mathbf{B}$  is nonzero (for anticlockwise rotation of  $B$  we have  $B_x = -zB/\sqrt{x^2 + z^2}$ ,  $B_y = 0$  and  $B_z = xB/\sqrt{x^2 + z^2}$ , which gives at  $z = 0$  locations  $\partial B_x / \partial z = -B/x$ ). Performing the time integration from  $t = 0$  to  $t = t_{exit}$  and using the expressions for  $v_y(t)$ ,  $r_x(t)$  and  $t_{exit}$  from Eqs.(28.51), (28.53) and (28.56), we get:

$$k(t_{exit}) = BD_y \frac{\arcsin(x)}{x}, \quad (28.63)$$

$$J(t_{exit}) = B \left( \arcsin(x) - 2 \frac{1 + 1/y}{\sqrt{1 + 2/y}} \arctan \left( \frac{x\sqrt{1 + 2/y}}{1 + \sqrt{1 - x^2}} \right) \right), \quad (28.64)$$

where we have used the dimensionless quantities:

$$x = \frac{D_y B Q_m}{c v_{0y}}, \quad y = \frac{r_{0x} B Q_m}{c v_{0y}}. \quad (28.65)$$

Since the electrical field is zero, inclusion of relativity into the deflection equations is trivial. There is no increase in velocity as the proton travels inside the domain, which means that  $\gamma(\mathbf{v})$  stays constant. To account for relativistic effects, one simply has to replace  $Q_m$  in the radial deflection and magnetic diagnostic equations by  $Q_{m\gamma}$ , where  $\gamma(\mathbf{v}) = 1/\sqrt{1 - v_{0y}^2/c^2}$ . For fixed parameters, the relativistic radial deflection is always less than the non-relativistic one. This can be shown by evaluating the ratio between the two radial deflections. Using the non-relativistic  $x$  as defined in Eq.(28.65), we have:

$$\frac{\Delta r_{non-rel}}{\Delta r_{rel}} = \frac{1 - \sqrt{1 - x^2}}{\gamma - \sqrt{\gamma^2 - x^2}} = \gamma \frac{1 + \sqrt{1 - (x/\gamma)^2}}{1 + \sqrt{1 - x^2}}, \quad (28.66)$$

and, since  $x^2 \leq 1$  and  $\gamma \geq 1$ , it follows that:

$$\frac{\Delta r_{non-rel}}{\Delta r_{rel}} \geq \gamma. \quad (28.67)$$

### 28.0.7.2 Deflection of a Proton Ring by a Uniform Radial Outward Electric Field

In this section we state the radial deflection equations in the presence of a radial outward directed electric field. To simplify again the calculations, we assume that the proton is located such that at this point we have only a  $x$  electric component, that is  $E_x = \pm E$  and  $E_y, E_z = 0$ . To include the relativistic case, we start from the relativistic acceleration in Eq.(28.16) and solve the set of differential equations corresponding to the unit test setup:

$$\dot{v}_x(t) = Q_{m\gamma} [E - v_x^2(t)E/c^2] \quad (28.68)$$

$$\dot{v}_y(t) = Q_{m\gamma} [-v_x(t)v_y(t)E/c^2] \quad (28.69)$$

$$\dot{v}_z(t) = Q_{m\gamma} [-v_x(t)v_z(t)E/c^2] \quad (28.70)$$

with initial conditions  $v_y(0) = v_{0y}$  and  $v_x(0), v_z(0) = 0$ . The results are:

$$v_x(t) = EQ_{m\gamma} t / \sqrt{1 + \delta(EQ_{m\gamma} t/c)^2} \quad (28.71)$$

$$v_y(t) = v_{0y} / \sqrt{1 + \delta(EQ_{m\gamma} t/c)^2} \quad (28.72)$$

$$v_z(t) = 0 \quad (28.73)$$

where  $Q_{m\gamma}$  is evaluated using  $\gamma(\mathbf{v}) = 1/\sqrt{1 - v_{0y}^2/c^2}$  with  $Q_{m\gamma} = Q_m$  for the non-relativistic case, and  $\delta$  is 1 for the relativistic case and 0 otherwise. Integration over time gives (note the relativistic corrections to the classical expressions):

$$r_x(t) = r_{0x} + \frac{EQ_{m\gamma}t^2}{2} \left[ 2 \left( \frac{EQ_{m\gamma}t}{c} \right)^{-2} \left( \sqrt{1 + \left( \frac{EQ_{m\gamma}t}{c} \right)^2} - 1 \right) \right]^\delta \quad (28.74)$$

$$r_y(t) = r_{0y} + v_{0y}t \left[ \left( \frac{EQ_{m\gamma}t}{c} \right)^{-1} \operatorname{arsinh} \left( \frac{EQ_{m\gamma}t}{c} \right) \right]^\delta \quad (28.75)$$

$$r_z(t) = r_{0z} \quad (28.76)$$

From the equation for  $r_y(t)$  we get the domain crossing time in the  $y$  direction as:

$$t_{exit} = \frac{D_y}{v_{0y}} \left[ \left( \frac{D_y EQ_{m\gamma}}{cv_{0y}} \right)^{-1} \sinh \left( \frac{D_y EQ_{m\gamma}}{cv_{0y}} \right) \right]^\delta \quad (28.77)$$

and the resulting radial reflection  $r_x(t_{exit}) - r_{0x}$  is:

$$\Delta r = \frac{D_y^2 EQ_{m\gamma}}{2v_{0y}^2} \left[ \left( \frac{D_y EQ_{m\gamma}}{2cv_{0y}} \right)^{-2} \sinh^2 \left( \frac{D_y EQ_{m\gamma}}{2cv_{0y}} \right) \right]^\delta. \quad (28.78)$$

In contrast to the magnetic field radial deflection, there is, for a fixed  $E$  a certain  $v_{0y}$  and for a fixed  $v_{0y}$  a certain  $E$ , beyond which the relativistic radial deflection is greater than the non-relativistic one. Using the simplification:

$$x = \frac{D_y EQ_{m\gamma}}{2cv_{0y}}, \quad (28.79)$$

we have for the ratio:

$$\frac{\Delta r_{non-rel}}{\Delta r_{rel}} = \frac{\gamma x^2}{\sinh^2(x)}. \quad (28.80)$$

But the function  $f(x) = x^2/\sinh^2(x)$  is a monotonic decaying function between  $f(x) = 1$  for  $x = 0$  and  $f(x) = 0$  for  $x \rightarrow \infty$ . Hence at some  $x$  we must have  $f(x) = 1/\gamma$ , beyond which we have  $\Delta r_{rel} > \Delta r_{non-rel}$ . The crossover radial deflection  $\Delta r_{cross}$ , at which both the non-relativistic and the relativistic radial reflections coincide, can be calculated by finding the solutions for  $x_{cross}$  and  $\Delta r_{cross}$  in the following two equations:

$$\gamma x_{cross}^2 = \sinh^2(x_{cross}) \quad (28.81)$$

$$\Delta r_{cross} = \gamma x_{cross} D_y c / v_{0y}. \quad (28.82)$$

Note that the first equation, defining  $x_{cross}$  as a function of  $\gamma$ , cannot be solved analytically and one must resort to numerical approaches.

### 28.0.7.3 FLASH Setup of the Unit Test

We set up a 3D cartesian domain consisting of a rectangular box with dimensions (in cm):  $x$  axis  $[0, 3]$ ,  $y$  axis  $[0, 1]$  and  $z$  axis  $[0, 3]$ . A circular beam of 10000 protons with radius 0.5cm is shot along the  $y$  axis perpendicular to the  $xz$  plane and centered on the  $xz$  plane at (1.5,1.5). The square detector screen was placed extremely close to the domain exit in  $y$  direction and was of the same size as the domain's  $xz$  plane, i.e., of side length 3cm and centered at (1.5,1.5). The variable parameters used were: 1) the proton energies (currently only 20MeV), 2) the uniform refinement levels (3,4,5) and the magnetic / electric fields (in Gauss). The following radial deflections  $\Delta r$  were recorded: 1) the theoretical  $\Delta r_{thr}$ , 2) the maximum  $\Delta r_{max}$ , 3) the minimum  $\Delta r_{min}$  and 4) the average  $\Delta r_{avg}$ .

#### 28.0.7.4 FLASH Unit Test Results for Uniform Circular Magnetic Fields

From the FLASH setup parameters, we can calculate the following values for 20 MeV protons:

$$\begin{aligned}
 v_{0y} &= 0.2032392740948 \, c \\
 \gamma &= 1.0213157779052 \\
 B_{limit} &= 636085.837576 \, \text{Gauss (non-relativistic, Eq. 28.57)} \\
 \Delta r_{limit} &= 1.0 \, \text{cm (Eq. 28.60)}.
 \end{aligned}$$

Tables 28.1, 28.2 and 28.3 show the obtained FLASH results for the radial deviation, the magnetic diagnostic  $J$  value and the magnetic diagnostic  $\mathbf{k}$  magnitude, including a run with  $B_{limit}$ .

Table 28.1: Magnetic outward radial deflections: maximum  $\Delta r_{max}$ , minimum  $\Delta r_{min}$ , average  $\Delta r_{avg}$  and theoretical  $\Delta r_{thr}$  from Eq.(28.59) on a ring of 20 MeV protons for uniform domain refinement levels 3,4,5. Subscript numbers indicate multiplication by negative exponents:  $n_e$  means  $n \times 10^{-e}$ . The radial deflection values are in cm. Italic numbers represent the relativistic results.

Proton energy 20 MeV (0.203 c), Refinement Levels = 3,4,5				
$B$	$\Delta r_{max}$	$\Delta r_{min}$	$\Delta r_{avg}$	$\Delta r_{thr}$
10,000	7.86106 <sub>3</sub>	7.85565 <sub>3</sub>	7.85991 <sub>3</sub>	7.86106 <sub>3</sub>
	7.86106 <sub>3</sub>	7.85995 <sub>3</sub>	7.86075 <sub>3</sub>	
	7.86106 <sub>3</sub>	7.86081 <sub>3</sub>	7.86098 <sub>3</sub>	
	<i>7.69697<sub>3</sub></i>	<i>7.69166<sub>3</sub></i>	<i>7.69585<sub>3</sub></i>	<i>7.69697<sub>3</sub></i>
	<i>7.69697<sub>3</sub></i>	<i>7.69588<sub>3</sub></i>	<i>7.69667<sub>3</sub></i>	
	<i>7.69697<sub>3</sub></i>	<i>7.69672<sub>3</sub></i>	<i>7.69690<sub>3</sub></i>	
100,000	7.90975 <sub>2</sub>	7.90635 <sub>2</sub>	7.90862 <sub>2</sub>	7.90975 <sub>2</sub>
	7.90975 <sub>2</sub>	7.90891 <sub>2</sub>	7.90946 <sub>2</sub>	
	7.90975 <sub>2</sub>	7.90956 <sub>2</sub>	7.90968 <sub>2</sub>	
	<i>7.74266<sub>2</sub></i>	<i>7.73932<sub>2</sub></i>	<i>7.74154<sub>2</sub></i>	<i>7.74266<sub>2</sub></i>
	<i>7.74266<sub>2</sub></i>	<i>7.74183<sub>2</sub></i>	<i>7.74237<sub>2</sub></i>	
	<i>7.74266<sub>2</sub></i>	<i>7.74246<sub>2</sub></i>	<i>7.74258<sub>2</sub></i>	
636,085	9.98376 <sub>1</sub>	9.80750 <sub>1</sub>	9.86908 <sub>1</sub>	9.98375 <sub>1</sub>
	9.98376 <sub>1</sub>	9.90287 <sub>1</sub>	9.93061 <sub>1</sub>	
	9.98376 <sub>1</sub>	9.94942 <sub>1</sub>	9.96196 <sub>1</sub>	
	<i>8.13739<sub>1</sub></i>	<i>8.12900<sub>1</sub></i>	<i>8.13357<sub>1</sub></i>	<i>8.13739<sub>1</sub></i>
	<i>8.13739<sub>1</sub></i>	<i>8.13531<sub>1</sub></i>	<i>8.13638<sub>1</sub></i>	
	<i>8.13739<sub>1</sub></i>	<i>8.13687<sub>1</sub></i>	<i>8.13713<sub>1</sub></i>	

#### 28.0.7.5 FLASH Unit Test Results for Uniform Radial Outward Electric Fields

From the FLASH setup parameters, we can calculate the following values for 20 MeV protons:

$$\begin{aligned}
 v_{0y} &= 0.2032392740948 \, c \\
 \gamma &= 1.0213157779052 \\
 x_{cross} &= 0.2518110572090 \, (\text{solution of Eq. 28.81}) \\
 E_{cross} &= 327175.337727 \, \text{Gauss (Eq. 28.79)} \\
 \Delta r_{cross} &= 1.2653981713131 \, \text{cm (Eq. 28.82)}.
 \end{aligned} \tag{28.83}$$

Table 28.4 shows the obtained FLASH results, including a run with  $E_{cross}$ .

Table 28.2: Magnetic diagnostic values: maximum  $J_{max}$ , minimum  $J_{min}$ , average  $J_{avg}$  and theoretical  $J_{thr}$  from Eq.(28.64) on a ring of 20 MeV protons for uniform domain refinement levels 3,4,5. Subscript numbers indicate multiplication by positive exponents:  $n_e$  means  $n \times 10^{+e}$ . All  $J$  values are in Gauss. Italic numbers represent the relativistic results.

Proton energy 20 MeV (0.203 c), Refinement Levels = 3,4,5				
$B$	$J_{max}$	$J_{min}$	$J_{avg}$	$J_{thr}$
10,000	-1.93499 <sub>4</sub>	-2.06117 <sub>4</sub>	-1.99244 <sub>4</sub>	-1.98962 <sub>4</sub>
	-1.96351 <sub>4</sub>	-2.01282 <sub>4</sub>	-1.98985 <sub>4</sub>	
	-1.98027 <sub>4</sub>	-1.99475 <sub>4</sub>	-1.98977 <sub>4</sub>	
	<i>-1.93499<sub>4</sub></i>	<i>-2.06117<sub>4</sub></i>	<i>-1.99272<sub>4</sub></i>	<i>-1.98983<sub>4</sub></i>
	<i>-1.96351<sub>4</sub></i>	<i>-2.01295<sub>4</sub></i>	<i>-1.99008<sub>4</sub></i>	
	<i>-1.98027<sub>4</sub></i>	<i>-1.99504<sub>4</sub></i>	<i>-1.98997<sub>4</sub></i>	
100,000	-1.88339 <sub>5</sub>	-1.92096 <sub>5</sub>	-1.90503 <sub>5</sub>	-1.90377 <sub>5</sub>
	-1.89600 <sub>5</sub>	-1.90952 <sub>5</sub>	-1.90412 <sub>5</sub>	
	-1.90084 <sub>5</sub>	-1.90575 <sub>5</sub>	-1.90385 <sub>5</sub>	
	<i>-1.88506<sub>5</sub></i>	<i>-1.92296<sub>5</sub></i>	<i>-1.90692<sub>5</sub></i>	<i>-1.90563<sub>5</sub></i>
	<i>-1.89761<sub>5</sub></i>	<i>-1.91166<sub>5</sub></i>	<i>-1.90596<sub>5</sub></i>	
	<i>-1.90277<sub>5</sub></i>	<i>-1.90757<sub>5</sub></i>	<i>-1.90571<sub>5</sub></i>	
636,085	-9.58761 <sub>5</sub>	-9.69203 <sub>5</sub>	-9.64623 <sub>5</sub>	-9.64102 <sub>5</sub>
	-9.62174 <sub>5</sub>	-9.65740 <sub>5</sub>	-9.64229 <sub>5</sub>	
	-9.63384 <sub>5</sub>	-9.64623 <sub>5</sub>	-9.64133 <sub>5</sub>	
	<i>-9.65418<sub>5</sub></i>	<i>-9.75959<sub>5</sub></i>	<i>-9.71335<sub>5</sub></i>	<i>-9.70812<sub>5</sub></i>
	<i>-9.68864<sub>5</sub></i>	<i>-9.72467<sub>5</sub></i>	<i>-9.70939<sub>5</sub></i>	
	<i>-9.70087<sub>5</sub></i>	<i>-9.71338<sub>5</sub></i>	<i>-9.70844<sub>5</sub></i>	

Table 28.3: Magnetic diagnostic values: maximum  $k_{max}$ , minimum  $k_{min}$ , average  $k_{avg}$  and theoretical  $k_{thr}$  from Eq.(28.63) on a ring of 20 MeV protons for uniform domain refinement levels 3,4,5. Subscript numbers indicate multiplication by positive exponents:  $n_e$  means  $n \times 10^{+e}$ . All  $k$  values are in units of Gauss cm. Italic numbers represent the relativistic results.

Proton energy 20 MeV (0.203 c), Refinement Levels = 3,4,5				
$B$	$k_{max}$	$k_{min}$	$k_{avg}$	$k_{thr}$
10,000	1.00004 <sub>4</sub>	9.99561 <sub>3</sub>	1.00001 <sub>4</sub>	1.00004 <sub>4</sub>
	1.00004 <sub>4</sub>	9.99924 <sub>3</sub>	1.00003 <sub>4</sub>	
	1.00004 <sub>4</sub>	1.00001 <sub>4</sub>	1.00004 <sub>4</sub>	
	<i>1.00004<sub>4</sub></i>	<i>9.99560<sub>3</sub></i>	<i>1.00001<sub>4</sub></i>	<i>1.00004<sub>4</sub></i>
	<i>1.00004<sub>4</sub></i>	<i>9.99922<sub>3</sub></i>	<i>1.00003<sub>4</sub></i>	
	<i>1.00004<sub>4</sub></i>	<i>1.00001<sub>4</sub></i>	<i>1.00003<sub>4</sub></i>	
100,000	1.00417 <sub>5</sub>	1.00378 <sub>5</sub>	1.00406 <sub>5</sub>	1.00417 <sub>5</sub>
	1.00417 <sub>5</sub>	1.00407 <sub>5</sub>	1.00413 <sub>5</sub>	
	1.00417 <sub>5</sub>	1.00414 <sub>5</sub>	1.00416 <sub>5</sub>	
	<i>1.00399<sub>5</sub></i>	<i>1.00360<sub>5</sub></i>	<i>1.00388<sub>5</sub></i>	<i>1.00399<sub>5</sub></i>
	<i>1.00399<sub>5</sub></i>	<i>1.00389<sub>5</sub></i>	<i>1.00396<sub>5</sub></i>	
	<i>1.00400<sub>5</sub></i>	<i>1.00397<sub>5</sub></i>	<i>1.00398<sub>5</sub></i>	
636,085	9.98129 <sub>5</sub>	9.86877 <sub>5</sub>	9.90802 <sub>5</sub>	9.98129 <sub>5</sub>
	9.98129 <sub>5</sub>	9.92972 <sub>5</sub>	9.94739 <sub>5</sub>	
	9.98129 <sub>5</sub>	9.95942 <sub>5</sub>	9.96739 <sub>5</sub>	
	<i>8.87495<sub>5</sub></i>	<i>8.86913<sub>5</sub></i>	<i>8.87219<sub>5</sub></i>	<i>8.87496<sub>5</sub></i>
	<i>8.87495<sub>5</sub></i>	<i>8.87350<sub>5</sub></i>	<i>8.87422<sub>5</sub></i>	
	<i>8.87495<sub>5</sub></i>	<i>8.87459<sub>5</sub></i>	<i>8.87477<sub>5</sub></i>	



Table 28.4: Electric outward radial deflections: maximum  $\Delta r_{max}$ , minimum  $\Delta r_{min}$ , average  $\Delta r_{avg}$  and theoretical  $\Delta r_{thr}$  from Eq.(28.78) on a ring of 20 MeV protons for uniform domain refinement levels 3,4,5. Subscript numbers indicate multiplication by negative exponents:  $n_e$  means  $n \times 10^{-e}$ . The radial deflection values are in cm. Italic numbers represent the relativistic results.

Proton energy 20 MeV (0.203 c), Refinement Levels = 3,4,5				
$E$	$\Delta r_{max}$	$\Delta r_{min}$	$\Delta r_{avg}$	$\Delta r_{thr}$
10,000	3.86765 <sub>2</sub>	3.86592 <sub>2</sub>	3.86708 <sub>2</sub>	3.86765 <sub>2</sub>
	3.86765 <sub>2</sub>	3.86722 <sub>2</sub>	3.86750 <sub>2</sub>	
	3.86765 <sub>2</sub>	3.86754 <sub>2</sub>	3.86761 <sub>2</sub>	
	<i>3.78700<sub>2</sub></i>	<i>3.78530<sub>2</sub></i>	<i>3.78644<sub>2</sub></i>	<i>3.78700<sub>2</sub></i>
	<i>3.78700<sub>2</sub></i>	<i>3.78658<sub>2</sub></i>	<i>3.78685<sub>2</sub></i>	
	<i>3.78700<sub>2</sub></i>	<i>3.78689<sub>2</sub></i>	<i>3.78696<sub>2</sub></i>	
100,000	3.86765 <sub>1</sub>	3.86647 <sub>1</sub>	3.86718 <sub>1</sub>	3.86765 <sub>1</sub>
	3.86765 <sub>1</sub>	3.86736 <sub>1</sub>	3.86752 <sub>1</sub>	
	3.86765 <sub>1</sub>	3.86757 <sub>1</sub>	3.86761 <sub>1</sub>	
	<i>3.79441<sub>1</sub></i>	<i>3.79325<sub>1</sub></i>	<i>3.79394<sub>1</sub></i>	<i>3.79441<sub>1</sub></i>
	<i>3.79441<sub>1</sub></i>	<i>3.79412<sub>1</sub></i>	<i>3.79429<sub>1</sub></i>	
	<i>3.79441<sub>1</sub></i>	<i>3.79434<sub>1</sub></i>	<i>3.79438<sub>1</sub></i>	
327,176	1.26540 <sub>0</sub>	1.24971 <sub>0</sub>	1.25927 <sub>0</sub>	1.26540 <sub>0</sub>
	1.26540 <sub>0</sub>	1.24978 <sub>0</sub>	1.25936 <sub>0</sub>	
	1.26540 <sub>0</sub>	1.24980 <sub>0</sub>	1.25938 <sub>0</sub>	
	<i>1.26540<sub>0</sub></i>	<i>1.24893<sub>0</sub></i>	<i>1.25896<sub>0</sub></i>	<i>1.26540<sub>0</sub></i>
	<i>1.26540<sub>0</sub></i>	<i>1.24899<sub>0</sub></i>	<i>1.25904<sub>0</sub></i>	
	<i>1.26540<sub>0</sub></i>	<i>1.24901<sub>0</sub></i>	<i>1.25907<sub>0</sub></i>	



## Chapter 29

# Proton Emission Unit

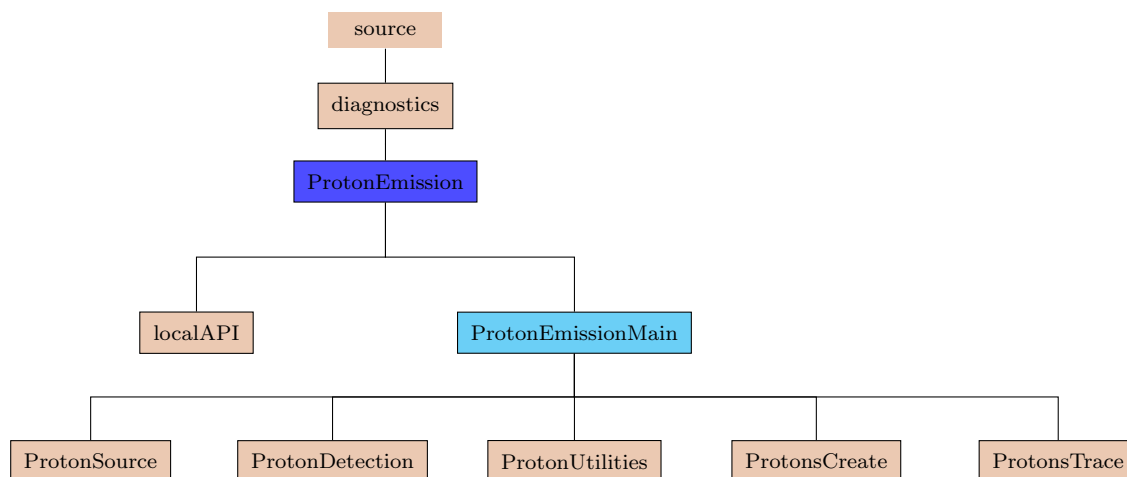


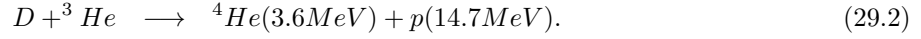
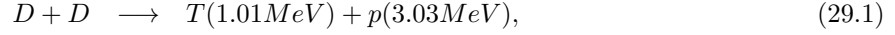
Figure 29.1: The `Proton Emission` unit directory tree.

When activated, the `Proton Emission` unit generates protons within the domain from nuclear fusion reactions, follows the generated protons through the domain and detects them on specific detector screens. Protons are deflected by Lorentz forces inside the domain due to presence of electric and magnetic fields. The function `ProtonEmission` controls the generation and translation of the protons inside the domain as well as their recording on the detector screens. For each cell in the domain the average electric and magnetic fields are used and the electric and magnetic components do not change within each cell. Each cell emits a certain number of protons (depending on temperature and fusion reaction used) statistically inside a spherical cone characterized by an opening half-angle and a direction. The half-angle of the emission cone can range between 0 and 180 degrees, the latter implying emission of the protons into the complete surrounding sphere. For each detector an optional pinhole can be placed between the domain and the detector screen. Currently the proton emission code assumes emission of protons without altering the temperature and the chemical composition of the cell. This is an approximation valid only for low reaction rates.

### 29.0.1 Proton Generation

Protons are generated by fusion reactions in domain regions of high temperature. Depending on the type of nuclei present, several fusion reactions can take place in a cell. Currently there are two proton generating

fusion reactions implemented in FLASH:



Nuclear reaction cross sections  $\sigma$  are direct measures of probability of nuclear reactions. Let  $\Phi_{P0}$  be the initial flux of projectile nuclei P onto a homogeneous thin sheet of nuclei S and  $\Phi_P$  the outcoming flux of projectile nuclei from the sheet. The flux of P is the number of P's per unit area per unit time. The difference in flux  $\Phi_{P0} - \Phi_P$  due to nuclear reactions between P and S (there are other phenomena that lead to a reduction in flux, for example scattering) is the number of reactions that happened in the sheet per unit area per unit time and is directly proportional to the thickness  $x$  of the sheet, the initial flux of P's and the nuclei number density  $n_S$  of S's in the sheet:

$$\Phi_{P0} - \Phi_P = \sigma_S n_S x \Phi_{P0}. \quad (29.3)$$

The proportionality constant  $\sigma_S$  has dimensions of area and is defined as the nuclear reaction cross section for S. Since always  $0 \leq \Phi_P \leq \Phi_{P0}$ , the dimensionless quantity  $(\Phi_{P0} - \Phi_P)/\Phi_{P0}$  denotes the probability of the nuclear reaction to happen in the sheet. The quantity  $n_S x$  can be viewed as the number of S's per unit area of the sheet. Hence  $1/n_S x$  is the area in the sheet covered by each S. Rewriting the above equation we have:

$$\sigma_S = \left( \frac{1}{n_S x} \right) \left( \frac{\Phi_{P0} - \Phi_P}{\Phi_{P0}} \right). \quad (29.4)$$

The cross section of S is thus equal to the area of each S in the sheet times the probability of nuclear reaction occurring in the sheet. It is thus the effective area of each S in the sheet that leads to a nuclear reaction.  $\sigma_S$  is dependent on the relative velocity of each P towards the S's in the sheet. Let us denote this velocity by  $v$  and let us further consider only sheet sections in which all incident P's have the same  $v$ . The sheet thickness will be traversed by each P in time  $t$  and we have  $vt = x$ . Substituting this into the above expression for  $\sigma_S$  and rearranging gives:

$$\sigma_S(v)v = \left( \frac{1}{n_S t} \right) \left( \frac{\Phi_{P0} - \Phi_P}{\Phi_{P0}} \right). \quad (29.5)$$

The quantity  $\sigma_S(v)v$  is called the reactivity of S and represents the probability of reaction per unit density of S per unit time in the sheet. Let us multiply the expression on the r.h.s. by  $n_P/n_P$ , where  $n_P$  is the number density of incident P's onto the sheet. We have, after grouping together:

$$\sigma_S(v)v = \left( \frac{1}{n_S n_P} \right) \left( \frac{\Phi_{P0} - \Phi_P}{\Phi_{P0}} \right) \left( \frac{n_P}{t} \right). \quad (29.6)$$

The quantity  $(\Phi_{P0} - \Phi_P)n_P/\Phi_{P0}t$  represents the number of P's in the sheet that reacted per volume unit per time unit. It is known as the volumetric reaction rate between P and S (number of reactions per volume per time) in the sheet. If we swap the particles with the target nuclei in the sheet, i.e. if we consider the P's stationary in the sheet and the S's moving in the opposite direction, then  $v$  is the same and we have:

$$\sigma_P(v)v = \left( \frac{1}{n_S n_P} \right) \left( \frac{\Phi_{S0} - \Phi_S}{\Phi_{S0}} \right) \left( \frac{n_S}{t} \right), \quad (29.7)$$

where now  $\Phi_S$  refers to the flux of S's in the opposite direction. The volumetric reaction rate  $R_{PS}$  between P and S must be identical in both pictures, hence:

$$R_{PS} = \left( \frac{\Phi_{P0} - \Phi_P}{\Phi_{P0}} \right) \left( \frac{n_P}{t} \right) = \left( \frac{\Phi_{S0} - \Phi_S}{\Phi_{S0}} \right) \left( \frac{n_S}{t} \right) \quad (29.8)$$

and therefore the nuclear cross sections  $\sigma_S(v)$  and  $\sigma_P(v)$  are equal and will be referred to as the nuclear reaction cross section  $\sigma(v)$ . The volumetric reaction rate is

$$R_{PS} = n_P n_S \sigma(v)v, \quad (29.9)$$

where  $v$  is the relative velocity between P's and S's. This equation allows for a comparison to reaction rates of 2nd order chemical reactions. The number densities  $n_P$  and  $n_S$  of the nuclei play the role of concentrations of reactants. The quantity  $\sigma(v)v$  is analogous to the chemical reaction constant, which in case of nuclear reactions depends on the approach velocity of the nuclei and their effective interaction area. If both nuclei P and S are the same, then using both  $n_P$  and  $n_S$  separately in the rate equation would overcount the number of reactions by 2. If there are  $n$  identical nuclei in a certain volume, one can only form  $n(n-1)/2$  distinct pairs, which for large  $n$  becomes  $\approx n^2/2$ . The volumetric reaction rate including the possibility for identical nuclei reads

$$R_{PS} = \frac{n_P n_S}{1 + \delta_{PS}} \sigma(v)v, \quad (29.10)$$

where  $\delta_{PS}$  is 1, if P and S are identical and 0 elsewhere.

In a cell with a definite temperature  $T$ , the relative velocities between the interacting nuclei vary and one must use a velocity distribution function  $f(v)$  (most common used is Maxwellian). The volumetric nuclear reactivity  $\sigma(v)v$  must therefore be replaced by an averaged reactivity

$$\langle \sigma v \rangle = \int_0^\infty \sigma(v) f(v) v dv \quad (29.11)$$

and the volumetric reaction rate becomes

$$R_{PS} = \frac{n_P n_S}{1 + \delta_{PS}} \langle \sigma v \rangle. \quad (29.12)$$

The dimensions of  $\langle \sigma v \rangle$  are usually given in  $\text{cm}^3 \text{s}^{-1}$ . It can be calculated from experimentally determined cross sections assuming a Maxwellian velocity distribution of the nuclei at a certain temperature and integrating over the entire velocity range. Since these integrations are time consuming, the values of  $\langle \sigma v \rangle$  as a function of temperature are conveniently fitted using simple functional forms. For both above reactions, FLASH uses the functional fit as provided by Bosch and Hale (1992):

$$\langle \sigma v \rangle = C_1 \eta^{-5/6} \xi^2 \exp(-3\eta^{1/3} \xi), \quad (29.13)$$

where

$$\xi = C_0 / T^{1/3} \quad (29.14)$$

$$\eta = 1 - \frac{C_2 T + C_4 T^2 + C_6 T^3}{1 + C_3 T + C_5 T^2 + C_7 T^3}. \quad (29.15)$$

The values of the coefficients from  $C_0$  through  $C_7$  and the valid temperature range of the fit can be found in Atzeni and Meyer-Ter-Vehn (2004).

The tracing of the protons through the domain is analogous to the one used for proton imaging. Currently only  $(x, y)$  pairs are recorded for each proton on each detector screen. Time resolved proton emission is not yet implemented, i.e. the protons exit the domain in one time step once they are created.

## 29.0.2 Proton Detector Screens

The setup of the proton detector screens and the proton recording technique is the same as for the proton imaging unit (section 28.0.4). Option for additional pinholes is provided as well. As there are no beams in this unit, the protons are recorded on the nearest detector screen in 3D space. If protons do not hit any detector screen, they will be lost. As the protons are not associated with a particular beam and detector, there is no option for recording offscreen protons.

## 29.0.3 Proton Emission Boxes

In order to allow for more simulation flexibility, the proton emission code is equipped with the possibility of selecting active proton emission regions (boxes) inside the domain. If no such boxes are specified, the entire domain is active, i.e. protons will be generated from the entire domain. If one or more emission boxes are given, protons will only be generated from inside the box. Proton emission boxes are allowed to overlap in space, but are not allowed to be completely outside the domain boundaries. Each emission box is characterized by its rectangular bounding box coordinates (lower left and upper right corners).

### 29.0.4 Usage

To include the use of the Proton Emission unit, the following should be included into the setup line command:

```
+protonEmission [pem_maxDetectors=<number> pem_maxEmissionBoxes=<number> threadProtonTrace=True]
```

The +protonEmission shortcut handles all the logistics for properly including the `ProtonEmission` unit. The default settings are: maximum number of detectors = 1, maximum number of emission boxes = 1 and no use of threading during tracing of protons through the domain, which can be changed by the following three setup variables:

- **pem\_maxDetectors:** The maximum number of proton emission detectors for the simulation.
- **pem\_maxEmissionBoxes:** The maximum number of proton emission boxes for the simulation.
- **threadProtonTrace:** Enables threading during proton domain tracing.

The runtime parameters for the proton emission unit are very similar to those for the proton imaging. Setup and placement of detector parameters are identical to the proton imaging detectors.

#### 29.0.4.1 Proton Emission Detectors Runtime Parameters

The following are the runtime parameters for the proton emission detectors. The `_n` at the end of each runtime parameter characterizes the detector number.

- **pem\_numberOfDetectors:** The number of proton emission detectors that are going to be used.
- **pem\_detectorCenter[X,Y,Z]\_n:** The global 3D components of the detector center position vector  $\mathbf{C}$ .
- **pem\_detectorNormal[X,Y,Z]\_n:** The local 3D components of the detector normal vector  $\mathbf{n}$ .
- **pem\_detectorSideLength\_n:** The side length (cm) of the detector square screen.
- **pem\_detectorSideTiltingAngle\_n:** The tilting angle (degrees) of two sides parallel to the detector screen unit axis  $\mathbf{u}_y$  with respect to the tilting axis.
- **pem\_detectorSideTiltingAxis\_n:** The global tilting axis ('x','y' or 'z') for the detector screen.
- **pem\_detectorPinholeRadius\_n:** The radius of the pinhole. If  $\leq 0$  no pinhole is used.
- **pem\_detectorPinholeDist2Det\_n:** The pinhole center distance from the detector center, useful only if a pinhole radius  $> 0$  was specified.

#### 29.0.4.2 Proton Emission Boxes Runtime Parameters

The following are the runtime parameters for the proton emission boxes. The `_n` at the end of each runtime parameter characterizes the emission box number.

- **pem\_numberOfEmissionBoxes:** The number of proton emission boxes that are going to be used.
- **pem\_emissionBoxCornerL[X,Y,Z]\_n:** The global 3D components of the emission box lower (left) corner.
- **pem\_emissionBoxCornerU[X,Y,Z]\_n:** The global 3D components of the emission box upper (right) corner.

### 29.0.4.3 Proton Emission General Runtime Parameters

- **pem.appendOldDetectorFiles:** If set to true, emission protons will be added to old existing detector files, provided old and new detector file names match.
- **pem.cellStepTolerance:** This factor times the smallest dimension of each cell is taken as the positional error tolerance for the protons during their path (parabolic or Runge-Kutta) tracing through each cell.
- **pem.cellWallThicknessFactor:** Controls the (imaginary) thickness of the cell walls to ensure computational stability of the proton emission code. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to  $10^{-6}$  and should only very rarely be changed.
- **pem.detailedTiming:** If set to true, detailed timing info for several stages during the proton emission will be provided in the application logfile.
- **pem.detectorFileNameTimeStamp:** If set to true, a time stamp will be added to each detector file name. This allows for time splitting of detectors.
- **pem.detectorXYwriteFormat:** Controls formatted ascii output of the proton  $(x, y)$  pairs on the detector screens (default 'es20.10').
- **pem.emissionAmplificationFactor:** Amplifies the proton emission count of each cell by this factor.
- **pem.emissionConeHalfApexAngle:** The half apex angle (degrees) of each cell emission cone.
- **pem.emissionConeCenter[X,Y,Z]:** The global 3D components of the emission cone center. This is for directional purposes only. Each cell sends an emission cone with the specified half apex angle to this center location.
- **pem.ignoreElectricalField:** If true, the electrical field in each cell is ignored.
- **pem.ignoreMagneticField:** If true, the magnetic field in each cell is ignored.
- **pem.maxProtonCount:** The maximum number of emitted protons that can be created on one processor per batch in the domain.
- **pem.opaqueBoundaries:** If true, the protons do not go through cells marked as opaque boundaries.
- **pem.printDetectors:** If true, it prints detailed information about the proton detectors to a file with name <basename>ProtonEmissionDetectors.txt, where <basename> is the base name of the simulation.
- **pem.printMain:** If true, it prints general information regarding the proton emission setup to a file with name <basename>ProtonEmissionMain.txt, where <basename> is the base name of the simulation.
- **pem.printProtons:** If true, it prints detailed information about the all protons generated during the simulation. Protons are 'generated' on the domain surface and the info is written to several files labeled by batch number BID, processor rank number PID and a time stamp. Each processor writes its own file(s) with name(s): <basename>printProtonsBatch<BID>Proc<PID>.txt, where <basename> is the base name of the simulation. Use of this feature is reserved ONLY for debugging purposes and is currently limited to 10 batches and 100 processors per time stamp. Usage of a larger number of batches/processors during a simulation does not abort the run, but protons on batches with BID > 10 and processors with PID > 99 are simply ignored and not printed. Users other than code developers should not activate this feature.
- **pem.printProtonSources:** If true, it prints general information regarding the proton emission setup to a file with name <basename>ProtonEmissionMain.txt, where <basename> is the base name of the simulation.

- **pem\_protonDeterminism**: If true, the Grid Unit will be forced to use the sieve algorithm to move the proton particle data. Forcing this algorithm will result in a slower movement of data, but will fix the order the processors pass data and eliminate round off differences in consecutive runs.
- **pem\_RungeKuttaMethod**: Specifies which Runge Kutta method to use for proton tracing. Current options are: 'CashKarp45' (order 4, default), 'EulerHeu12' (order 1), 'BogShamp23' (order 2), 'Fehlberg34' (order 3) and 'Fehlberg45' (order 4).
- **pem\_screenProtonBucketSize**: Sets the bucket size for flushing screen protons out to disk.
- **pem\_useParabolicApproximation**: If true, the code traces protons parabolically through cells for low  $B$  / high  $v$  combinations (section 28.0.1.2).
- **useProtonEmission**: If false, no proton emission will be performed, even if the code was compiled to do so. Bypasses the need to rebuild the code.
- **threadProtonTrace**: If true, proton tracing through a block is threaded. This runtime parameter can only be set during setup of the code.



## Chapter 30

# Thomson Scattering Unit

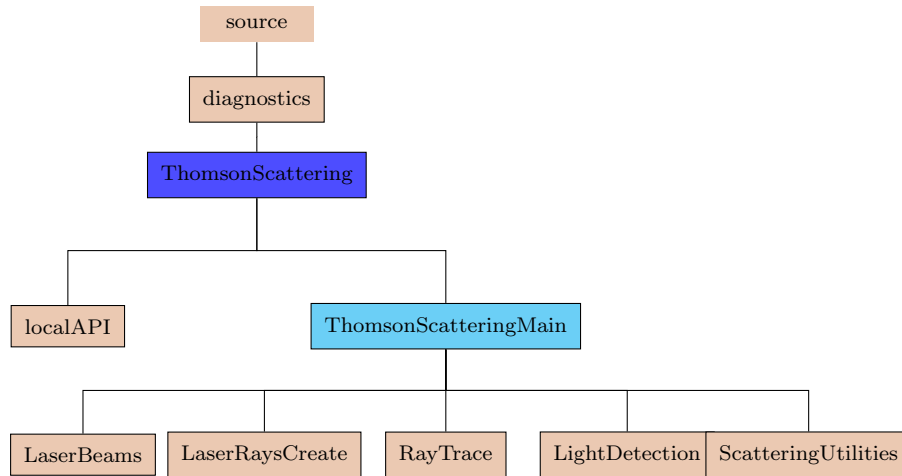


Figure 30.1: The `ThomsonScattering` unit with ray tracing directory tree.

The `ThomsonScattering` unit records a Thomson scattering spectrum of certain sections of the domain at certain simulation times. There are currently two versions of the unit: 1) a more recent version, which takes into account deflection of both laser and detector rays as well as energy deposition by the laser rays and 2) an initial simple version in which no ray tracing is performed (i.e. all laser and detector rays move in straight lines). The first version is computationally more involved, as it is not known beforehand where the Thomson scattering region is located due to ray deflection. To preserve flexibility and for comparison tests, the first version allows for suppression of ray deflection and energy deposition as well as ray power depletion while crossing the domain. This way one can study different ray effects on the Thomson spectra.

Currently, for the first version, ray deflection is done using the cell average algorithm [AVG 18.4.4.1](#) described in the `EnergyDeposition` unit. Placement of several detectors are possible, with more than one detector receiving rays from the same laser. However, no detector can record spectra coming from different lasers. The possibility of placing several detectors associated with the same laser allows a detailed study of how the Thomson spectra change with orientation of the detector.

### 30.1 Thomson Scattering including Ray Tracing and Deflection

The Thomson scattering unit including ray tracing and deflection is a detector driven unit, meaning that the outer loop is over all Thomson detectors specified. Since for each detector there is only one associated laser, the code checks whether the associated laser is currently active and if it is, this laser/detector pair is

being processed. The inclusion of ray deflection in a Thomson scattering code presents some computational challenges. The main difficulty that arises is that the Thomson scattering interaction region (IR) from where the spectra will be accumulated, is not known beforehand, but must be established in a first pass through the ray tracing procedure. For each laser/detector pair, the ray tracing Thomson scattering code consists of two main sections during each time step of the simulation: 1) determination of the interaction region and 2) assembling the Thomson spectra. Both sections require a complete laser/detector ray tracing procedure over the entire set of both laser and detector rays. In what follows we will describe each section separately.

### 30.1.1 Identifying the Interaction Region

Per definition, the interaction region (IR) consists of those cells of the domain that will have both: 1) incoming laser (I, from incident) rays and 2) emerging detector (S, from scattered) rays. Each of these so called IR cells contribute to the final spectrum. In order to find all the IR cells for a specific laser/detector pair, the complete set of rays for both the laser and the detector are launched towards the domain and the cells on which both sets intersect will be identified as IR cells. Both laser and detector rays are launched statistically from their corresponding lenses and towards their target areas. Ray creation, specification and storage follow much the same way as presented for the `EnergyDeposition` unit in chapter 18.4.7. Setup of laser/detector beams and pulses are also done as presented in chapters 18.4.6 and 18.4.5. If no Ray deflection is specified and both laser and detector beams are cylindrical in shape, the resulting set of IR cells are contained in a region resembling a Steiner solid, i.e. the space obtained from two intersecting cylinders. Ray deflection usually broadens up the IR region and there can be even cases in which the set of IR cells within the IR region is not connected, for example if the number of rays used for the laser and the detector is very low. The user must therefore make sure that the resulting IR region is dense by launching enough rays from the laser and the detector. On the other hand, as we will see below, assembling the Thomson spectrum from the set of each IR cell's laser (I) and detector (S) rays is a multiplicative task in which each I-ray is allowed to scatter into all S-rays. There is hence a delicate balance as to how many rays are launched and the size of the intended IR region for computations to remain manageable.

The identification of each IR cell is done by introducing extra two counting cell variables ICNT (for laser) and SCNT (for detector) into the UNK array (see chapter 6.1). ICNT and SCNT are incremented by +1 whenever a laser or detector ray hits the cell. After the complete sets of laser and detector rays have been processed through the domain, the IR cells are those for which both ICNT and SCNT are  $> 0$ . Once an IR cell has been identified, its ICNT value is overwritten by its unique tag, which will serve both as a counting index as well as identifying this cell as an IR cell (for example, for visualization purposes) in the domain.

During identification of the IR cells, the laser rays will not be allowed to deposit energy in the domain, as this will change subsequent ray tracing for the same laser/detector pair as well as other still to be processed laser/detector pairs. Also power loss by both laser and detector rays is not done at this stage.

### 30.1.2 Assembling the Thomson Scattering Spectra

Using maximum sized array dimension information from the interaction region finding step, storage arrays are allocated that will hold detailed info about each I-ray and S-ray for each IR cell. The complete set of laser and detector rays for the current laser/detector pair is retraced through the domain, this time all the laser and detector ray info (like direction components of each ray, current power and distance travelled inside each IR cell) as well each IR cell info (like number electron density, electron and ion temperature, etc.) is stored for subsequent assembly of the spectra. This time each ray is allowed to lose power: a laser ray loses absolute power, a detector ray loses relative power starting from an initial power of 1. The total power loss a ray experiences when travelling from the laser lens to the detector lens is given by the absolute power loss times the relative power loss, since power is lost exponentially depending on the time the ray spends in each cell (see Eq.(18.47)).

On each node the local Thomson spectra are assembled by looping over all local IR cells. Each IR cell contains per definition a set  $nI$  of incoming laser rays and a set  $nS$  of scattered detector rays. Each laser/detector ray pair (total of  $nI * nS$ ) info is now used to determine its contribution to the local spectrum. Each such pair defines a local IR cell scattering angle as well as a scattering direction. Together with the info of the local IR cell, a spectrum contribution to all recording frequencies is calculated using the time-averaged

scattered power equation for a collection of charges in the low-velocity charge ( $v/c \ll 1$ ) limit:

$$\frac{dP_s}{d\omega_s} = P_i r_e^2 L n_e d\Omega |\hat{s} \times (\hat{s} \times \hat{E}_i)|^2 S(\mathbf{k}, \omega), \quad (30.1)$$

where  $P_s$  is the time-averaged scattered power,  $\omega_s$  the scattered frequency,  $\omega = \omega_s - \omega_i$  the frequency shift between the laser and scattered frequency,  $P_i$  the incident laser ray power,  $r_e$  the classical electron radius,  $L$  the distance travelled by the ray inside the IR cell,  $n_e$  the electron number density in the IR cell,  $d\Omega$  the detector lens area in steradians,  $\hat{s}$  the scattering direction unit vector,  $\hat{E}_i$  the unit vector along the incident laser electric field,  $\mathbf{k} = \mathbf{k}_s - \mathbf{k}_i$  the shift in wavenumber between the laser and scattering wave vectors and  $S(\mathbf{k}, \omega)$  the spectral density function. For an unpolarized laser, the unitless scattering direction factor  $|\hat{s} \times (\hat{s} \times \hat{E}_i)|^2$  is averaged over all possible  $\hat{E}_i$  orientations:

$$|\hat{s} \times (\hat{s} \times \hat{E}_i)|^2 = 1 - \frac{1}{2} \sin^2 \theta, \quad (30.2)$$

where  $\theta$  is the angle between the laser direction and the scattering direction (zero in case of same direction). If a polarized laser is used, an additional angle  $\phi$  between the polarization plane and the scattering plane is needed, giving:

$$|\hat{s} \times (\hat{s} \times \hat{E}_i)|^2 = 1 - \sin^2 \theta * \cos^2 \phi. \quad (30.3)$$

The spectral density function describes how the spectrum varies in terms of velocity distributions and correlations of the ions with the scattering electrons and thus contains all the plasma information of the IR cell. Currently only the spectral density function for an unmagnetized, collisionless, low-temperature plasma containing possibly several ions is implemented:

$$S(\mathbf{k}, \omega) = \frac{1}{k} \left| \frac{1 + \sum_i \chi_i}{1 + \chi_e + \sum_i \chi_i} \right|^2 f_e\left(\frac{\omega}{k}\right) + \frac{1}{k} \left| \frac{\chi_e}{1 + \chi_e + \sum_i \chi_i} \right|^2 \sum_i \frac{n_{ei}}{n_e} Z_i f_i\left(\frac{\omega}{k}\right), \quad (30.4)$$

where  $\chi_e$  and  $\chi_i$  are the electron and individual ion susceptibilities and  $n_{ei}$  and  $n_e$  the number of electrons from the  $i$ -th ion and the total number of electrons, respectively.  $f_e(\omega/k)$  and  $f_i(\omega/k)$  are the electron and  $i$ -th ion one-dimensional Maxwellian velocity distributions for the phase velocity  $\omega/k$ :

$$f_q(\omega/k) = \sqrt{\frac{m_q}{2\pi k_B T_q}} * \exp\left(-\frac{m_q(\omega/k)^2}{2k_B T_q}\right), \quad (30.5)$$

with  $k_B$  the Boltzmann constant and  $m_q$  and  $T_q$  the electron ( $q = e$ ) and ion ( $q = i$ ) mass and temperature. Note the two kind of electron-ion correlation parts in the above  $S(\mathbf{k}, \omega)$  expression: 1) the first term represents the ion influence on the electron velocity distribution and 2) the second term is the electron influence on the ions velocity distribution. Derivation of the  $S(\mathbf{k}, \omega)$  expression is involved and the reader is referred to the literature (Froula et al. 2011, chapters 3 and 5). Since the units of the lhs of Eq.(30.1) are power per radial frequency (i.e. power per rad/s), we see that the units for  $S(\mathbf{k}, \omega)$  must be s/rad, which is also seen from Eq.(30.4), where the units of  $k$  are rad/cm and the units for the velocity distribution functions is s/cm and the other quantities are dimensionless. The electron and ion susceptibilities are calculated using derivatives of the plasma dispersion function  $W$ :

$$\chi_q(k, \omega) = -\frac{1}{2} \left( \frac{\omega_{Pq}}{v_{Tq} k} \right)^2 \frac{\partial W(\xi)}{\partial \xi}, \quad (30.6)$$

where again  $q = e, i$  represent the electrons or ions,  $\omega_{Pq}$  is the charge plasma frequency and  $v_{Tq} = \sqrt{k_B T_q / m_q}$  is the RMS thermal velocity of the charge.  $\xi = (\omega/k) / \bar{v}_{Tq}$  is the mean thermal velocity normalized phase velocity of the charge, with  $\bar{v}_{Tq} = \sqrt{2} v_{Tq}$ . Details about how to evaluate  $\partial W(\xi) / \partial \xi$  efficiently can again be inferred from Froula et al. 2011.

For higher temperature plasmas, relativistic effects start to get important, but their inclusion into the scattering equation becomes much more involved. In specific cases it is possible to include relativistic effects

up to first order in  $v/c$ . For example, if the incident Thomson laser is polarized such that its electrical field  $\mathbf{E}_i$  is perpendicular to the scattering plane and if a polarizer is placed before the detector that only detects scattered fields with electrical field parallel to  $\mathbf{E}_i$ , then it can be shown that Eq.(30.1) needs only be augmented multiplicatively by the relativistic term:

$$\text{rel.Term} = \left(1 + \frac{2\omega}{\omega_s}\right). \quad (30.7)$$

This option has been added to the Thomson scattering code, with a default of setting this term equal to 1 when no relativistic treatment is wanted. The user must however remember that this relativistic correction applies only under very specific Thomson scattering setups.

After all local spectra contributions have been evaluated, the global contribution to the spectrum for the current laser/detector pair is calculated and added to the overall Thomson spectrum for the detector. Writeout to the Thomson detector file is done after the laser for each detector is no longer active.

### 30.1.3 Usage

To include the use of the `ThomsonScattering` unit with ray tracing, the following should be included into the setup line command:

```
+ThomsonScatteringWrt thsc\_maxPulses=<number> thsc\_maxPulseSections=<number>
thsc\_maxLaserBeams=<number> thsc\_maxDetectors=<number>
```

The `+ThomsonScatteringWrt` is a shortcut that handles all the logistics for properly including the Thomson scattering unit in its ray tracing version. The next four setup variables override default values and it is always recommended to give appropriate values corresponding to the intended application.

- **thsc\_maxPulses:** The maximum number of different laser pulses for the simulation.
- **thsc\_maxPulseSections:** The maximum number of power/time pairs per pulse.
- **thsc\_maxLaserBeams:** The maximum number of Thomson laser beams for the simulation.
- **thsc\_maxDetectors:** The maximum number of Thomson detectors for the simulation.

The `ThomsonScattering` unit reads all the information it needs to construct the Thomson scattering environment from runtime parameters specified in the `flash.par` file. Below is the list of runtime parameters that is needed to properly set up the Thomson scattering unit. As the Thomson pulse, laser and detector setup resembles the `EnergyDeposition` unit pulse and laser setup, it is encouraged to consult Figures 18.5, 18.6 and 18.7. Both the Thomson laser and the Thomson detector rays are created statistically on their respective lens and target areas. Regular grids, as for the lasers of the `EnergyDeposition` unit, are not supported.

#### 30.1.3.1 Thomson Laser Pulses Runtime Parameters

- **thsc\_numberOfPulses:** Controls the number of different Thomson laser pulses that are going to be used.
- **thsc\_numberOfSections\_n:** Gives the number of power/time pairs that are going to be used to set up the shape of the n-th Thomson laser pulse. There must be at least as many of these runtime parameters as there are number of Thomson laser pulses defined, i.e.  $n = 1, \dots, \text{thsc\_numberOfPulses}$ .
- **thsc\_power\_n.i:** Sets the i-th power of the i-th power/time pair of the n-th Thomson laser pulse. The ranges of the indices must be at least:  $i = 1, \dots, \text{thsc\_numberOfSections\_n}$  and  $n = 1, \dots, \text{thsc\_numberOfPulses}$ .
- **thsc\_time\_n.i:** Sets the i-th time of the i-th power/time pair of the n-th Thomson laser pulse. The ranges of the indices must be at least:  $i = 1, \dots, \text{thsc\_numberOfSections\_n}$  and  $n = 1, \dots, \text{thsc\_numberOfPulses}$ .

### 30.1.3.2 Thomson Laser Beams Runtime Parameters

- **thsc\_numberOfLaserBeams**: The number of Thomson laser beams that are going to be used.
- **thsc\_laserCrossSecFuncType\_n**: Thomson laser beam cross section power function (flat or gaussian decay) type. For a flat profile use 'uniform', for gaussian decay use 'gaussian2D' (n-th Thomson laser beam).
- **thsc\_laserGaussCenterMajor\_n**: The Gaussian center location along the major elliptical semiaxis (n-th Thomson laser beam).
- **thsc\_laserGaussCenterMinor\_n**: The Gaussian center location along the minor elliptical semiaxis (n-th Thomson laser beam).
- **thsc\_laserGaussExponent\_n**: The Gaussian super exponent  $\gamma$  for the Thomson laser beam cross section power function as presented in equation 18.106 (n-th Thomson laser beam).
- **thsc\_laserGaussRadiusMajor\_n**: The Gaussian radius (e-folding length)  $R_x$  along the major elliptical semiaxis as shown in equation 18.106 (n-th Thomson laser beam).
- **thsc\_laserGaussRadiusMinor\_n**: The Gaussian radius (e-folding length)  $R_y$  along the minor elliptical semiaxis as shown in equation 18.106 (n-th Thomson laser beam).
- **thsc\_laserInitialRaySpeed\_n**: The initial speed of the Thomson laser rays when hitting the domain boundary, in units of the speed of light (n-th Thomson laser beam).
- **thsc\_laserLensSAMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the laser elliptical lens area (n-th Thomson laser beam).
- **thsc\_laserLensX\_n**: The x-component of the laser global lens center position vector  $\mathbf{L}$  (n-th Thomson laser beam).
- **thsc\_laserLensY\_n**: The y-component of the laser global lens center position vector  $\mathbf{L}$  (n-th Thomson laser beam).
- **thsc\_laserLensZ\_n**: The z-component of the laser global lens center position vector  $\mathbf{L}$  (n-th Thomson laser beam).
- **thsc\_laserNoEnergyDeposition\_n**: If set true, no energy deposition is done for the Thomson laser rays (n-th Thomson laser beam).
- **thsc\_laserNumberOfRays\_n**: Number of statistical Thomson laser rays to be created for the Thomson laser beam (n-th Thomson laser beam).
- **thsc\_laserPolarizationAngle\_n**: Polarization angle  $\phi$  (in degrees) of the incident Thomson laser electrical field with respect to the scattering plane (cf. Eq.(30.3)). If a negative value is given, the Thomson laser beam is considered to be unpolarized (n-th Thomson laser beam).
- **thsc\_laserPulseNumber\_n**: The pulse shape identification number for the laser (n-th Thomson laser beam).
- **thsc\_laserSAMajorTorsionAngle\_n**: The major elliptical semiaxis torsion angle  $\phi_1$  along the Thomson beam's lens target center line (n-th Thomson laser beam).
- **thsc\_laserSAMajorTorsionAxis\_n**: The major elliptical semiaxis torsion axis ('x', 'y' or 'z') from which the torsion angle  $\phi_1$  is defined (n-th Thomson laser beam).
- **thsc\_laserTargetSAMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the laser elliptical target area (n-th Thomson laser beam).
- **thsc\_laserTargetSAMinor\_n**: The minor (smallest) semiaxis length  $\ell_2$  for the laser elliptical target area (n-th Thomson laser beam).

- **thsc\_laserTargetX\_n**: The x-component of the laser global target center position vector  $\mathbf{T}$  (n-th Thomson laser beam).
- **thsc\_laserTargetY\_n**: The y-component of the laser global target center position vector  $\mathbf{T}$  (n-th Thomson laser beam).
- **thsc\_laserTargetZ\_n**: The z-component of the laser global target center position vector  $\mathbf{T}$  (n-th Thomson laser beam).
- **thsc\_laserWavelength\_n**: The wavelength (in nm) of the Thomson laser (n-th Thomson laser beam).

### 30.1.3.3 Thomson Detectors Runtime Parameters

- **thsc\_numberOfDetectors**: The number of Thomson detectors that are going to be used.
- **thsc\_detectorLaserBeamNumber\_n**: The laser beam identification number associated with the detector (n-th Thomson detector).
- **thsc\_detectorLensSAMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the detector elliptical lens area (n-th Thomson detector).
- **thsc\_detectorLensX\_n**: The x-component of the detector global lens center position vector  $\mathbf{L}$  (n-th Thomson detector).
- **thsc\_detectorLensY\_n**: The y-component of the detector global lens center position vector  $\mathbf{L}$  (n-th Thomson detector).
- **thsc\_detectorLensZ\_n**: The z-component of the detector global lens center position vector  $\mathbf{L}$  (n-th Thomson detector).
- **thsc\_detectorNoRayDeflection\_n**: If set true, the complete set of laser and detector rays will not be deflected, i.e. all rays will be straight lines. As the **ThomsonScattering** unit is detector driven, the detectors decide over the entire ray paths and transmit this info to the associated laser (n-th Thomson detector).
- **thsc\_detectorNoRayPowerLoss\_n**: If set true, the complete set of laser and detector rays will not experience power loss as they traverse the domain. Useful for comparisons and to see the effect of the domain on the Thomson spectra (n-th Thomson detector).
- **thsc\_detectorNumberOfRays\_n**: Number of statistical Thomson detector rays to be created for the Thomson detector (n-th Thomson detector).
- **thsc\_detectorNumberOfSpecPoints\_n**: Number of spectral points to be recorded on the Thomson detector (n-th Thomson detector).
- **thsc\_detectorSAMajorTorsionAngle\_n**: The major elliptical semiaxis torsion angle  $\phi_1$  along the Thomson detector's lens target center line (n-th Thomson detector).
- **thsc\_detectorSAMajorTorsionAxis\_n**: The major elliptical semiaxis torsion axis ('x', 'y' or 'z') from which the torsion angle  $\phi_1$  is defined (n-th Thomson detector).
- **thsc\_detectorScreenSpectralFlux\_n**: If set true, the output (y-axis) on the detector will be  $dP_s/d\omega_s$  as in Eq.(30.1). If false, the output is in power  $\Delta P_s$ , which is  $dP_s/d\omega_s$  multiplied by the corresponding bin size  $\Delta\omega_s$ . One has to be aware that  $\Delta P_s$  can sometimes be grossly overestimated, particularly for very narrow Thomson spectra with small widths. This happens if the recording wavelength window is too broadly set (n-th Thomson detector).
- **thsc\_detectorScreenWavelengths\_n**: If set true, the horizontal output (x-axis, range of wavelengths or frequencies) on the detector will be given in wavelengths. If false, the horizontal output is in frequencies. The wavelength output will have equally spaced ticks, while the frequency output will not (n-th Thomson detector).

- **thsc\_detectorSkipTimeResolve\_n**: If set true, the additional time-resolved Thomson spectra files will not be produced. Only the total Thomson spectra (i.e. without simulation time label) will be on disk after the simulation. The option of only time-resolved Thomson spectra is not available, meaning that the total Thomson spectra will always be produced (n-th Thomson detector).
- **thsc\_detectorTargetSAMajor\_n**: The major (largest) semiaxis length  $\ell_1$  for the detector elliptical target area (n-th Thomson detector).
- **thsc\_detectorTargetSAMinor\_n**: The minor (smallest) semiaxis length  $\ell_2$  for the detector elliptical target area (n-th Thomson detector).
- **thsc\_detectorTargetX\_n**: The x-component of the detector global target center position vector  $\mathbf{T}$  (n-th Thomson detector).
- **thsc\_detectorTargetY\_n**: The y-component of the detector global target center position vector  $\mathbf{T}$  (n-th Thomson detector).
- **thsc\_detectorTargetZ\_n**: The z-component of the detector global target center position vector  $\mathbf{T}$  (n-th Thomson detector).
- **thsc\_detectorUseRelativityTerm\_n**: If set true, an additional relativistic term as stated in Eq.(30.7) will be multiplicatively added to the scattered power equation (30.1). One must be aware, however, that this relativistic term was derived for very special scattering setups: incident laser electric field  $\mathbf{E}_i$  is polarized and perpendicular to the scattering plane and a polarizer is used at the detector to select the scattered radiation electric field  $\mathbf{E}_s$  to be parallel to  $\mathbf{E}_i$ . Unless the experimental setup to be simulated does correspond exactly to this situation, it is best to avoid this term (n-th Thomson detector).
- **thsc\_detectorWavelengthMax\_n**: The maximum recording wavelength (in nm) of the Thomson detector (n-th Thomson detector).
- **thsc\_detectorWavelengthMin\_n**: The minimum recording wavelength (in nm) of the Thomson detector (n-th Thomson detector).

#### 30.1.3.4 Thomson Scattering General Runtime Parameters

- **thsc\_cellTimeEnergyDeposition**: If set true, the energy deposition by the Thomson laser rays is calculated based only on the time spent in each cell by calculating cell center values of  $\ln \Lambda$  and  $\nu_{ib}$  using Eq.(18.46) and replacing the time integration term in Eq.(18.47) by  $\nu_{ib}t$ , i.e. treating  $\nu_{ib}$  as constant in the cell. If false, the more expensive time integration formalism from Eq.(18.59) is used.
- **thsc\_cellWallThicknessFactor**: Controls the (imaginary) thickness of the cell walls to ensure computational stability of the Thomson scattering ray tracing algorithm. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to a default of  $10^{-6}$  and should only very rarely be changed.
- **thsc\_detectorWriteFormatX**: Format string for writing out the x-axis data (wavelengths or frequencies) on the detector screens (default 'es20.10').
- **thsc\_detectorWriteFormatY**: Format string for writing out the y-axis data ( $dP_s/d\omega_s$  or  $\Delta P_s$ ) on the detector screens (default 'es20.10').
- **thsc\_enforcePositiveNele**: (AVG algorithm 18.4.4.1) If true, the x-, y- and z-components of the gradients  $\langle \nabla n_e \rangle$  will be rescaled such that they always deliver a positive (greater or equal zero) value for the number of electrons in a cell.
- **thsc\_enforcePositiveTele**: (AVG algorithm 18.4.4.1) If true, the x-, y- and z-components of the gradients  $\langle \nabla T_e \rangle$  will be rescaled such that they always deliver a positive (greater or equal zero) value for the electron temperature in a cell.



- **thsc\_maxRayCount**: The maximum number of laser/detector rays that can be created on one processor. For the Thomson scattering this number must be at least as large as the maximum over all individual laser and detector sets of rays, i.e. always the entire set of either the laser or the detector rays for each possible laser/detector pair must fit into memory.
- **thsc\_printEnergyInfo**: If true, it prints info about the Thomson laser energy entering and leaving the domain (cumulative and at each time step) to a file with name <basename>ThscEnergyProfile.dat, where <basename> is the base name of the simulation.
- **thsc\_printLaserBeams**: If true, it prints detailed information about the Thomson laser beams to a file with name <basename>ThscLaserBeamsPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printDetectors**: If true, it prints detailed information about the Thomson detectors to a file with name <basename>ThscDetectorsPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printMain**: If true, it prints general information regarding the Thomson scattering setup to a file with name <basename>ThscMainPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printPulses**: If true, it prints detailed information about the Thomson laser pulses to a file with name <basename>ThscLaserPulsesPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printRays**: If true, it prints detailed information about all rays initially generated on each processor to a file(s) with name(s) <basename><fileLabel><PID>.txt, where <basename> is the base name of the simulation, <fileLabel> is a file label name given by the user and PID is the processor rank number. This runtime parameter should only be used for debugging purposes (default = false) by developers of the code.
- **thsc\_printSpecies**: If true, it prints detailed information about the species present in the simulation to a file with name <basename>ThscSpecies.txt, where <basename> is the base name of the simulation.
- **thsc\_rayDeterminism**: If true, the Grid Unit will be forced to use the Sieve Algorithm to move the ray particle data. Forcing this algorithm will result in a slower movement of data, but will fix the order the processors pass data and eliminate round off differences in consecutive runs.
- **thsc\_rayZeroPower**: Below this value (erg/s), the Thomson laser ray is considered to have zero power.
- **useThomsonScattering**: If false, the Thomson scattering is not activated, even if the code was compiled to do so. Bypasses the need to rebuild the code.
- **threadThscRayTrace**: If true, in the innermost ray loop, tracing of all rays through a block is threaded. This runtime parameter can only be set during setup of the code.



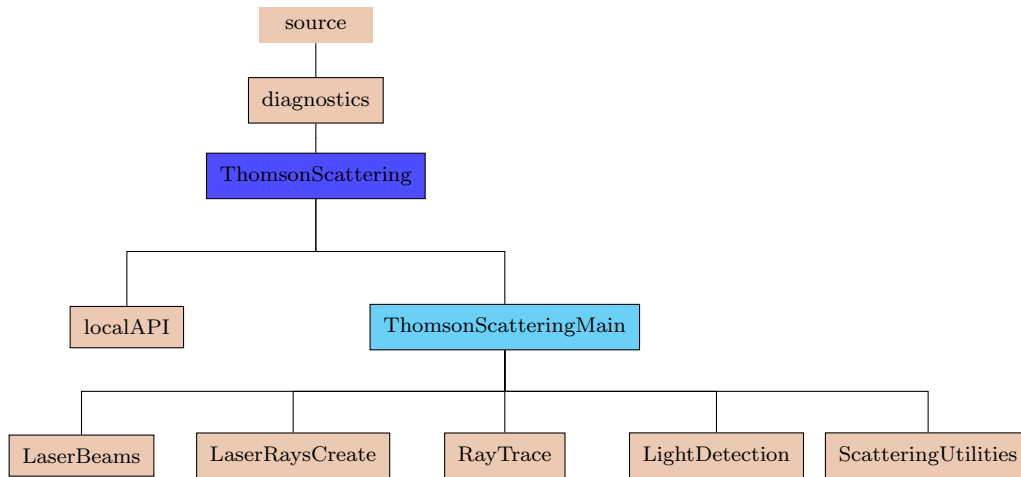


Figure 30.2: The simple `ThomsonScattering` unit with no ray tracing directory tree.

## 30.2 Simple Thomson Scattering without Ray Tracing

As of FLASH4.5, we provide the a “Simple” implementation of Thomson Scattering. In the future, an implementation that employs true ray tracing may be provided.

### 30.2.1 Ray Tracing in the Geometric Optics Limit

In the “simple” implementation, We do not “trace” rays but “cast” them: “rays” consist of two straight-line parts, which we’ll refer to as “leg 1” and “leg 2”.

### 30.2.2 Laser Power Deposition

The power deposited in an experiment by a probe laser may significantly alter the behavior of the physical system, by deposition energy into the system. This is usually an undesired effect. The `ThomsonScattering` unit does not model this effect. As a purely diagnostic unit, it has no impact on the dynamics of the physical system. However, a `ThomsonScattering` configuration can easily be combined with a configuration of a laser for energy deposition, provided by the `EnergyDeposition` unit, see [Section 18.4](#). Commonly this configuration will consist of adding a laser beam, in addition to the beams already present in an experimental setup. The definitions of pulse, beam parameters, *etc.*, for this additional beams should closely mirror the runtime parameters chosen to configure the Thomson probe laser beam.

### 30.2.3 Laser Energy Density

The `ThomsonScattering` unit will fill several variables with information about the Thomson probe laser power and its spatial distribution, if variables of the following names are defined:

- `pwin` - Power in, restricted to a “measurement region” that is used to calibrate the configured laser power.
- `pwia` - Power in, everywhere except in locations that are forbidden because the critical electron density is exceeded.
- `pwil` - Power in, with leg 1 attenuation applied, computed everywhere in the domain. **Note that this can increase run time very significantly**, since integrations along many more ray paths have to be performed than what is needed for computing scattering spectra.

- `pwi2` - Power in, with leg 1 and leg 2 attenuation applied. Only computed in regions within the cone defined by a detector's opening angle.
- `pwde` - Total power of detectable scattered light, without attenuation.
- `pwi1` - Total power of detectable scattered light, with leg 1 attenuation applied.
- `pwi2` - Total power of detectable scattered light, with leg 1 and leg 2 attenuation applied.

Laser Energy Density for the probe laser can be made available for visualization in the `lase` variable, if an `EnergyDeposition` beam has been added to the configuration, as described in [Section 18.4.11.5](#).

### 30.2.4 Algorithmic Implementations of the Ray Tracing

As mentioned above, we provide the a “Simple” implementation of Thomson Scattering. Even though the code uses “raytracing” terminology in some variable names *etc.*, really we are “casting” rays along predetermined paths, rather than letting each ray find its own path through set-by-step integration.

### 30.2.5 Setting up the Probe Laser Pulse

We use runtime parameters analogous to those of the `EnergyDeposition` unit to configure the time behavior of the `ThomsonScattering` unit. Computations are only performed by this code unit at times when the probe laser pulse is nonzero. Additionally, a time-dependent probe laser power will directly affect the absolute scaling of computed spectro, so it can be useful for comparing a sequence of spectra computed at different times.

The laser's pulse contains information about the laser's energy profile and can thus be characterized by a function of laser power in terms of time  $P(t)$ . The curve of  $P(t)$  gives the power of the laser pulse at any given simulation time. In FLASH, the form of  $P(t)$  is given by a series of points, which are connected via straight lines (see [Figure 18.5](#)).

Note, that  $t_1$  is the time the laser first comes into existence. In [Figure 18.5](#) there is therefore no line connecting the origin (zero power) with  $P_1$ . If the user wants a gradual buildup of laser power from zero, an explicit point  $P_1 = 0$  must be given at time  $t_1$ . The same consideration applies for the other end of the timescale.

Runtime parameters `thsc.spectrumFileIntervalStep` and `thsc.spectrumFileIntervalTime` can be used to limit the frequency of scattering computations, so that they won't happen at the end of every time step while the probe laser is on.

### 30.2.6 Setting up the Laser Beam

**Note:** currently only one probe beam is supported.

The setup of each laser beam follows closely the setup of the laser beams in [18.4.6](#) for energy-depositing lasers. While the implementation of rays is quite different, we aim to make setting up a Thomson scattering probe laser beam very similar to setting up a beam for laser energy deposition. Lens and target coordinates are used for directing a beam. Additional runtime parameters needed for each beam are: 1) the launching time (the beam fires once the simulation time exceeds the launching time), 2) its target detector screen and 3) the beam power. For the `ThomsonScattering` unit, lens and target ellipses can both be fully or partially inside or outside the simulation volume.

### 30.2.7 Creating the Rays

The rays inside each beam are created by connecting one-to-one grid points inside the lens area with the subcell center points inside the interaction region. The ray directions inside each beam can be parallel or conically diverging or converging collinear and are not allowed to cross. The lens positions and directions at the lens are calculated using the same strategy as presented in [18.4.7.6](#). Rays missing the domain can either be simply ignored or directly recorded on the detector screen.

### 30.2.7.1 Ray Specifications

Each beam/disk ray needs to know its position  $x, y, z$  and velocity  $v_x, v_y, v_z$  as it traverses the domain. In addition to these 6 components it needs to know: 1) the time spent travelling in domain, 2) the current block and processor numbers, 3) a global identification tag, 4) the originating beam and target detector numbers. Screen rays on the other hand are only defined on the detector screen and hence need no specific information for domain travelling. Their information is reduced to a screen position coordinate pair  $x, y$  and a detector number to which detector they are associated with.

Additionally, upon request, the Thomson scattering code is able to calculate extra magnetic diagnostic quantities  $\mathbf{k}$  and  $J$  for each beam/disk and screen ray:

$$\mathbf{k} = \int \mathbf{B} d\ell, \quad (30.8)$$

$$J = \int \mathbf{v}_n \cdot (\nabla \times \mathbf{B}) d\ell, \quad (30.9)$$

where  $\int d\ell$  denotes path integration,  $\mathbf{B}$  is the magnetic field vector and  $\mathbf{v}_n$  is the unit normal velocity vector along the ray path. Transforming the path integrals into time integrals using the relation  $d\ell = |\mathbf{v}(t)|dt$ , we get the differentials:

$$\frac{d\mathbf{k}}{dt} = \mathbf{B}|\mathbf{v}(t)|, \quad (30.10)$$

$$\frac{dJ}{dt} = \mathbf{v}(t) \cdot (\nabla \times \mathbf{B}). \quad (30.11)$$

### 30.2.8 Setting up the Detectors

**Note:** currently only one detector is supported.

Each detector is defined as a point together with a cone of directions from which it can receive light, the latter defined by a direction and an opening angle. Thus the geometry is specified by three components: 1) the detector location  $\mathbf{C}$ , 2) the unit vector  $\mathbf{n}$  of the direction of the cone axis, and 3) the angle  $\alpha$ .

#### 30.2.8.1 Recording Rays at the Detector

Each ray that reaches the detector gets recorded. Two kinds of files can be generated at the end of ThomsonScattering processing:

1. a detector file, and/or
2. a spectrum file.

Runtime parameters `thsc_spectrumFileIntervalStep` and `thsc_spectrumFileIntervalTime` determine approximately how often these files are generated. This is only approximate because these runtime parameters do not modify the simulation time step. Currently, generation of one kind of these files may force generation of the other one at the same time. Each detector file is currently an ascii file (formatted) and named:

$$\langle \text{basename} \rangle \text{ThomsonDetectorFile} \langle \text{detectorID} \rangle \_ \langle \text{timestamp} \rangle \quad (30.12)$$

where  $\langle \text{basename} \rangle$  is the name of the simulation,  $\langle \text{detectorID} \rangle$  is the detector number (currently limited to the range [01-99]) and (optionally)  $\langle \text{timestamp} \rangle$  is the simulation time when the file was created. Only the  $(x, y)$  pairs are written (first  $x$  followed by  $y$  on each line) without any headers. Thus the file can be directly incorporated and read by other software for graphical output (see for example the proton imaging ASCII  $\rightarrow$  PGM greyscale converter). Each spectrum file is also a text file and named:

$$\langle \text{basename} \rangle \text{ThomsonSpectrumFile} \langle \text{detectorID} \rangle \_ \langle \text{timestamp} \rangle \quad (30.13)$$

where  $\langle \text{basename} \rangle$  is the name of the simulation,  $\langle \text{detectorID} \rangle$  is the detector number (currently limited to the range [01-99]) and (optionally)  $\langle \text{timestamp} \rangle$  is the simulation time when the file was created. The first two columns are wave length and detected intensity, respectively. There will normally be `thsc_spectrumNpts_N` lines.

### 30.2.9 Usage

To include the use of the Thomson Scattering unit, something like the following can be included into the setup command line:

```
-unit=diagnostics/ThomsonScattering [thsc_maxBeams=<number> thsc_maxDetectors=<number> threadThomsonSc=
```

The default settings are: maximum number of beams = 6, maximum number of detectors = 1 and no use of threading during tracing of rays through the domain, which can be changed by the following three setup variables:

- **thsc\_maxBeams**: The maximum number ray beams for the simulation.
- **thsc\_maxDetectors**: The maximum number of Thomson scattering detectors for the simulation.
- **threadThomsonSc**: Enables threading during ray domain tracing.

Using Thomson scattering runtime parameters in the flash.par file, the user can set up the proper ray imaging configuration for his specific needs. The following list of runtime parameters is currently available for the user.

#### 30.2.9.1 Thomson Scattering Beams Runtime Parameters

The following are some runtime parameters for the Thomson scattering beams. The `_n` at the end of each runtime parameter characterizes the beam number, hence replace `_n` by `_1` for the first beam, `_n` by `_2` for the second beam, etc.

See [Beam Runtime Parameters](#) for a more complete list of beam parameters.

- **thsc\_numberOfBeams**: The number of Thomson scattering beams that are going to be used.
- **thsc\_beamLens[X,Y,Z]\_n**: The global 3D components of the lens center position vector **C**.
- **thsc\_beamTarget[X,Y,Z]\_n**: The global 3D components of the target position vector **T**.
- **thsc\_beamDetector\_n**: The detector number, where the rays of the beam should be recorded. This should be set to 1 as long as only one detector is used.
- **thsc\_maxRayCount**: Maximum total number of rays. Rays may be dropped if this number is exceeded.
- 

**Note:** only configurations with one beam have been tested as of FLASH4.5.

#### 30.2.9.2 Thomson Scattering Detectors Runtime Parameters

The following are some runtime parameters for the Thomson scattering detectors. In most of them, the `_n` at the end of each runtime parameter characterizes the detector number, usually “1”. See [Detector Runtime Parameters](#) for a more complete list of detector parameters.

- **thsc\_numberOfDetectors**: The number of Thomson scattering detectors that are going to be used.
- **thsc\_detectorCenter[X,Y,Z]\_n**: The global 3D components of the detector center position vector **C**.
- **thsc\_detectorTarget[X,Y,Z]\_n**: The local 3D components of the center of the target area to which the detector is oriented.
- **thsc\_detectorTargetRadius\_n**: The radius of a circular area perpendicular to the detector–target connecting line, from which the detector receives scattered radiation. This is converted to an equivalent opening angle and used if **thsc\_detectorApertureAngle\_n** is not specified.
- **thsc\_detectorApertureAngle\_n**: The conical full aperture angle, specified directly.
- **thsc\_detectorSideLength\_n**: The side length (cm) of the detector square screen.

**Note:** currently only one detector is supported.

### 30.2.9.3 Thomson Scattering General Runtime Parameters

- **thsc\_detectorFileNameTimeStamp**: If set to true, a time stamp will be added to each detector file name. This allows for time splitting of detectors.
- **thsc\_detectorXYwriteFormat**: Controls formatted ascii output of the ray  $(x, y)$  pairs on the detector screens (default 'es20.10').
- **thsc\_detectorDGwriteFormat**: Controls formatted ascii output of the magnetic diagnostic quantities  $k_x, k_y, k_z, J$  on the detector screens (default 'es15.5').
- **thsc\_maxRayCount**: The maximum number of beam/disk rays that can be created/read-in on one processor per batch in the domain.
- **thsc\_printPulses**: If true, it prints detailed information about the laser pulses to a file with name <basename>ThomsonPulsesPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printBeams**: If true, it prints detailed information about the ray beams to a file with name <basename>ThomsonBeamsPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printDetectors**: If true, it prints detailed information about the ray detectors to a file with name <basename>ThomsonDetectorsPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_printMain**: If true, it prints general information regarding the ray imaging setup to a file with name <basename>ThomsonScatteringMainPrint.txt, where <basename> is the base name of the simulation.
- **thsc\_recordOffScreenRays**: If true, rays not hitting their target detector screen are also recorded on the detector files. These rays will have screen coordinate pairs  $(x, y) \notin [0, 1]$ .
- **thsc\_screenRayBucketSize**: Sets the bucket size for flushing screen rays out to disk.
- **thsc\_screenRayDiagnostics**: If true, the magnetic diagnostic quantities  $k_x, k_y, k_z, J$  are evaluated for each ray and recorded on the detector screens.
- **useThomsonScattering**: If false, no Thomson scattering will be performed, even if the code was compiled to do so. Bypasses the need to rebuild the code.

### 30.2.10 Example Setup

A version of the laser Slab setup with an added Thomson scattering diagnostic is provided, see [Section 35.7.6](#) for a description.



# Chapter 31

## X-ray Imaging Unit

The **X-ray Imaging** unit obtains an X-ray image of the domain at specific simulation time(s) using the main driver function **XrayImaging**. The implementation is based on sending one X-ray per detector screen resolution pixel through the domain and record it's intensity reduction due to attenuation in the domain media. Currently only cold atomic opacities are being used to determine the combined opacity of each cell's materials. Each detector can record X-ray intensities for a mix of X-ray energies, thus each X-ray for each detector in the code represents a bundle of X-rays with predefined distributions of X-ray energies. Each detector is also given an X-ray point origin from which the X-rays originate.

### 31.0.1 X-ray Paths through Domain

All X-rays travel in straight lines through the domain. Each X-ray is given a dimensionless 3D unit direction vector  $\mathbf{u}$ ,  $|\mathbf{u}| = 1$  with components  $(u_x, u_y, u_z)$  pointing from the detector screen to its X-ray origin. Each direction vector component is a direct measure of how much 3D distance the X-ray travels between two such component points. For example, the direction vector component  $u_x$  can be used to evaluate the 3D distance  $d$  covered by the X-ray between two points  $\mathbf{p}_1$  and  $\mathbf{p}_2$  by using their components  $p_{1x}$  and  $p_{2x}$ :

$$d = |(p_{1x} - p_{2x})/u_x|, \quad (31.1)$$

where, when removing the absolute value bars, we can judge, if the X-ray actually travelled that distance (positive  $d$ ) or not (negative  $d$ ), depending on the location of  $p_{1x}$  and  $p_{2x}$  and the sign of  $u_x$ . For 3D cartesian domains it is therefore straightforward to find the minimum distance travelled through a cell by choosing the minimum of all positive distances that are obtained from the above Eq.(31.1) when replacing  $p_{1x}$  and  $p_{2x}$  by the 3D cartesian cell bounding box component pairs. Once  $d$  is found, we can use it to find the new X-ray's location  $\mathbf{p}_{new}$  on the cell's bounding box from its old location  $\mathbf{p}_{old}$ :

$$\mathbf{p}_{new} = \mathbf{p}_{old} + d * \mathbf{u}. \quad (31.2)$$

This simple procedure for 3D cartesian domain geometries changes for 3D X-ray tracing in 2D cylindrical domain geometries. While the axial z-component is identical to the 3D cartesian case, the curved mantle in the radial space makes it necessary to solve quadratic equations. Eq.(31.2) can be seen as the parametric line equation of a 3D X-ray. We only look at the x- and y-components of the line equation and add their squares together:

$$r_{new}^2 = r_{old}^2 + 2(p_{old,x}u_x + p_{old,y}u_y)d + (u_x^2 + u_y^2)d^2, \quad (31.3)$$

where

$$r_{old}^2 = p_{old,x}^2 + p_{old,y}^2 \quad (31.4)$$

$$r_{new}^2 = p_{new,x}^2 + p_{new,y}^2 \quad (31.5)$$

are the old and new radial 2D cylindrical positions of the 3D X-ray. The quadratic equation in  $d$  is solved for positive  $d$  by inserting the 2D cylindrical cell radial bounding box pair  $(r_{min}, r_{max})$ . Due to the necessity

of solving quadratics, the computational accuracy attainable for the radial  $d$  can be lower than for the axial  $d$ , which is solved by using the linear Eq.(31.1). Instead of forcing the quadratic  $d$  solution to be as accurate as the linear (axial)  $d$  solution (for example by solving the quadratic in higher precision), the linear X-ray propagation through the cell is allowed to take place using more than one step (at most two steps are ever required). The 3D X-ray tracing in a 2D cylindrical domain also requires constant adjustments between its 3D x- and y-position components and its 2D radial component, as for example cell nudging of the X-ray must occur in the 2D cylindrical radial space. One cannot allow the X-ray's x- and y-position components to get out of sync with the radial component, as all of them are needed to solve Eq.(31.3).

### 31.0.2 Creating and Tracing the X-rays

Each pixel center coordinate on the detector screen is the starting point of an X-ray. It's trajectory is a straight line from the pixel center to the X-ray point origin of the detector screen. The X-ray's direction vector  $\mathbf{u}$  is pointing from the pixel center to the origin. Initial domain entry points for each X-ray are determined using their  $\mathbf{u}$  and Eq.(31.2), where  $\mathbf{p}_{old}$  is the X-ray's global pixel center coordinates and  $\mathbf{p}_{new}$  the domain boundaries. If an X-ray misses the domain, it is recorded directly as a screen X-ray with zero attenuation. There is no screen miss possibility for the X-ray, because per construction each X-ray belongs to a pixel on the detector screen. If the X-ray hits the domain, its entry position and direction vector are stored to prepare for X-ray tracing through the domain, as well as it's initial attenuation value is set to zero. Besides position, direction and attenuation information, each domain X-ray needs additional data to help with its movement through the domain and its transformation to a screen X-ray, such as current block and processor numbers, a global tag number and the integer pixel pair label from where it originated on the detector screen. The screen X-rays will only possess the local screen coordinate pair, it's calculated relative intensity in the range  $[0, 1]$  and the detector screen number.

The mechanism for tracing the X-rays through the domain resembles the one used for tracing protons through the domain. X-rays are created and traced through a frozen domain at a particular time step, hence no time resolved X-ray tracing is needed. Batches of domain X-rays are generated and emptied into screen X-ray buckets once the X-rays leave the domain. An important difference when compared to proton imaging is that each batch of X-rays must correspond to one specific detector screen. The reason behind this is that calculation of the cell opacities depends on the X-ray energies used and have to be calculated for each detector separately. The X-ray energies hence influence the domain properties through material opacities, whereas the proton energies only have an influence on their initial velocities and this property will not change any domain values.

### 31.0.3 Setting up and Recording on the X-ray Detector Screens

The X-ray detector screens are identical in shape and specifications to the ones used for proton imaging (see section 28.0.4). The same parameters are used to specify size (sidelength), location (global position  $\mathbf{C}$  of the detector center) and orientation (screen normal vector  $\mathbf{n}$  and tilting axis and angle) in space. Additional parameters used specifically for X-ray detectors are: 1) global origin  $\mathbf{O}$  position as the source point for the X-rays, 2) activation time, indicating the simulation time when the X-rays should be recorded, 3) screen resolution  $R$ , which indicates how many pixels will be along each detector side (leading to a total of  $R^2$  pixels), 4) the energy distribution for each X-ray (finite set of X-ray energies and fractions of these X-ray energies adding up to 1) and 5) data related to each pixel size to facilitate recording of the X-rays on the screen. Recording the X-rays on the detector screen is much simpler than for protons, since the target pixel is already known. The X-rays integer pixel coordinate pair is converted to the local screen coordinate pair and the accumulated attenuation  $a$  is translated to it's relative screen intensity  $I_{rel}$ :

$$I_{rel} = I_{exit}/I_0 = I_0 e^{-a}/I_0 = e^{-a}, \quad (31.6)$$

where  $I_{exit}$  is the X-ray intensity when exiting the domain and  $I_0$  the initial X-ray intensity. Note, that currently the initial intensity needs not be supplied as part of the detector info, since only relative intensities are recorded. As for proton imaging, each X-ray detector file is set up as a formatted ascii file with name:

$$\langle \text{basename} \rangle \text{XrayDetectorFile} \langle \text{detectorID} \rangle \_ \langle \text{timestamp} \rangle \quad (31.7)$$



where <basename> is the simulation name, <detectorID> is the detector number (currently limited to the range [01-99]) and (optionally) <timestamp> is the simulation time when the file was created. Three quantities are written to the detector file for each X-ray: 1) the local screen coordinate pair  $(x, y)$  in the range  $[0, 1]$  and 2) it's relative intensity  $I_{rel}$ . Translation to a greyscale PGM picture can be done by using the supplied ASCII  $\rightarrow$  PGM greyscale converter software.

### 31.0.4 Usage

The following additions to the setup command activate the X-ray Imaging unit:

`+XrayImaging [xray_maxDetectors=<number> threadXrayTrace=True]`

The `+XrayImaging` shortcut handles all the logistics for properly including the `XrayImaging` unit. The setting options are:

- **xray\_maxDetectors**: The maximum number of X-ray imaging detectors for the simulation (default = 1).
- **xray\_maxEnergyLevels**: The maximum number of X-ray energy levels for a detector (default = 3).
- **threadXrayTrace**: Enables threading during X-ray domain tracing (default = false).

The following is a list of flash.par runtime parameters for the X-ray imaging unit.

#### 31.0.4.1 X-ray Imaging Detectors Runtime Parameters

The following are the runtime parameters for the X-ray imaging detectors. The `_n` at the end of each runtime parameter characterizes the detector number and needs to be replaced by `_1` for the first detector, `_n` by `_2` for the second detector, etc.

- **xray\_numberOfDetectors**: The number of X-ray imaging detectors that are going to be used.
- **xray\_detectorCenter[X,Y,Z]\_n**: The global 3D components of the detector center position vector  $\mathbf{C}$ .
- **xray\_detectorOrigin[X,Y,Z]\_n**: The global 3D components of the X-rays origin position vector  $\mathbf{O}$ .
- **xray\_detectorNormal[X,Y,Z]\_n**: The local 3D components of the detector normal vector  $\mathbf{n}$ .
- **xray\_detectorEnergyLevelCount\_n**: The number of X-ray energy levels.
- **xray\_detectorEnergy\_n\_Level\_i**: The  $i$ -th X-ray energy level in eV. There must be as many X-ray energy levels as the number of X-ray energy levels specified.
- **xray\_detectorEnergy\_n\_Fraction\_i**: The  $i$ -th X-ray energy level fraction contribution. There must be as many X-ray energy level fractions as the number of X-ray energy levels specified. All fractions must be in the range  $[0-1]$ , and they must add up to 1.
- **xray\_detectorPerpXrays\_n**: If this parameter is set true, the origin X-ray position will be ignored and all the X-rays will be hitting the screen in a perpendicular fashion, i.e. parallel to the screen's normal direction vector. This is to avoid having to specify origin positions mimicking an infinite distance away from the screen.
- **xray\_detectorResolution\_n**: The number of pixels along each side of the detector.
- **xray\_detectorSideLength\_n**: The side length (cm) of the detector square screen.
- **xray\_detectorSideTiltingAngle\_n**: The tilting angle (degrees) of two sides parallel to the detector screen unit axis  $\mathbf{u}_y$  with respect to the tilting axis.
- **xray\_detectorSideTiltingAxis\_n**: The global tilting axis ('x','y' or 'z') for the detector screen.
- **xray\_detectorAlignWRTorigin\_n**: A useful shortcut to orient the detector screen in such a way that the normal vector coincides with the origin / detector center line. Given the origin  $\mathbf{O}$  and the detector center  $\mathbf{C}$ , the normal vector  $\mathbf{n}$  is recalculated such that  $(\mathbf{O} - \mathbf{C}) \cdot \mathbf{n} = 0$ .

### 31.0.4.2 X-ray Imaging General Runtime Parameters

- **xray\_cellWallThicknessFactor**: Controls the (imaginary) thickness of the cell walls to ensure computational stability of the X-ray imaging code. The cell thickness is defined as this factor times the smallest cell dimension along all geometrical axes. The factor is currently set to  $10^{-6}$  and should only very rarely be changed.
- **xray\_3Din2D**: Indicates, whether this is a 3D X-ray in 2D cylindrical domain simulation.
- **xray\_detectorFileNameTimeStamp**: If set to true, a time stamp will be added to each detector file name. This allows for time splitting of detectors.
- **xray\_detectorXYwriteFormat**: Controls formatted ascii output of the screen X-ray local coordinate  $(x, y)$  pairs on the detector screens (default 'es20.10').
- **xray\_detectorDGwriteFormat**: Controls formatted ascii output of the screen X-ray relative intensities  $I_{rel}$  on the detector screens (default 'es15.5').
- **xray\_maxXrayCount**: The maximum number of X-rays that can be created on one processor per batch in the domain.
- **xray\_printDetectors**: If true, it prints detailed information about the X-ray detectors to a file with name <basename>XrayImagingDetectors.txt, where <basename> is the base name of the simulation.
- **xray\_printMain**: If true, it prints general information regarding the X-ray imaging setup to a file with name <basename>XrayImagingMainPrint.txt, where <basename> is the base name of the simulation.
- **xray\_printSpecies**: If true, it prints general information regarding the all the species present in the current simulation to a file named <basename>XrayImagingMainSpecies.txt, where <basename> is the base name of the simulation. It records the atomic element decomposition of each species with number and mass fractions and also records the cold opacities of each species for each detector (depending on the X-ray energy specified).
- **xray\_printXrays**: If true, it activates printout of domain X-ray data at specific stages during the domain X-ray tracing. This runtime parameter is meant solely for debugging purposes and calls to the corresponding X-ray printing routine can be placed anywhere in the X-ray imaging code. Each processor writes its own file(s) with name(s):

<basename>printXraysDetBatch<DetectorBatchNr>Proc<ProcessorID>\_<simulationTime>.txt,

where <basename> is the base name of the simulation. An optional .Tag<Tagnumber> can be added to these files to distinguish between printouts at different strategic places. Use of this feature is reserved ONLY for debugging purposes and is currently limited to 1000 detector batches and 1000 processors per time stamp. Users other than code developers should not activate this feature.

- **xray\_XrayDeterminism**: If true, the Grid Unit will be forced to use the sieve algorithm to move the X-ray particle data. Forcing this algorithm will result in a slower movement of data, but will fix the order the processors pass data and eliminate round off differences in consecutive runs.
- **xray\_screenXrayBucketSize**: Sets the bucket size for flushing screen X-rays out to disk.
- **useXrayImaging**: If false, no X-ray imaging will be performed, even if the code was compiled to do so. Bypasses the need to rebuild the code.
- **threadXrayTrace**: If true, X-ray tracing through a block is threaded. This runtime parameter can only be set during setup of the code.

### 31.0.5 Unit Test

The unit test supplied for the X-ray imaging unit is based on putting a sphere of pure iron inside a domain filled with pure hydrogen. The hydrogen opacity is enforced to be equal to zero, hence attenuation of the X-rays is solely due to the atomic opacity of iron in the sphere. For each type of domain (3D cartesian and 2D cylindrical), three detector screens, each with resolution  $R = 1000$ , are placed at three different positions in space. For the cubic 3D cartesian domain, the three screen positions are along a face center, edge center and vertex direction. For the 2D cylindrical domain, we also place the three screens along the radial, axial and cylinder edge of the implied 3D cylinder. Attenuation of the X-rays are recorded and compared to analytically obtained bounds.

#### 31.0.5.1 X-ray Attenuation inside a Sphere

Given a sphere with radius  $R$  and located with its center at  $\mathbf{C}$ , any X-ray with screen location  $\mathbf{S}$  and origin location  $\mathbf{O}$  will travel a distance  $d$  inside the sphere given by:

$$d = 2\sqrt{\max(0, R^2 - \sin^2 \phi * |CS|^2)}, \quad (31.8)$$

where  $|CS|$  is the distance between  $\mathbf{C}$  and  $\mathbf{S}$  and  $\phi$  is the angle between the vectors  $\overrightarrow{SC}$  and  $\overrightarrow{SO}$ . The sphere representation inside the domain is blocky in all three directions for 3D cartesian domains and blocky in the axial direction (stacked cylindrical disks) for 2D cylindrical domains. Hence Eq.(31.8) cannot be directly used to determine the exact distance travelled through the blocky spheres. However, one can define two extra spheres: 1) Maximum inner sphere, with radius  $R_{min}$ , which is the largest sphere that can be placed completely inside the blocky spheres and 2) Minimum outer sphere, with radius  $R_{max}$ , which is the smallest sphere that can contains the blocky spheres completely. Since the positions of the X-ray don't change,  $\phi$  and  $|CS|$  does not change and each  $R_{min}$  and  $R_{max}$  also define through Eq.(31.8) corresponding X-ray distances  $d_{min}$  and  $d_{max}$  travelled through the maximum inner and minimum outer spheres. We then have the following three inequalities

$$\begin{aligned} R_{min} &\leq R \leq R_{max} \\ d_{min} &\leq d \leq d_{max} \\ I_{min} &\geq I \geq I_{max}, \end{aligned} \quad (31.9)$$

where  $R, d$  and X-ray intensity  $I$  correspond to the blocky sphere quantities. Using  $d_{min}$  and  $d_{max}$ , upper and lower bounds for the X-ray attenuation can be given when crossing the domain, leading to X-ray intensities  $I_{min}$  and  $I_{max}$  due to the maximum inner and minimum outer spheres.

#### 31.0.5.2 FLASH Setup of the Unit Test

For the 3D cartesian unit test, we set up a cubic box with side length 10 cm. A pure iron sphere with density  $0.3 \text{ g/cm}^3$  and radius 5 cm is created at the center of the domain, using uniform refinement level 6 for the AMR grid. Detector screens with resolution  $R = 1000$  and of side length 25 cm are placed at the following 3 different places, such that the sphere center and the detector screen centers are 10 cm apart: 1) along the y face center direction (0,1,0), along the edge center direction (1,1,0) and along the vertex direction (1,1,1). The corresponding X-ray origins are all placed in the opposite directions with origin / screen center distance of 20 cm. All X-ray energies are set to 60 keV. The three intensities  $I$ ,  $I_{min}$ ,  $I_{max}$  of each X-ray from all three detectors are analyzed and the unit test is marked successful, if for all X-rays the intensity inequality in Eq.(31.9) is fulfilled.

The 2D cylindrical unit test is set up to match the 3D cartesian one. The 2D cylindrical domain has an axial dimension of 10 cm and a radius dimension of 5 cm. The implicit 3D sphere is placed in form of a semicircle with center at the domain location (radial,axial) = (0,5). All other sphere specifications (material and density) and detector specifications (side length 25 cm,  $R = 1000$ , detector center to detector origin distance = 20 cm, detector center to sphere center distance = 10cm) match the 3D cartesian unit test. The three detectors are placed as follows: 1) along the axial direction, 2) along the radial direction and 3) along the cylindrical edge direction. The 2D cylindrical unit test is successful, if all X-ray intensities are within the bounds mentioned previously.



## Part VIII

# Numerical Tools Units



## Chapter 32

# Interpolate Unit

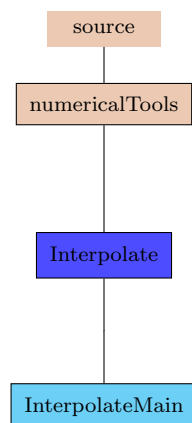


Figure 32.1: The `Interpolate` unit directory tree.

### 32.1 Introduction

The `Interpolate` unit supplied with FLASH4 contains a collection of interpolation utilities that can be applied to interpolate quantities on specific grids. It currently contains only cubic interpolation routines for 1D, 2D and 3D rectangular grids, based on piecewise (Hermite) cubic interpolation techniques.

### 32.2 Piecewise Cubic Interpolation

Piecewise cubic interpolation is a technique that can be used to create functional  $C^1$  surfaces (1st derivative continuous) throughout a grid domain by using functional and derivative information at each vertex of a grid cell. The domain function  $F$  and its derivatives  $dF$  are assumed to be known or at least calculable at each vertex. In order to show the essential features of the piecewise cubic interpolation method and to keep the formulas and matrix sizes manageable, we present the method for 2D domain geometries.

Consider a particular rectangular 2D cell. We wish to calculate a domain function  $F$  as a piecewise cubic polynomial in terms of  $[0, 1]$  rescaled variables  $x, y$  inside each cell:

$$F(x, y) = \sum_{i,j=0}^3 a_{ij} x^i y^j. \quad (32.1)$$

The  $[0, 1]$  rescaled variable  $x$  is obtained from the domain global variable  $X$  and the cell's lower  $X_\ell$  and upper  $X_u$  dimension as

$$x = \frac{X - X_\ell}{X_u - X_\ell}, \quad (32.2)$$

with a similar expression for  $y$ . From the expansion in 32.1, there are 16 expansion coefficients which need to be determined (fitted) to an appropriate set of data for specific points on the cell, which, in order to assure continuous representation of  $F$  over the entire domain, have to sit on the cell's boundaries. The most obvious choice for these points are the cell vertices, each vertex belonging to four cells. The data needed for each vertex are  $F$ ,  $dF/dx$ ,  $dF/dy$  and  $d^2F/dxdy$ . This leads to a linearly independent and rotationally invariant set of 16 data values, from which the 16 expansion coefficients can be uniquely determined. It can also be shown that for other dimensions only the inclusion of all mixed simple higher order derivatives leads to a linearly independent and rotationally invariant set of data values. For example, for rectangular 3D geometries the data needed at each of the 8 vertices of a cubic cell is  $F$ ,  $dF/dx$ ,  $dF/dy$ ,  $dF/dz$ ,  $d^2F/dxdy$ ,  $d^2F/dxdz$ ,  $d^2F/dydz$  and  $d^3F/dxdydz$ .

Let us now organize the 16 expansion coefficients  $a_{ij}$  into a vector  $\mathbf{a}$ , such that each element  $a(k)$  of  $\mathbf{a}$  is associated with the following expansion coefficient:

$$a(k) = a_{ij} \quad ; \quad k = 1 + i + 4j. \quad (32.3)$$

Likewise, we stack the data values of the four vertices  $v = 1, 2, 3, 4$  into a vector  $\mathbf{b}$  as follows:

$$\mathbf{b} = \begin{cases} F_v \\ (dF/dx)_v \\ (dF/dy)_v \\ (d^2F/dxdy)_v \end{cases} \quad ; \quad F_v = \begin{cases} F_1 \\ F_2 \\ F_3 \\ F_4 \end{cases} \quad ; \quad (dF/dx)_v = \begin{cases} (dF/dx)_1 \\ (dF/dx)_2 \\ (dF/dx)_3 \\ (dF/dx)_4 \end{cases} \quad ; \quad \text{etc...} \quad (32.4)$$

Let us associate the four vertex indices with the following four  $[0, 1]$  rescaled variables of the cell:

$$\begin{aligned} v = 1 & \longrightarrow (x, y) = (0, 0) \\ v = 2 & \longrightarrow (x, y) = (1, 0) \\ v = 3 & \longrightarrow (x, y) = (0, 1) \\ v = 4 & \longrightarrow (x, y) = (1, 1) \end{aligned} \quad (32.5)$$

Substituting these values for  $x$  and  $y$  into equation 32.1 and its derivatives, we can establish a connection between the expansion coefficient vector and the data vector in the form

$$\mathbf{b} = \mathbf{B}\mathbf{a}, \quad (32.6)$$

where the  $16 \times 16$  matrix  $\mathbf{B}$  has the following structure

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 0 & 2 & 4 & 6 & 0 & 3 & 6 & 9 \end{pmatrix}, \quad (32.7)$$



containing only positive integers and many zeros (175 out of 256 elements). Since we are ultimately interested in the expansion coefficients for the cell, we invert equation 32.6 and get

$$\mathbf{a} = \mathbf{B}^{-1}\mathbf{b}. \quad (32.8)$$

The inverse  $\mathbf{B}^{-1}$  is also an integer matrix and again contains many zero elements, although not as many as  $\mathbf{B}$  itself (156 out of 256 elements):

$$\mathbf{B}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 \\ -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 \\ 9 & -9 & -9 & 9 & 6 & 3 & -6 & -3 & 6 & -6 & 3 & -3 & 4 & 2 & 2 & 1 \\ -6 & 6 & 6 & -6 & -3 & -3 & 3 & 3 & -4 & 4 & -2 & 2 & -2 & -2 & -1 & -1 \\ 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ -6 & 6 & 6 & -6 & -4 & -2 & 4 & 2 & -3 & 3 & -3 & 3 & -2 & -1 & -2 & -1 \\ 4 & -4 & -4 & 4 & 2 & 2 & -2 & -2 & 2 & -2 & 2 & -2 & 1 & 1 & 1 & 1 \end{pmatrix} \quad (32.9)$$

The matrix  $\mathbf{B}^{-1}$  is universal for all 2D cells. Each cell has its specific data vector  $\mathbf{b}$  from which its expansion coefficient vector  $\mathbf{a}$  can be established via 32.8. Rather than storing  $\mathbf{B}^{-1}$  in matrix form and performing a direct matrix times vector operation for each cell,  $\mathbf{B}^{-1}$  is used implicitly when converting  $\mathbf{b}$  to  $\mathbf{a}$ . This avoids the redundant zero multiplications and is thus much more efficient. In fact, by using cleverly arranged reusable intermediates, this  $\mathbf{b}$  to  $\mathbf{a}$  transformation can be coded using only additions and subtractions and no multiplications at all.

Once the expansion coefficient vector  $\mathbf{a}$  for a cell has been established, equation 32.1 can be used to obtain the function value  $F$  at any point inside the cell, provided the  $[0, 1]$  rescaled coordinates of that point are given. The function values are given by Eq.(32.1), which can be computed efficiently using a double Horner scheme:

$$\tilde{a}_j = ((a_{3j}x + a_{2j})x + a_{1j})x + a_{0j} \quad (32.10)$$

$$F(x, y) = ((\tilde{a}_3y + \tilde{a}_2)y + \tilde{a}_1)y + \tilde{a}_0 \quad (32.11)$$

The evaluation of the global coordinate differentials is done using the chain rule and leads to:

$$\frac{d^{m+n}F(X, Y)}{dX^m dY^n} = (X_u - X_\ell)^{-m} (Y_u - Y_\ell)^{-n} \frac{d^{m+n}F(x, y)}{dx^m dy^n} \quad (32.12)$$

$$\frac{d^{m+n}F(x, y)}{dx^m dy^n} = \sum_{i=m}^3 \sum_{j=n}^3 (i)_m (j)_n a_{ij} x^{i-m} y^{j-n}, \quad (32.13)$$

where the Pochhammer symbol  $(i)_m$  is defined as  $(i)_m = i(i-1)\dots(i-m+1)$ .

## 32.3 Usage

To include the cubic interpolation tool into your FLASH development, add the line

```
REQUIRES numericalTools/Interpolate
```

into the Config file of the directory where the global API's of the interpolation unit will be used. Another way would be to include the option

-with-unit=numericalTools/Interpolate

into your command line when you configure the code with **setup**. Both ways will give you access to the following cubic interpolation tools in order of their application, where XX stand for the 3 rectangular domain geometries 1D,2D and 3D:

- **Interpolate\_cubicXXmonoDerv**: Handles the logistics for producing optimum sets of mixed vertex derivatives for a (sub)grid from vertex functional values to ensure monotonicity of the generated  $C^1$  cubic interpolation surfaces. Note that in order to generate these mixed derivatives the user needs to provide the vertex function values on a larger (sub)grid than the target (sub)grid: 1,2,3 extra vertex layers for 1D,2D,3D rectangular geometries.
- **Interpolate\_cubicXXcoeffs**: Calculate the cubic expansion coefficients  $a_{ij}$  in equation 32.1 from user provided sets of vertex data **b**, arranged as shown in 32.4. The calculation is performed in situ, i.e. the input vector **b** is overwritten by the cubic expansion coefficient vector **a** using equation 32.8. The routine accepts a collection of different **b** vectors and processes them all at once. For XX = 2D or 3D, the code allows for multithreading.
- **Interpolate\_cubicXXF(d1)(d2)**: For a specific cubic expansion coefficient vector **a** and a set of up to three  $[0, 1]$  rescaled coordinates  $x, y, z$ , these function routines calculate the functional value  $F$  using equation 32.1 and additionally the complete set of 1st ( $dx, dy, dz$ ) and 2nd ( $d^2x, d^2y, d^2z, dxdy, dxdz, dydz$ ) order  $[0, 1]$  rescaled coordinate derivatives using equation 32.13. Transformation to global coordinate derivatives via equation 32.12 is not done here and has to be done by the user in his specific application.

## Chapter 33

# Roots Unit

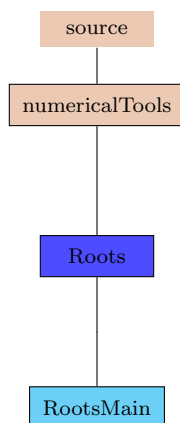


Figure 33.1: The **Roots** unit directory tree.

### 33.1 Introduction

The **Roots** unit supplied with FLASH4 aims at collecting all those utilities that find solutions (roots) to equations of the form  $f(x) = 0$ . Currently the **Roots** unit only contains routines to find all the roots of quadratic, cubic and quartic polynomials with real coefficients. Particular emphasis has been put on speed and stability of root evaluation.

### 33.2 Roots of Quadratic Polynomials

Here we solve

$$x^2 + a_1x + a_0 = 0, \quad (33.1)$$

where  $a_1$  and  $a_0$  are real. Well known techniques (larger magnitude root  $x_1$  computed by quadratic formula, smaller magnitude root computed by applying Vieta's rule:  $x_2 = a_0/x_1$ ) are used to ensure numerical accuracy of the roots. Additionally the code is equipped to deal with extremely large coefficients, such that computational overflow will not occur. A coefficient rescaling technique is applied if during evaluation of the quadratic formula either the term  $(a_1/2)^2$  or the discriminant  $(a_1/2)^2 - a_0$  become larger than the largest positive number on the machine.

### 33.3 Roots of Cubic Polynomials

This solves the cubic polynomial equation

$$x^3 + a_2x^2 + a_1x + a_0 = 0, \quad (33.2)$$

where  $a_2, a_1$  and  $a_0$  are real. Rather than calculating the roots by the analytical formulas, a special technique has been devised to pinpoint down one of the real roots by iteration. After rescaling the cubic polynomial coefficients such that  $-1 \leq a_2, a_1, a_0 \leq +1$  and classifying the rescaled cubic polynomials into 6 classes, an analysis of the real root(s) surfaces for each class enables one to find an optimum starting point for a fast convergent Newton-Raphson iteration. In rare cases when the Newton-Raphson iterations start oscillating around the root due to numerical rounding errors, a bisection iteration follows to get the root to within machine accuracy. Details of this procedure can be found in Flocke 2015. After one of the real roots has been found, the remaining roots (real or complex) are found by solving the residual quadratic, which is obtained by composite (forward/backward) deflation.

### 33.4 Roots of Quartic Polynomials

Solutions to the quartic polynomial equation

$$x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0, \quad (33.3)$$

where  $a_3, a_2, a_1$  and  $a_0$  are all real, are found again by Newton-Raphson and bisection iterations techniques, even for quartics possessing only complex roots. Please consult Flocke 2015 for the theoretical and computational details.

### 33.5 Usage

To include the root tools into your FLASH development, add the line

```
REQUIRES numericalTools/Roots
```

into the Config file of the directory where the global API's of the interpolation unit will be used. Another way would be to include the option

```
-with-unit=numericalTools/Roots
```

into your command line when you configure the code with **setup**. Both ways will give you access to the following root tools:

- **Roots\_x2Polynomial:** Finds all roots of a quadratic polynomial. The coefficient of the leading quadratic term must be equal to 1. For ordering of the roots, please consult the info in the header of this routine.
- **Roots\_x3Polynomial:** Finds all roots of a cubic polynomial. The coefficient of the leading cubic term must be equal to 1. For ordering of the roots, please consult the info in the header of this routine.
- **Roots\_x4Polynomial:** Finds all roots of a quartic polynomial. The coefficient of the leading quartic term must be equal to 1. For ordering of the roots, please consult the info in the header of this routine.

### 33.6 Unit Tests

There is currently a unit test for each kind of polynomial. They are all based on setting up the polynomial coefficients for known roots and checking the obtained root accuracies from the respective root solvers. The

unit tests try to stress test the root solvers by setting up polynomials which are hard to solve. Two measures are used for judging accuracy of the roots obtained. The relative accuracy of a root is defined as

$$\text{Relative root accuracy} = |(x_e - x_c)/x_e|, \quad (33.4)$$

where  $x_e$  is the exact analytical root and  $x_c$  the computational root. The absolute accuracy of a computational root  $x_c$  is

$$\text{Absolute root accuracy} = |x_c|. \quad (33.5)$$

For the unit tests we will use the relative root accuracy if  $x_e \neq 0$  and the absolute root accuracy otherwise.

### 33.6.1 Quadratic Polynomials Root Test

This test solves quadratic polynomials with extreme coefficients involving the largest positive number  $L$  and the smallest positive number  $S$  representable on the machine under the working precision (for double precision we would have  $L \approx 1.797... \times 10^{308}$  and  $S \approx 2.225... \times 10^{-308}$ ). All possible combinations are tested. We examine only the  $LL$  combination in detail. The four quadratic  $LL$  polynomials are:

$$x^2 \pm Lx \pm L = 0 \quad (33.6)$$

with the two roots:

$$x_1 = \mp L/2 + \sqrt{(L/2)^2 \mp L} \quad (33.7)$$

$$x_2 = \mp L/2 - \sqrt{(L/2)^2 \mp L}. \quad (33.8)$$

Since  $(L/2)^2$  is much larger than  $L$ , we can expand the square root in a Taylor series

$$\sqrt{(L/2)^2 \mp L} \approx L/2 \mp 1 + O[L^{-1}]. \quad (33.9)$$

The roots are approximated as

$$x_1 = \mp L/2 + L/2 \mp 1 + O[L^{-1}] \quad (33.10)$$

$$x_2 = \mp L/2 - L/2 \mp 1 + O[L^{-1}]. \quad (33.11)$$

However, a quantity like  $L + 1$  cannot be accurately represented on a machine due to limited mantissa space and will get converted to  $L$ . The following table shows the computational roots to be expected for the quadratic  $LL$  polynomials

$a_1$	$a_0$	$x_1$	$x_2$
$+L$	$+L$	$-1$	$-L$
$+L$	$-L$	$+1$	$-L$
$-L$	$+L$	$+L$	$+1$
$-L$	$-L$	$+L$	$-1$

Likewise, after going through the same kind of analysis for the remaining  $LS$ ,  $SL$  and  $SS$  polynomials, we can establish their computational roots. For the four  $SS$  polynomials we get (since  $\sqrt{S} \gg S$ ):

$a_1$	$a_0$	$x_1$	$x_2$
$+S$	$+S$	$+i\sqrt{S}$	$-i\sqrt{S}$
$+S$	$-S$	$+\sqrt{S}$	$-\sqrt{S}$
$-S$	$+S$	$+i\sqrt{S}$	$-i\sqrt{S}$
$-S$	$-S$	$+\sqrt{S}$	$-\sqrt{S}$

The four  $LS$  polynomials give (since  $S/L$  is not representable and hence equal to zero)

Table 33.1: Coefficients of the cubics I-IX used for the cubic polynomial root test.

Cubic	$a_2$	$a_1$	$a_0$
I	$-1.00000010000001e+14$	$+1.00000010000001e+21$	$-1.e+21$
II	$-3.000003$	$+3.000006000002$	$-1.000003000002$
III	$+8.999e+80$	$-1.0009e+161$	$+1.e+238$
IV	$+1.e+24$	$-1.0$	$-1.e+24$
V	$-1.99999999999999e+14$	$+0.99999999999998e+28$	$+1.e+28$
VI	$-3.e+5$	$+3.0000000001e+10$	$-1.0000000001e+15$
VII	$-3.0$	$+1.00000000000003e+14$	$-1.00000000000001e+14$
VIII	$-2.0000001e+7$	$+1.00000020000001e+14$	$-1.00000000000001e+14$
IX	$+0.99999999999998e+14$	$-1.99999999999998e+14$	$+2.e+14$

Table 33.2: Analytical roots and FLASH4 cubic solver accuracy performance of the cubics I-IX used for the cubic polynomial root test.

Cubic	Roots	Comments on Roots	Accuracy
I	$+10^{+14}, +10^{+7}, +1$	3 widely spaced real	$< 1.e-15$
II	$+1.000002, +1.000001, +1$	3 closely spaced real	$< 1.e-9$
III	$-10^{+81}, +10^{+80}, +10^{+77}$	3 large real	$< 1.e-15$
IV	$-10^{+24}, +1, -1$	1 large, 2 small real	$< 1.e-15$
V	$+10^{+14}, +10^{+14}, -1$	2 degenerate large, 1 small real	$< 1.e-15$
VI	$+10^{+5}, +10^{+5} \pm i$	3 closely spaced, small imag parts	$< 1.e-15$
VII	$+1, +1 \pm 10^{+7}i$	1 small real, 2 large complex, large imag parts	$< 1.e-15$
VIII	$+1, +10^{+7} \pm i$	1 small real, 2 large complex, small imag parts	$< 1.e-15$
IX	$-10^{+14}, +1 \pm i$	1 large real, 2 small complex	$< 1.e-15$

$a_1$	$a_0$	$x_1$	$x_2$
$+L$	$+S$	0	$-L$
$+L$	$-S$	0	$-L$
$-L$	$+S$	$+L$	0
$-L$	$-S$	$+L$	0

and the four  $SL$  polynomials have computational roots

$a_1$	$a_0$	$x_1$	$x_2$
$+S$	$+L$	$+i\sqrt{L}$	$-i\sqrt{L}$
$+S$	$-L$	$+\sqrt{L}$	$-\sqrt{L}$
$-S$	$+L$	$+i\sqrt{L}$	$-i\sqrt{L}$
$-S$	$-L$	$+\sqrt{L}$	$-\sqrt{L}$

The quadratic solver implemented in FLASH4 obtains all the above roots to within machine epsilon accuracy.

### 33.6.2 Cubic Polynomials Root Test

The following set of cubic polynomials I-IX is used for the test. It corresponds to the set used in xxx and represents difficult to solve cases that stress test the cubic solver. In Tables 33.1 and 33.2 we list all the coefficients, analytical roots and FLASH4 cubic solver accuracy performance of these polynomials. The polynomials were designed to be tested with double precision machine working precision.

Table 33.3: Coefficients of the quartics I-XIII used for the quartic polynomial root test.

Quartic	$a_3$	$a_2$	$a_1$	$a_0$
I	$-1.001001001e+9$	$+1.001002001001e+15$	$-1.001001001e+18$	$+1.e+18$
II	$-4.006$	$+6.018011$	$-4.018022006$	$+1.006011006$
III	$-9.98899e+79$	$-1.1008989e+157$	$-1.010999e+233$	$-1.e+307$
IV	$-1.00000000000002e+14$	$+1.9999999999999e+14$	$+1.00000000000002e+14$	$-2.e+14$
V	$+1.e+7$	$-2.00000000000001e+14$	$-1.e+7$	$+2.e+14$
VI	$-9.000002e+6$	$-0.9999981999998e+13$	$+1.9999982e+13$	$-2.e+13$
VII	$+2.000011e+6$	$+1.010022000028e+12$	$+1.1110056e+13$	$+2.828e+13$
VIII	$-1.00002011e+8$	$+2.01101022001e+11$	$-1.02200111000011e+14$	$+1.1000011e+15$
IX	$-2.0000002e+7$	$+1.01000040000005e+14$	$-2.020001e+14$	$+5.05e+14$
X	$-1.9986e+4$	$+1.00720058e+8$	$-1.8600979874e+10$	$+1.00004909000441e+14$
XI	$-4.006$	$+5.6008018e+1$	$-1.04148036024e+2$	$+6.75896068064016e+2$
XII	$-4.0e+3$	$+6.00001e+6$	$-4.00002e+9$	$+1.000010000009e+12$
XIII	$-6.0$	$+1.01000013e+8$	$-2.04000012e+8$	$+1.00000104000004e+14$

Table 33.4: Analytical roots and FLASH4 quartic solver accuracy performance of the quartics I-XIII used for the quartic polynomial root test.

Quartic	Roots	Comments on Roots	Accuracy
I	$+10^{+9}, +10^{+6}, +10^{+3}, +1$	4 widely spaced real	$< 1.e-15$
II	$+1.003, +1.002, +1.001, +1$	4 closely spaced real	$< 1.e-7$
III	$+10^{+80}, -10^{+77}, -10^{+76}, -10^{+74}$	4 large real	$< 1.e-15$
IV	$+10^{+14}, +2, +1, -1$	1 large, 3 small real	$< 1.e-15$
V	$-2 \times 10^{+7}, +10^{+7}, +1, -1$	2 large, 2 small real	$< 1.e-15$
VI	$+10^{+7}, -10^{+6}, +1 \pm i$	2 large real, 2 small complex	$< 1.e-15$
VII	$-7, -4, -10^{+6} \pm 10^{+5}i$	2 small real, 2 large complex	$< 1.e-15$
VIII	$+10^{+8}, +11, +10^{+3} \pm i$	1 large and small real, 2 medium complex	$< 1.e-15$
IX	$+10^{+7} \pm 10^{+6}i, +1 \pm 2i$	2 large, 2 small complex	$< 1.e-15$
X	$+10^{+4} \pm 3i, -7 \pm 10^{+3}i$	4 complex, mixed size real and imag parts	$< 1.e-14$
XI	$+1.002 \pm 4.998i, +1.001 \pm 5.001i$	4 closely spaced complex	$< 1.e-12$
XII	$+10^{+3} \pm 3i, +10^{+3} \pm i$	4 complex, equal real, small imag parts	$< 1.e-15$
XIII	$+2 \pm 10^{+4}i, +1 \pm 10^{+3}i$	4 complex, small real, large imag parts	$< 1.e-15$

### 33.6.3 Quartic Polynomials Root Test

Again the set of quartic polynomials I-XIII have been taken from xxx. Tables 33.3 and 33.4 list all the coefficients, analytical roots and FLASH4 quartic solver accuracy performance of these polynomials. The polynomials were designed to be tested with double precision machine working precision.





## Chapter 34

# RungeKutta Unit

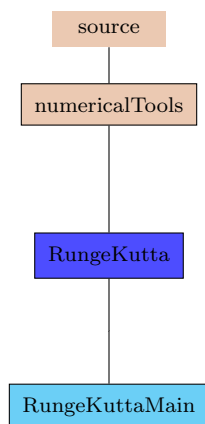


Figure 34.1: The `RungeKutta` unit directory tree.

### 34.1 Introduction

The `RungeKutta` unit in FLASH4 provides users with the tools to perform various kinds of (embedded) Runge Kutta (RK) iterations. The RK iterations are characterized by their implemented Butcher tableaus of various sizes and orders. The specific ODE to be solved must be transmitted via argument to the individual RK stepping routines. The RK routines contain no specifics of the ODE equations to be solved. This clean separation between the ODE function and the details of the RK iterations gives a universally applicable RK tool, which can be called from any FLASH4 application.

### 34.2 Runge Kutta Integration

The RK integration scheme starts from an initial set of first order ODE dependent variable values  $\mathbf{y}_0$  and advances the ODE solution  $\mathbf{y}$  through well defined steps in the independent variable  $x$ . Control of the dependent variable errors allows for optimum step size control. We describe here the embedded RK schemes, which use the same intermediate vectors  $\mathbf{k}$  to get an estimate of the error vector  $\mathbf{e}$ . An embedded RK step

can be represented by the following equations:

$$\frac{d}{dx}\mathbf{y}(x) = f(\mathbf{y}(x), x) \quad (34.1)$$

$$\mathbf{k}_i = hf(\mathbf{y}(x) + \sum_{j=1}^s a_{ij}\mathbf{k}_j, x + c_i h) \quad (34.2)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \sum_{i=1}^s b_i \mathbf{k}_i + O[h^{p+2}] \quad (34.3)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \sum_{i=1}^s b_i^* \mathbf{k}_i + O[h^{p+1}] \quad (34.4)$$

$$\mathbf{e} = \sum_{i=1}^s (b_i - b_i^*) \mathbf{k}_i + O[h^{p+1}] \quad (34.5)$$

Here  $h$  denotes the step size and  $s$  is the size of the embedded RK scheme. Each particular embedded RK scheme is characterized by the four sets of coefficients  $a_{ij}, b_i, b_i^*, c_i$ , which can be arranged into a table known as a Butcher tableau:

$c_1$	$a_{11}$	$a_{12}$	$\cdots$	$a_{1s}$
$c_2$	$a_{21}$	$a_{22}$	$\cdots$	$a_{2s}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$c_s$	$a_{s1}$	$a_{s2}$	$\cdots$	$a_{ss}$
	$b_1$	$b_2$	$\cdots$	$b_s$
	$b_1^*$	$b_2^*$	$\cdots$	$b_s^*$

The coefficients  $b$  and  $b^*$  correspond to RK methods of order  $p+1$  and  $p$ , from which the error vector is constructed via equation 34.5. If only the strict lower triangle of the  $a_{ij}$  values are non-zero, then the embedded RK scheme is explicit. Otherwise we have an implicit scheme. Explicit RK schemes are much easier to solve, because all intermediate  $\mathbf{k}$  vectors can be evaluated one after another using previously evaluated  $\mathbf{k}$  vectors. Implicit RK schemes on the other hand have to be solved in an iterative fashion and are much more expensive. FLASH4 currently has only explicit RK schemes implemented. Error control on each RK step is done by passing a maximum allowed absolute error vector  $\mathbf{e}_{max}$  corresponding to each of the dependent variables in  $\mathbf{y}$ . Step size control is currently implemented using information from the obtained error vector  $\mathbf{e}$  after the RK step and the maximum error vector  $\mathbf{e}_{max}$ . Using the fact that the error scales as  $O[h^{p+1}]$  due to the lower  $p$  order embedded RK method, the new step size  $h_{new}$  is adjusted from the old step size  $h_{old}$  as follows

$$h_{new} = h_{old} S \left( \max \left| \frac{\mathbf{e}_{max}}{\mathbf{e}} \right| \right)^{1/(p+1)} \quad (34.6)$$

and used as the next trial step size. A safety factor (typically  $S = 0.9$ ) is used to account for the fact that only the leading  $h^{p+1}$  term from  $O[h^{p+1}]$  was used in deriving this formula.

In some situations one needs an initial guess for the very first initial trial step size. This can be done from the supplied ODE vector  $\mathbf{y}$  and the maximum allowed error vector  $\mathbf{e}_{max}$  as follows. Consider the Taylor expansion of  $\mathbf{y}(x+h)$  around  $x$ :

$$\mathbf{y}(x+h) = \mathbf{y}(x) + h\mathbf{y}^{(1)}(x) + h^2\mathbf{y}^{(2)}(x)/2 + \cdots + h^p\mathbf{y}^{(p)}(x)/p! + O[h^{p+1}], \quad (34.7)$$

where  $\mathbf{y}^{(p)}$  denotes the  $p$ -th derivative of  $\mathbf{y}$  with respect to  $x$ . The expansion error  $O[h^{p+1}]$  can be directly related to the  $O[h^{p+1}]$  error of a RK method of order  $p$  (cf. equation 34.4). As an approximation one can equate the absolute value of the Taylor remainder term with the error goal of the RK method (i.e. plugging in formulas for both  $O[h^{p+1}]$  terms):

$$h^{p+1}|\mathbf{y}^{(p+1)}(x^*)|/(p+1)! = |\mathbf{e}_{max}|, \quad (34.8)$$

where  $x \leq x^* \leq x + h$ . Since we are dealing with an approximation, we further set  $x^* = x$  (this in effect approximates  $\mathbf{O}[h^{p+1}]$  by its leading term and ignores the  $> p + 1$  terms), leading to

$$h = \min \left( (p+1)! \left| \frac{\mathbf{e}_{max}}{\mathbf{y}^{(p+1)}(x)} \right| \right)^{1/p+1}. \quad (34.9)$$

We then solve for  $h$  and take the minimum to satisfy all components. To use this formula we need to evaluate the derivative. The following are second order central difference formulas based on equal spacing  $h$  (not to be confused with the RK step size) between functions:

$$\mathbf{y}^{(m+2k+\delta)}(x) = \frac{1}{h^{2k+\delta}} \sum_{n=-(k+\delta)}^{k+\delta} (-1)^{k+n-\delta} \frac{n^\delta (2k+\delta)!}{(k+n+\delta)!(k-n+\delta)!} \mathbf{y}^{(m)}(x+nh) + \mathbf{R}[h^2], \quad (34.10)$$

where the remainder term is equal to

$$\mathbf{R}[h^2] = -\frac{k+2\delta}{12} h^2 \mathbf{y}^{(m+2k+2+\delta)}(x^*) \quad (34.11)$$

with  $x - (k+\delta)h \leq x^* \leq x + (k+\delta)h$  and  $\delta = 0, 1$ , depending on if one wants even or odd higher order derivatives. For  $m = 0$  the derivatives are given in terms of the original functions  $\mathbf{y}(x+ih)$ , whereas for  $m > 0$  we get higher order derivatives from lower order derivatives. These formulas can be derived by setting up a system of derivative linear equations from the Taylor expansions of all the (derivative) functions  $\mathbf{y}^{(m)}(x \pm nh)$  for a certain range of  $n$  integers and solving these equations for the lowest derivatives (the technique is described in J. Mathews, Computer Derivations of Numerical Differentiation Formulae, International Journal of Mathematics Education in Science and Technology, Volume 34 No 2, pp.280-287). The functions  $\mathbf{y}(x \pm nh)$  could in principle be obtained using the RK equation 34.3, but a much more economical approach would be to use the 1st derivatives (i.e. using  $m = 1$  in equation 34.10). We have from equation 34.1:

$$\mathbf{y}^{(1)}(x+nh) = f(\mathbf{y}(x+nh), x+nh) \quad (34.12)$$

$$\approx f(\mathbf{y}(x) + nh\mathbf{y}^{(1)}(x), x+nh) \quad (34.13)$$

$$\approx f(\mathbf{y}(x) + nhf(\mathbf{y}(x), x), x+nh), \quad (34.14)$$

where, since the  $nh$  are small, the first order expansion of  $\mathbf{y}(x+nh)$  has been used. Note the nested use of the derivative ODE function  $f$  in equation 34.14. Since the goal is to estimate  $\mathbf{y}^{(p+1)}(x)$  via equation 34.10, we set  $p = 2k + \delta$ , from which  $k$  and  $\delta$  are uniquely determined.

In practice it turns out that equation 34.10 is numerically very unstable for higher order derivatives. We will use it only for estimating low order derivatives. This leaves us with the problem of estimating higher order derivatives in equation 34.9. We follow ideas presented in J. Comp. Appl. Math. **18**, 176-192 (1987). To this end we first split the maximum error vector into an error base vector and an error fraction scalar

$$\mathbf{e}_{max} = e_{frac} \cdot \mathbf{e}_{base} \quad (34.15)$$

and have the user supplied both quantities. The purpose of this splitting is that we separate the maximum error vector into an accuracy part (the  $e_{frac}$ , typically of the order of  $10^{-6}$  to  $10^{-12}$ ) and a part which relates to the magnitude of the vectors involved in the ODE (the  $\mathbf{e}_{base}$ ). We further assume that the user has sufficient knowledge about his ODE problem, such that he can construct these two quantities obeying roughly the following inequalities:

$$e_{frac} < 1 \quad (34.16)$$

$$\mathbf{e}_{base} \neq \mathbf{0} \quad (34.17)$$

$$|\mathbf{e}_{base}| \geq \max \left\{ |h^n \mathbf{y}^{(n)}(x)/n!|; n = 0, 1, \dots, \infty \right\}. \quad (34.18)$$

With this choice of  $\mathbf{e}_{base}$ , all terms in equation 34.7 can be divided by  $\mathbf{e}_{base}$ . If we now assume a smooth convergence of the Taylor series and further postulate an exponential type decay between the individual

$\mathbf{e}_{base}$ -rescaled Taylor terms (which all must be  $\leq 1$  due to the assumptions in equation 34.18)

$$\left| \frac{h^n \mathbf{y}^{(n)}(x)}{n! \mathbf{e}_{base}} \right| = \left| \frac{h^m \mathbf{y}^{(m)}(x)}{m! \mathbf{e}_{base}} \right|^{n/m} \quad n \geq m, \quad (34.19)$$

we can replace the higher order derivatives in equation 34.9 by lower order ones and obtain:

$$h = e_{frac}^{1/p+1} \cdot \min \left( m! \left| \frac{\mathbf{e}_{base}}{\mathbf{y}^{(m)}(x)} \right| \right)^{1/m}. \quad (34.20)$$

Note the similar structure to equation 34.9. The higher order contribution has been relegated to  $e_{frac}$ , which, due to the condition in equation 34.16, leads to larger initial step size estimates for higher order RK schemes. In practice we calculate step size estimates using only  $m = 2$  and  $m = 1$ . In rare cases when all 1st and 2nd order derivatives of  $\mathbf{y}(x)$  vanish, the (user supplied) default maximum step size estimate is used. This maximum step size estimate can be given for example as a cell/domain size measure, when it is clear that the RK stepper should not step beyond a cell or outside of the domain.

A separate RK stepping routine exists in FLASH4, which performs confined RK steps, which will be explained next. Looking at equation 34.2, we see that for an RK step the ODE function  $f$  has to be evaluated repeatedly for several intermediate  $\mathbf{y}_i$  vectors

$$\mathbf{y}_i = \mathbf{y}(x) + \sum_{j=1}^s a_{ij} \mathbf{k}_j. \quad (34.21)$$

For certain applications it may not be possible to evaluate the ODE function at some of these intermediate vectors due to lack of sufficient data. An example would be tracing the (classical) path of a particle through a cell. Only the acceleration vector field of that particular cell is available. If one of the position variables in  $\mathbf{y}_i$  steps outside of the cell boundaries, the acceleration vector at that point cannot be evaluated and the step size  $h$  must be reduced. The confined RK stepping routine needs extra boundary vectors corresponding to the variables in the  $\mathbf{y}$  vectors. The boundary vectors can be only a subset of all the  $\mathbf{y}$  vector variables. Those variables which have no corresponding boundary vector entries are assumed to be unbound. Since the boundaries might itself depend on the variables in  $\mathbf{y}$ , the boundary vectors are transmitted to the confined RK stepper as functions, which have to be designed by the user.

Currently there are five embedded explicit RK schemes implemented in FLASH, each corresponding to a particular order: 2nd order Euler-Heun, 3rd order Bogacki-Shampine, 4th order Fehlberg, 5th order Fehlberg and 5th order Cash-Karp. Addition of other embedded RK schemes to the code is easy and consists only in extending the existing Butcher tableau repository and adding the appropriate RK scheme keyword to the Runge Kutta unit.

### 34.3 Usage

To include the RK integration tool into your FLASH development, add the line

```
REQUIRES numericalTools/RungeKutta
```

into the Config file of the directory where the global API's of the interpolation unit will be used. Another way would be to include the option

```
-with-unit=numericalTools/RungeKutta
```

into your command line when you configure the code with **setup**. This gives you access to the following RK tools:

- **RungeKutta\_step**: Performs one unconfined RK step. Optimum stepping size is calculated internally by using equation 34.6 and readjusted until target accuracy is met. The routine requires as input an initial step size to start the RK stepping process.

- **RungeKutta\_stepConfined:** Performs one confined RK step. Same underlying machinery as for the unconfined RK stepper, except stepping size is additionally readjusted to comply with the supplied boundaries of the confined dependent variables.
- **RungeKutta\_stepSizeEstimate:** Returns an estimate for initial step sizes in situations where no good initial guess of step size is available. The code is based on equations 34.10, 34.14 and 34.20.

## 34.4 Unit Tests

### 34.4.1 Runge Kutta FLASH Test for a 2D Elliptical Path

#### 34.4.1.1 Derivation of the 2D elliptical path ODE

Consider the following ODE in 2D, operating on a 2D vector  $\mathbf{r}$ :

$$\frac{d\mathbf{r}}{dt} = \begin{pmatrix} \lambda_1 & \omega \\ -\omega & \lambda_2 \end{pmatrix} \mathbf{r} = \mathbf{A}\mathbf{r}. \quad (34.22)$$

This ODE describes a rotational ( $\omega$ ) and sheer ( $\lambda_1, \lambda_2$ ) force on a particle located at  $\mathbf{r}$ . If  $\lambda_1, \lambda_2 = 0$ , then the ODE describes a particle revolving in a circle of radius  $\mathbf{r}(0)$ , the initial radius of the particle. The general solution to the above ODE is:

$$\mathbf{r}(t) = e^{\mathbf{A}t} \mathbf{r}(0). \quad (34.23)$$

The matrix exponential is given by (using  $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^{-1}$  and  $e^{\mathbf{A}} = \mathbf{U}e^{\mathbf{D}}\mathbf{U}^{-1}$ , where  $\mathbf{D}$  is diagonal):

$$e^{\mathbf{A}t} = \frac{1}{\Delta} \begin{pmatrix} (e^{\mathbf{A}t})_{11} & (e^{\mathbf{A}t})_{12} \\ (e^{\mathbf{A}t})_{21} & (e^{\mathbf{A}t})_{22} \end{pmatrix}, \quad (34.24)$$

where

$$(e^{\mathbf{A}t})_{11} = \frac{1}{2}e^{t(\lambda_1+\lambda_2+\Delta)/2}(\lambda_1 - \lambda_2 + \Delta) + \frac{1}{2}e^{t(\lambda_1+\lambda_2-\Delta)/2}(-\lambda_1 + \lambda_2 + \Delta) \quad (34.25)$$

$$(e^{\mathbf{A}t})_{12} = \omega(e^{t(\lambda_1+\lambda_2+\Delta)/2} - e^{t(\lambda_1+\lambda_2-\Delta)/2}) \quad (34.26)$$

$$(e^{\mathbf{A}t})_{21} = -(e^{\mathbf{A}t})_{12} \quad (34.27)$$

$$(e^{\mathbf{A}t})_{22} = \frac{1}{2}e^{t(\lambda_1+\lambda_2+\Delta)/2}(-\lambda_1 + \lambda_2 + \Delta) + \frac{1}{2}e^{t(\lambda_1+\lambda_2-\Delta)/2}(\lambda_1 - \lambda_2 + \Delta) \quad (34.28)$$

and

$$\Delta = \sqrt{(\lambda_1 - \lambda_2)^2 - 4\omega^2}. \quad (34.29)$$

For general values of  $\lambda_1, \lambda_2, \omega$ , the path of the particle is a spiral around the origin. Closed paths are possible, if at certain times  $t \neq 0$  we have  $e^{\mathbf{A}t} = \mathbf{I}$ . A necessary condition is thus that the offdiagonal elements of  $e^{\mathbf{A}t}$  have to become zero for certain  $t \neq 0$ . Since each of these elements has a factor of the structure  $e^B - e^C$ , then either both  $B$  and  $C$  are real and equal or  $B$  and  $C$  are imaginary with opposite sign. Both situations can only happen, if  $\lambda_2 = -\lambda_1$ . The first case ( $B, C$  real and equal) is only possible if  $\Delta = 0$  and the second case ( $B, C$  imaginary and of opposite sign) only if  $\Delta$  is imaginary. Hence, necessary conditions for a closed path are:

$$\lambda_2 = -\lambda_1 \quad (34.30)$$

$$\omega^2 \geq \lambda_1^2. \quad (34.31)$$

Let us now introduce the parameter  $|k| > 1$  such that:

$$\omega = k\lambda_1. \quad (34.32)$$

Then our closed path ODE becomes:

$$\frac{d\mathbf{r}}{dt} = \lambda_1 \begin{pmatrix} 1 & k \\ -k & -1 \end{pmatrix} \mathbf{r} \quad (34.33)$$

and, since  $\lambda_1$  is now just merely a scaling factor, we can further condense the closed path ODE to:

$$\frac{d\mathbf{r}}{dt} = \begin{pmatrix} 1 & k \\ -k & -1 \end{pmatrix} \mathbf{r}. \quad (34.34)$$

This is the 2D ellipse ODE we are going to use for our Runge Kutta test. Inserting  $\lambda_1 = 1$ ,  $\lambda_2 = -1$  and  $\omega = k$  into the above general solution, we obtain, after inserting the identity  $e^{i\phi} = \cos(\phi) + i\sin(\phi)$ , the following real path solution:

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} \frac{\sin(\phi)}{\sqrt{k^2-1}} + \cos(\phi) & \frac{k\sin(\phi)}{\sqrt{k^2-1}} \\ -\frac{k\sin(\phi)}{\sqrt{k^2-1}} & -\frac{\sin(\phi)}{\sqrt{k^2-1}} + \cos(\phi) \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \quad (34.35)$$

where

$$\phi = t\sqrt{k^2-1}. \quad (34.36)$$

For  $\phi = n\pi; n = 1, 3, 5, \dots$ , the 2x2 matrix is equal to  $-\mathbf{I}$ , i.e. the particle made a half revolution. For  $\phi = n\pi; n = 2, 4, 6, \dots$ , the 2x2 matrix is equal to  $\mathbf{I}$  and the particle made a complete revolution. The time  $T$  it takes for the particle to make one complete revolution (its time period) is thus:

$$T = \frac{2\pi}{\sqrt{k^2-1}}. \quad (34.37)$$

The square of the distance  $d^2 = [x(t)]^2 + [y(t)]^2$  the particle is found from the origin has a minimum and a maximum at the following angles:

$$\phi_{max}(k > 1) = \arctan \left[ \sqrt{\frac{k+1}{k-1}} \frac{x_0 + y_0}{x_0 - y_0} \right] \quad (34.38)$$

$$\phi_{min}(k > 1) = -\arctan \left[ \sqrt{\frac{k-1}{k+1}} \frac{x_0 - y_0}{x_0 + y_0} \right] \quad (34.39)$$

$$\phi_{max}(k < 1) = -\phi_{min}(k > 1) \quad (34.40)$$

$$\phi_{min}(k < 1) = -\phi_{max}(k > 1). \quad (34.41)$$

These angles (range  $[-\pi/2, +\pi/2]$ ) give only the angles to the nearest maximum/minimum point from the original position  $(x_0, y_0)$ . The other two angles corresponding to the remaining maximum/minimum distance points are obtained by adding  $\pi$ . The corresponding times are:

$$t_{max} = \phi_{max}/\sqrt{k^2-1} \quad (34.42)$$

$$t_{min} = \phi_{min}/\sqrt{k^2-1}. \quad (34.43)$$

The corresponding minimum and maximum square distances are:

$$d_{min}^2 = \frac{k(x_0^2 + y_0^2) + 2x_0y_0}{k + \text{sgn}[k]} \quad (34.44)$$

$$d_{max}^2 = \frac{k(x_0^2 + y_0^2) + 2x_0y_0}{k - \text{sgn}[k]}, \quad (34.45)$$

where the signum function  $\text{sgn}$  is:

$$k > 0 \longrightarrow \text{sgn}[k] = +1 \quad (34.46)$$

$$k < 0 \longrightarrow \text{sgn}[k] = -1. \quad (34.47)$$

From these formulas we see that

- if  $x_0 = y_0$  and  $k > 1$ , then the first  $t_{min}$  is equal to zero, i.e. the particle is at its minimum distance  $\sqrt{2}|x_0|$  from the origin.
- if  $x_0 = y_0$  and  $k < -1$ , then the first  $t_{max}$  is equal to zero, i.e. the particle is at its maximum distance  $\sqrt{2}|x_0|$  from the origin.
- if  $x_0 = -y_0$  and  $k > 1$ , then the first  $t_{max}$  is equal to zero, i.e. the particle is at its maximum distance  $\sqrt{2}|x_0|$  from the origin.
- if  $x_0 = -y_0$  and  $k < -1$ , then the first  $t_{min}$  is equal to zero, i.e. the particle is at its minimum distance  $\sqrt{2}|x_0|$  from the origin.
- the aspect ratio  $A$  of the elliptical path is equal to  $A = \sqrt{(k + \text{sgn}[k])/(k - \text{sgn}[k])}$ .

We next develop the approach and formulas to find the minimum/maximum distance from a general point in a 2D domain to a general ellipse inside this 2D domain.

### 34.4.1.2 Minimum and maximum distance from a point to a general ellipse in a 2D cartesian domain

We will first get the general parametric equation of an ellipse in 2D cartesian space. We start by placing a 2D ellipse with its center at the cartesian origin and with its minor/major axis aligned along the cartesian y/x axis. Let the semi-major axis have length  $a$  and the semi-minor axis have length  $b$ . The parametric equation for this ellipse is:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \sin \theta \\ b \cos \theta \end{pmatrix}, \quad (34.48)$$

where the angle parameter is in the range  $0 \leq \theta \leq 2\pi$ . Next we clockwise rotate the entire ellipse such that the minor axis makes an angle  $\alpha$  with the cartesian y-axis. The range of this rotation angle is  $0 \leq \alpha \leq \pi$ . The parametric equation for the rotated ellipse becomes

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{R} \begin{pmatrix} a \sin \theta \\ b \cos \theta \end{pmatrix} = \begin{pmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} a \sin \theta \\ b \cos \theta \end{pmatrix}, \quad (34.49)$$

where  $\mathbf{R}$  is the rotation matrix that sends every point to its new rotated point. Finally we add the center translation vector  $\mathbf{T} = (T_x, T_y)$  to the equation for ellipses that are not located at the cartesian origin:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} a \sin \theta \\ b \cos \theta \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \end{pmatrix}. \quad (34.50)$$

This is the general parametric equation of any ellipse in 2D cartesian space. Consider a specific point  $(x, y)$  on the ellipse. The tangent line at this ellipse point contains the tiny derivative vector

$$\mathbf{v} = \begin{pmatrix} dx/d\theta \\ dy/d\theta \end{pmatrix} = \begin{pmatrix} a \cos \alpha \cos \theta - b \sin \alpha \sin \theta \\ -a \sin \alpha \cos \theta - b \cos \alpha \sin \theta \end{pmatrix}, \quad (34.51)$$

with its origin at the ellipse point  $(x, y)$ . The other vector we consider is the vector

$$\mathbf{w} = \begin{pmatrix} x_p - x \\ y_p - y \end{pmatrix} = \begin{pmatrix} x_p - a \cos \alpha \sin \theta - b \sin \alpha \cos \theta - T_x \\ y_p + a \sin \alpha \sin \theta - b \cos \alpha \cos \theta - T_y \end{pmatrix}, \quad (34.52)$$

which also has its origin at the ellipse point  $(x, y)$ . The minimum distance occurs for those  $\theta$  for which  $\mathbf{v} \cdot \mathbf{w} = 0$  (perpendicular vectors). This gives the following equation for  $\theta$ :

$$R \cos \theta + S \sin \theta + U \cos \theta \sin \theta = 0, \quad (34.53)$$

where

$$R = a[(y_p - T_y) \sin \alpha - (x_p - T_x) \cos \alpha] \quad (34.54)$$

$$S = b[(x_p - T_x) \sin \alpha + (y_p - T_y) \cos \alpha] \quad (34.55)$$

$$U = a^2 - b^2. \quad (34.56)$$

Note, that  $U \neq 0$ , whenever  $a \neq b$ . For  $a = b$  (a circle) we will have  $U = 0$  and the equation becomes easily solvable. The elliptical  $U$  term makes the equation much harder to solve. Eliminating the  $\sin \theta$  term, we arrive at the following quartic equation in  $\cos \theta$ :

$$\cos^4 \theta + 2 \left( \frac{S}{U} \right) \cos^3 \theta + \left[ \left( \frac{R}{U} \right)^2 + \left( \frac{S}{U} \right)^2 - 1 \right] \cos^2 \theta - 2 \left( \frac{S}{U} \right) \cos \theta - \left( \frac{S}{U} \right)^2 = 0.$$

Its solutions give us up to 4 different possible angles  $\theta$ . However, to each  $\cos \theta$  solution there correspond two  $\sin \theta$  solutions via  $\sin \theta = \pm \sqrt{1 - \cos^2 \theta}$ , increasing the total possible solutions to 8. Each of these 8 solutions must be checked for the minimum and maximum distance given by  $|\mathbf{w}|$ . As far as the number of real solutions for the above quartic is concerned, when the point is inside or close to the ellipse, we can visualize that there are 4 possible directions from the point to the ellipse for which we have  $\mathbf{v} \cdot \mathbf{w} = 0$  (i.e. we hit the ellipse curve at right angles). Hence for this point location we expect 4 real quartic solutions. On the other hand, if we are far away from the ellipse, only 2 such points on the elliptical curve will exist and we expect 2 real and 2 complex solutions. We should never get 4 complex solutions of the above quartic.

### 34.4.1.3 The FLASH test problem for the Runge Kutta 2D ellipse test

We will now set up the unit test for the Runge Kutta 2D ellipse test. We start by stating the initial conditions and the requested shape of the ellipse:

1. The particle is initially at position  $(x_0, y_0) = (1, 1)$ .
2. We want the particle to follow an elliptical path with given aspect ratio  $A$ .

From this requested aspect ratio we calculate our needed  $k$  as:

$$k = \frac{A^2 + 1}{A^2 - 1} \quad (34.57)$$

and we take  $k$  to be positive from now on. This means that the particle will start moving along the elliptical curve in the clockwise direction. By looking at the general formulas derived in the previous sections, we can state the following properties and equations for the ellipse. The square distance extrema and locations will be:

$$d_{min}^2 = 2 \quad \text{located at } (1, 1) \text{ and } (-1, -1) \quad (34.58)$$

$$d_{max}^2 = 2A^2 \quad \text{located at } (A, -A) \text{ and } (-A, A). \quad (34.59)$$

From this we see that the major and minor semi-axis will have lengths:

$$a = \sqrt{2}A \quad (34.60)$$

$$b = \sqrt{2}. \quad (34.61)$$

From the particle's initial position at  $(1, 1)$ , the minor and major semi-axis will be reached at the following times ( $n = 0, 1, 2, \dots$ ):

$$t_{minor} = \frac{(A^2 - 1)}{2A} \pi n \quad (34.62)$$

$$t_{major} = \frac{(A^2 - 1)}{2A} \pi (n + 1/2). \quad (34.63)$$

Hence the period of one complete revolution (returning to  $(1, 1)$ ) is given by:

$$T = \frac{(A^2 - 1)}{A} \pi. \quad (34.64)$$

The ellipse is represented by the following two (equivalent) forms:

$$\frac{(x - y)^2}{4A^2} + \frac{(x + y)^2}{4} = 1 \quad (34.65)$$



Table 34.1: Number of RK steps, largest RK step coordinate error  $\max(\mathbf{e}_{step})$ , maximum time distance error  $\max(e_{time})$  and maximum closest distance error  $\max(e_{closest})$  values for a complete RK elliptical curve tracing with amplification  $A = 2$ , starting at the point  $(x_0, y_0) = (1, 1)$ . Results shown as a function of embedded RK scheme used with fractional errors  $e_{frac} = 10^{-8}$  and  $e_{frac} = 10^{-12}$ .

RK scheme (order)	RK steps	$\max(\mathbf{e}_{step})$	$\max(e_{time})$	$\max(e_{closest})$
Fractional error $e_{frac} = 10^{-8}$				
Euler-Heun (2)	67887	$8.10 \times 10^{-9}$	$2.63 \times 10^{-8}$	$1.55 \times 10^{-11}$
Bogacki-Shampine (3)	1102	$7.39 \times 10^{-9}$	$1.12 \times 10^{-7}$	$1.09 \times 10^{-7}$
Fehlberg (4)	228	$6.97 \times 10^{-9}$	$1.78 \times 10^{-6}$	$1.73 \times 10^{-6}$
Fehlberg (5)	84	$6.84 \times 10^{-9}$	$3.23 \times 10^{-8}$	$3.02 \times 10^{-8}$
Cash-Karp (5)	60	$7.14 \times 10^{-9}$	$2.72 \times 10^{-8}$	$2.61 \times 10^{-8}$
Fractional error $e_{frac} = 10^{-12}$				
Euler-Heun (2)	6788695	$8.10 \times 10^{-13}$	$3.85 \times 10^{-12}$	$4.14 \times 10^{-10}$
Bogacki-Shampine (3)	23729	$7.29 \times 10^{-13}$	$1.12 \times 10^{-11}$	$1.09 \times 10^{-11}$
Fehlberg (4)	2272	$6.60 \times 10^{-13}$	$1.75 \times 10^{-9}$	$1.72 \times 10^{-9}$
Fehlberg (5)	526	$6.06 \times 10^{-13}$	$3.22 \times 10^{-12}$	$3.10 \times 10^{-12}$
Cash-Karp (5)	372	$6.13 \times 10^{-13}$	$2.79 \times 10^{-12}$	$2.69 \times 10^{-12}$

and the parametric time equations:

$$x(t) = A \sin\left(\frac{2A}{A^2 - 1}t\right) + \cos\left(\frac{2A}{A^2 - 1}t\right) \quad (34.66)$$

$$y(t) = -A \sin\left(\frac{2A}{A^2 - 1}t\right) + \cos\left(\frac{2A}{A^2 - 1}t\right). \quad (34.67)$$

Each of the following Runge Kutta runs follows the elliptical curve through a full ellipse revolution, starting at the initial at position  $(x_0, y_0) = (1, 1)$ . The calculations are stopped as soon as the  $y = 1$  line is crossed once, with the last crossing point still included in the calculation. Choosing an ellipse with an aspect ration of  $A = 2$ , we report the following data using fractional errors of  $e_{frac} = 10^{-8}$  and  $10^{-12}$  and a  $(x, y)$  coordinate error base vector of  $\mathbf{e}_{base} = (1, 1)$  (Eq.(34.15) for each Runge Kutta run in separate tables:

- The embedded RK scheme used and its order.
- The number of RK steps used to trace the full elliptical curve.
- The overall largest RK step coordinate error  $\max(\mathbf{e}_{step})$ , which should be  $\leq e_{frac} \cdot \mathbf{e}_{base}$ .
- The maximum time distance error  $\max(e_{time})$  from the elliptical curve, where

$$e_{time} = \sqrt{[x - x(t)]^2 + [y - y(t)]^2} \quad (34.68)$$

is a measure of how far the RK curve deviates from the true elliptical curve at time  $t$ . The quantity  $e_{time}$  can be viewed as a global error of the RK scheme.

- The maximum of the closest distance error  $\max(e_{closest})$  to the true elliptical curve. The closest elliptical distance error  $e_{closest}$  is calculated using the approach outlined in section 34.4.1.2 and is a measure of how closely the RK scheme traces out the true elliptical curve.

Table 34.1 shows the obtained FLASH results. The higher order Fehlberg (5) and Cash-Karp (5) RK schemes outperform the lower order ones. The Cash-Karp RK scheme gives the best performance and is thus the set default RK scheme for the Runge Kutta unit.

### 34.4.2 Runge Kutta FLASH Test for Binomial Homogeneous Higher Order ODE

A binomial homogeneous ODE is an ODE of the form ( $n \geq 1$ ):

$$\sum_{i=0}^n \binom{n}{i} f^{(i)}(x) = 0, \quad (34.69)$$

whose general solution is

$$f(x) = e^{-x} \left( \sum_{k=0}^{n-1} c_k x^k \right). \quad (34.70)$$

We are interested in the particular solution with  $\forall c_k = 1$ :

$$f(x) = e^{-x} P(x) \quad (34.71)$$

$$P(x) = \sum_{k=0}^{n-1} x^k. \quad (34.72)$$

The derivatives of this particular solution are given by:

$$f^{(i)}(x) = e^{-x} \sum_{k=0}^i (-1)^{i+k} \binom{i}{k} P^{(k)}(x). \quad (34.73)$$

The values of the  $k$ -th derivative of  $P(x)$  at  $x = 0$  are the factorials of  $k$ , i.e. we have  $P^{(k)}(0) = k!$ . Thus the initial set of conditions at  $x = 0$  for the particular solution are:

$$\begin{aligned} f^{(i)}(0) &= i! \sum_{k=0}^i \frac{(-1)^k}{k!} \\ &= !i, \quad 0 \leq i \leq n-1 \end{aligned} \quad (34.74)$$

where  $!i$  denotes the number of out-of-place permutations (derangements, no numbers in original places) of  $i$ . A recursive formula for  $!i$  can be given:  $!i = (i-1)[!(i-1) + !(i-2)]$ , with starting values  $!0 = 1$  and  $!1 = 0$ . Note that the given set of initial condition implies

$$f^{(n)}(0) = !n - n!, \quad (34.75)$$

due to the definition of the binomial ODE equation.

To solve the binomial ODE using the Runge Kutta scheme, the usual transformation to a set of coupled 1st order ODE's is done. Define the  $n$ -dimensional vector

$$\mathbf{y} = \left( f, f^{(1)}, f^{(2)}, \dots, f^{(n-1)} \right)^T, \quad (34.76)$$

the coupled set of 1st order ODE's are in matrix form:

$$\frac{d\mathbf{y}}{dx} = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ -1 & -n & -\binom{n}{2} & -\binom{n}{3} & \cdots & -n \end{pmatrix} \mathbf{y}. \quad (34.77)$$

We compare the Runge Kutta vector  $\mathbf{y}$  with the vector  $\mathbf{y}$  evaluated using Eq.(34.73).

**Part IX**

**Simulation Units**



## Chapter 35

# The Supplied Test Problems

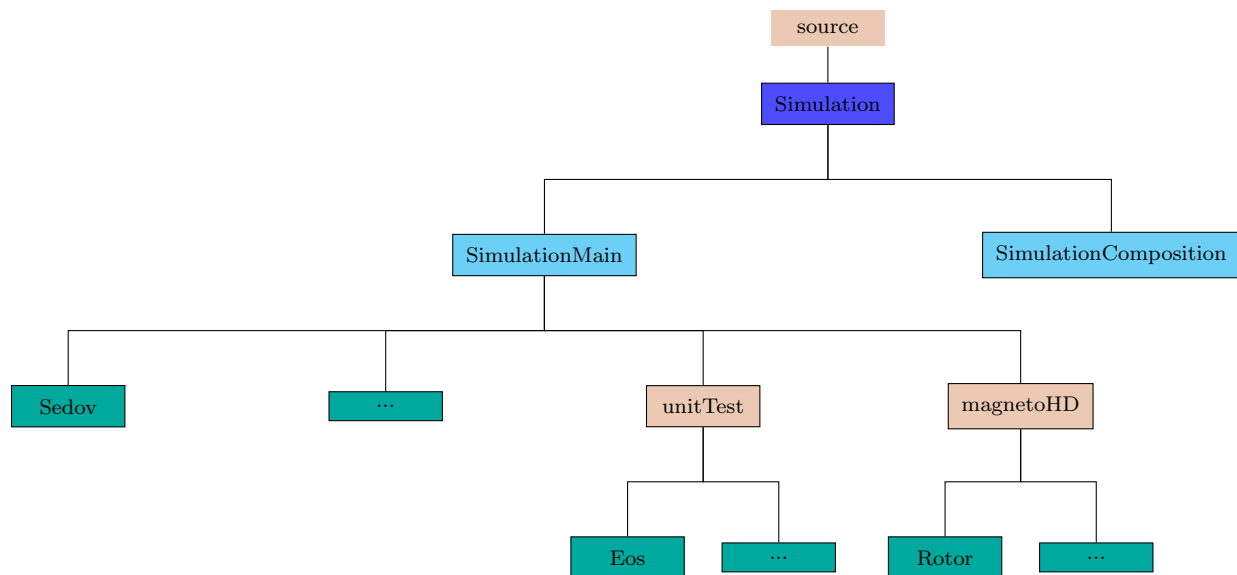


Figure 35.1: The **Simulation** unit directory tree. Only some of the provided simulation implementations are shown. Users are expected to add their own simulations to the tree.

To verify that FLASH works as expected and to debug changes in the code, we have created a suite of standard test problems. Many of these problems have analytical solutions that can be used to test the accuracy of the code. Most of the problems that do not have analytical solutions produce well-defined flow features that have been verified by experiments and are stringent tests of the code. For the remaining problems, converged solutions, which can be used to test the accuracy of lower resolution simulations, are easy to obtain. The test suite configuration code is included with the FLASH source tree (in the **Simulation/** directory), so it is easy to configure and run FLASH with any of these problems ‘out of the box.’ Sample runtime parameter files are also included. All the test problems reside in the **Simulations** unit. The unit provides some general interfaces most of which do not have general implementations. Each application provides its own implementation for these interfaces, for example, **Simulation\_initBlock**. The exception is **Simulation\_initSpecies**, which provides general implementations for different classes of problems.

### 35.1 Hydrodynamics Test Problems

These problems are primarily designed to test the functioning of the hydrodynamics solvers within FLASH4.

### 35.1.1 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

$$\begin{aligned} \rho_L &= 1.0 & \rho_R &= 0.125 \\ p_L &= 1.0 & p_R &= 0.1 \end{aligned} \tag{35.1}$$

The ratio of specific heats  $\gamma$  is chosen to be 1.4 on both sides of the interface.

In FLASH, the Sod problem (Sod) uses the runtime parameters listed in Table 35.1 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 35.1 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 35.1: Runtime parameters used with the Sod test problem.

Variable	Type	Default	Description
<code>sim_rhoLeft</code>	real	1	Initial density to the left of the interface ( $\rho_L$ )
<code>sim_rhoRight</code>	real	0.125	Initial density to the right ( $\rho_R$ )
<code>sim_pLeft</code>	real	1	Initial pressure to the left ( $p_L$ )
<code>sim_pRight</code>	real	0.1	Initial pressure to the right ( $p_R$ )
<code>sim_uLeft</code>	real	0	Initial velocity (perpendicular to interface) to the left ( $u_L$ )
<code>sim_uRight</code>	real	0	Initial velocity (perpendicular to interface) to the right ( $u_R$ )
<code>sim_xangle</code>	real	0	Angle made by interface normal with the $x$ -axis (degrees)
<code>sim_yangle</code>	real	90	Angle made by interface normal with the $y$ -axis (degrees)
<code>sim_posn</code>	real	0.5	Point of intersection between the interface plane and the $x$ -axis

Figure 35.2 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 35.3 shows the result of running the Sod problem on the same two-dimensional grid with different shock normals: parallel to the  $x$ -axis ( $\theta = 0^\circ$ ) and along the box diagonal ( $\theta = 45^\circ$ ). For the diagonal solution, we have interpolated values of density, specific internal energy, and velocity to a set of 256 points

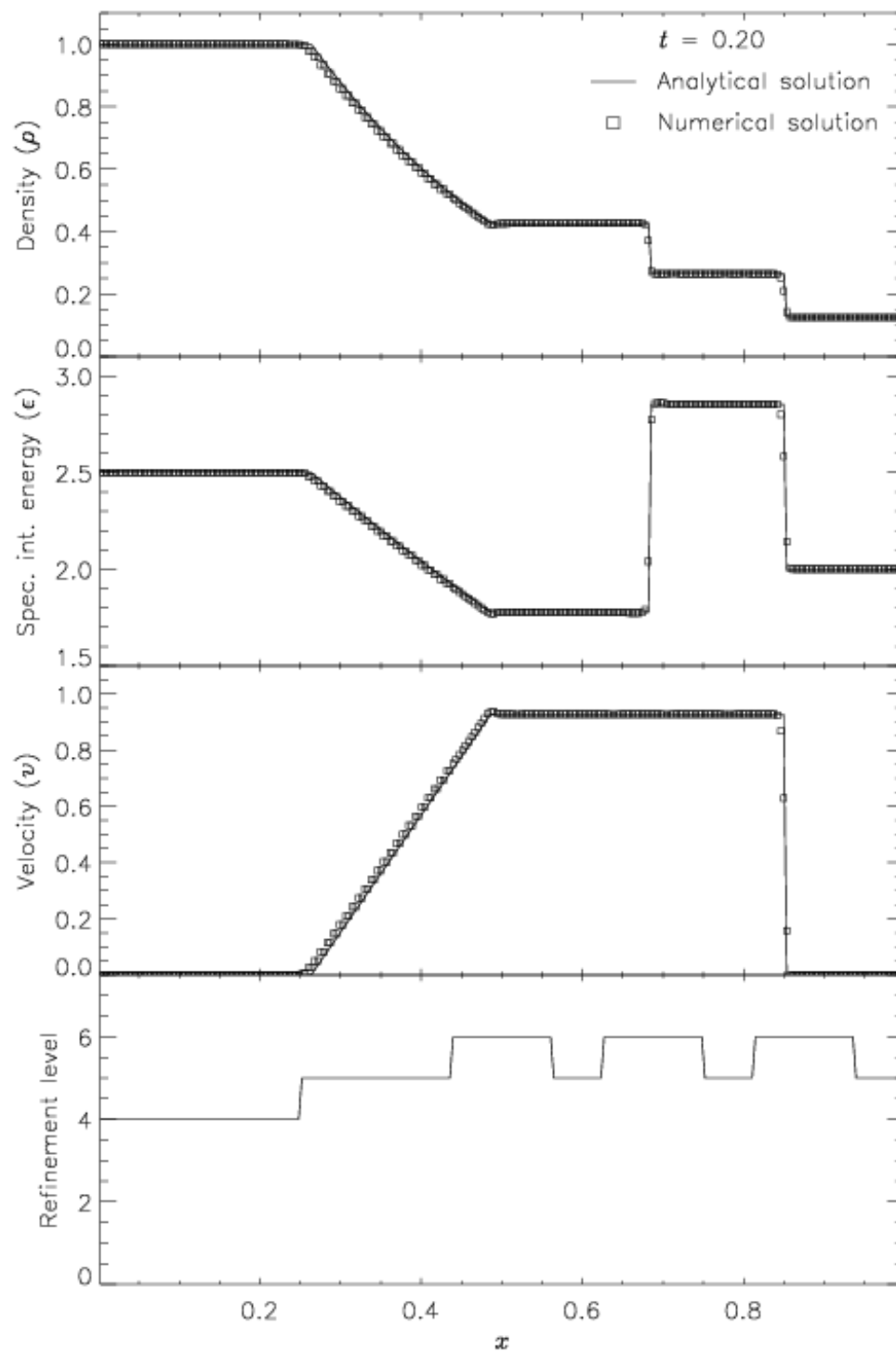


Figure 35.2: Comparison of numerical and analytical solutions to the Sod problem. A 2D grid with six levels of refinement is used. The shock normal is parallel to the  $x$ -axis.

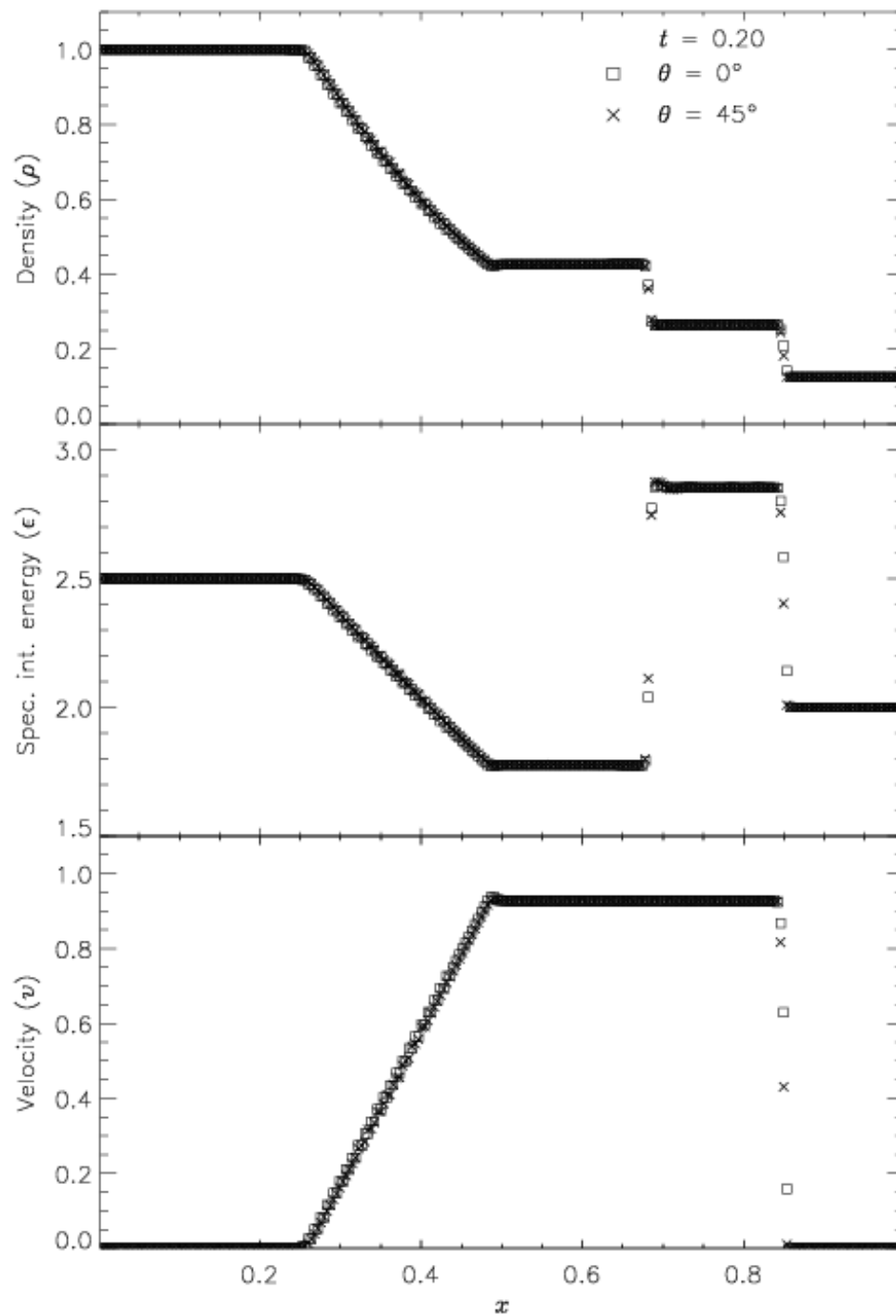


Figure 35.3: Comparison of numerical solutions to the Sod problem for two different angles ( $\theta$ ) of the shock normal relative to the  $x$ -axis. A 2D grid with six levels of refinement is used.



spaced exactly as in the  $x$ -axis solution. This comparison shows the effects of the second-order directional splitting used with FLASH on the resolution of shocks. At the right side of the rarefaction and at the contact discontinuity, the diagonal solution undergoes slightly larger oscillations (on the order of a few percent) than the  $x$ -axis solution. Also, the value of each variable inside the discontinuity regions differs between the two solutions by up to 10%. However, the location and thickness of the discontinuities is the same for the two solutions. In general, shocks at an angle to the grid are resolved with approximately the same number of cells as shocks parallel to a coordinate axis.

Figure 35.4 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.

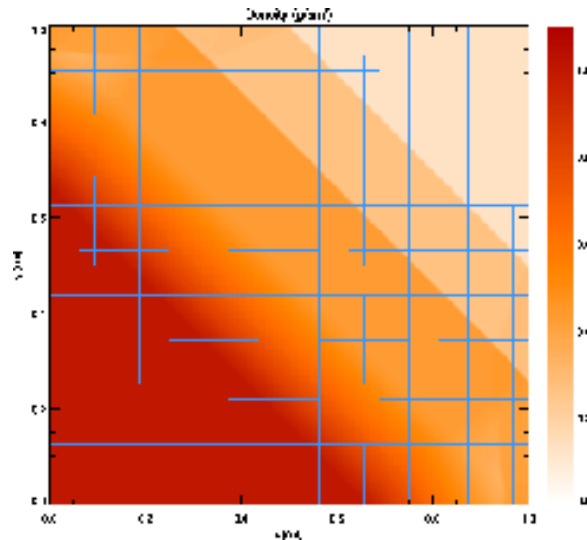


Figure 35.4: Density in the diagonal 2D Sod problem with six levels of refinement at  $t = 0.2$ . The outlines of AMR blocks are shown (each block contains  $8 \times 8$  cells).

#### SodStep Example

FLASH4 also contains under the name **SodStep** a variant of the Sod problem. This setup is provided as an example for setting up simulations on domains with steps and obstacles. See the files in the **SodStep** simulation directory and the **Simulation.defineDomain** description for more information on how to use this feature.

### 35.1.2 Variants of the Sod Problem in Curvilinear Geometries

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

- An axisymmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with `./setup Sod -auto -2d -geometry=cylindrical` and using runtime parameters that include `geometry = "cylindrical"`. Use `sim_xangle = 0` to configure an initial shock front that is shaped like a cylinder. Results as in those discussed in Toro 1999 can be obtained.

- A spherically symmetric variant of the Sod problem can be configured by setting up the regular Sod simulation with `./setup Sod -auto -1d -geometry=spherical` and using runtime parameters that include `geometry = "spherical"`. Again results as in those discussed in Toro 1999 can be obtained.
- To test the behavior of FLASH solutions when the physical symmetry of the problem does not match the geometry of the simulation, a separate simulation is provided under the name `SodSpherical`. To use this, configure with `./setup SodSpherical -auto -2d -geometry=spherical` and using runtime parameters that include `geometry = "spherical"`. As a 2D setup, `SodSpherical` represents physically axisymmetric initial conditions in spherical coordinates. The physical problem can be chosen to be the same as in the previous case with cylindrical Sod. Again results as in those discussed in Toro 1999 can be obtained.
- The `SodSpherical` setup can also be configured in 1D and will act like the 1D Sod setup in that case.

### 35.1.3 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

$$p_L = 1000, \quad p_M = 0.01, \quad p_R = 100. \quad (35.2)$$

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

**Figure 35.5** 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 35.5** 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 35.6** shows the self-convergence of density and pressure when FLASH is run on this problem. We compare the density, pressure, and total specific energy at  $t = 0.038$  obtained using FLASH with ten levels of refinement to solutions using several different maximum refinement levels. This figure plots the L1 error norm for each variable  $u$ , defined using

$$\mathcal{E}(N_{\text{ref}}; u) \equiv \frac{1}{N(N_{\text{ref}})} \sum_{i=1}^{N(N_{\text{ref}})} \left| \frac{u_i(N_{\text{ref}}) - Au_i(10)}{u_i(10)} \right|, \quad (35.3)$$

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.

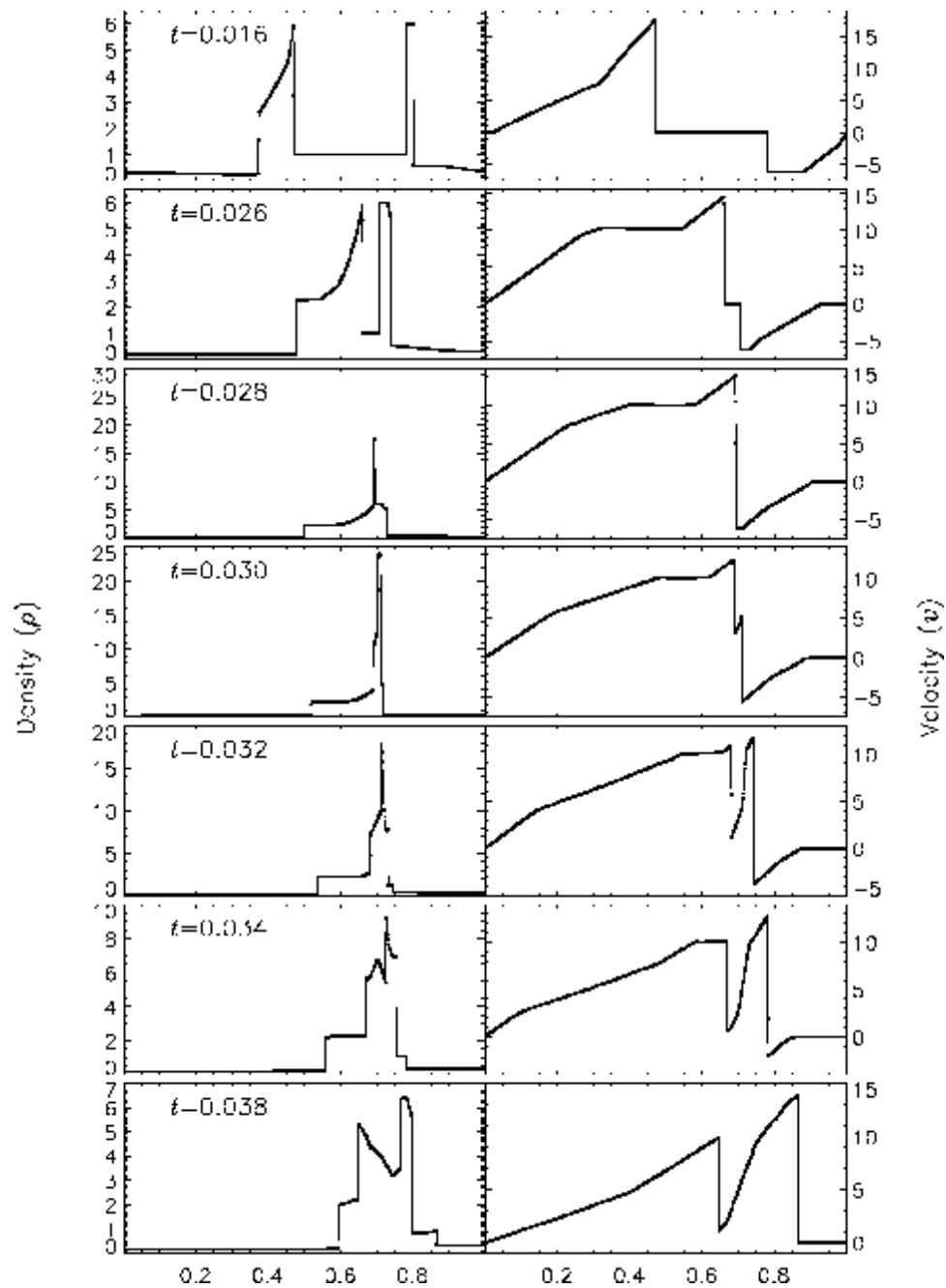
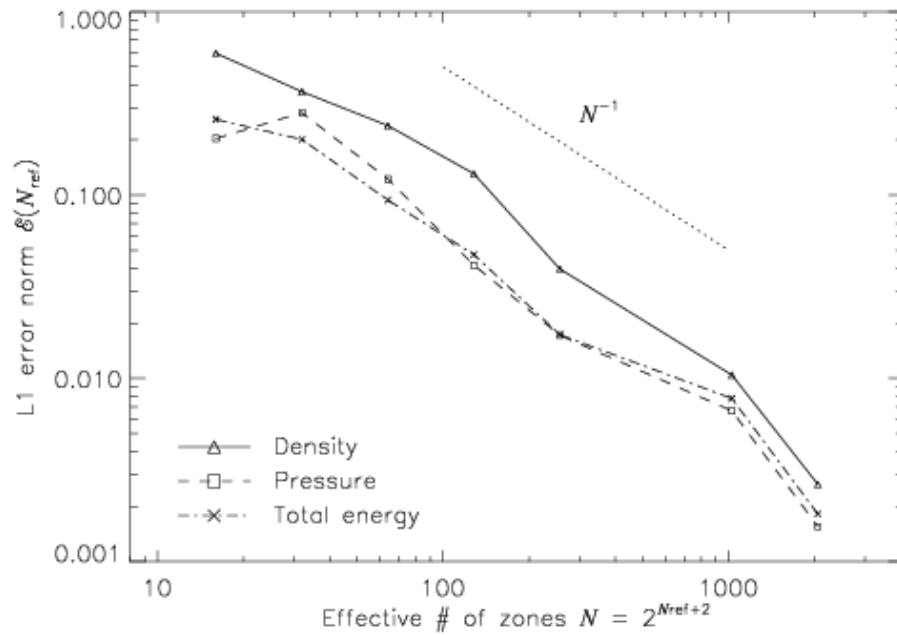


Figure 35.5: Density and velocity profiles in the Woodward-Colella interacting blast-wave problem **Blast2** as computed by FLASH using ten levels of refinement.

Although PPM is formally a second-order method, the convergence rate is only linear. This is not surprising, since the order of accuracy of a method applies only to smooth flow and not to flows containing discontinuities. In fact, all shock capturing schemes are only first-order accurate in the vicinity of discontinuities. Indeed, in their comparison of the performance of seven nominally second-order hydrodynamic methods on this problem, Woodward and Colella found that only PPM achieved even linear convergence; the other methods were worse. The error norm is very sensitive to the correct position and shape of the strong, narrow shocks generated in this problem.

The additional runtime parameters supplied with the `2blast` problem are listed in [Table 35.2](#). 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.

Figure 35.6: Self-convergence of the density, pressure, and total specific energy in the **Blast2** test problem.Table 35.2: Runtime parameters used with the **2blast** test problem.

Variable	Type	Default	Description
rho_left	real	1	Initial density to the left of the left interface ( $\rho_L$ )
rho_mid	real	1	Initial density between the two interfaces ( $\rho_M$ )
rho_right	real	1	Initial density to the right of the right interface ( $\rho_R$ )
p_left	real	1000	Initial pressure to the left ( $p_L$ )
p_mid	real	0.01	Initial pressure in the middle ( $p_M$ )
p_right	real	100	Initial pressure to the right ( $p_R$ )
u_left	real	0	Initial velocity (perpendicular to interface) to the left ( $u_L$ )
u_mid	real	0	Initial velocity (perpendicular to interface) in the middle ( $u_M$ )
u_right	real	0	Initial velocity (perpendicular to interface) to the right ( $u_R$ )
xangle	real	0	Angle made by interface normal with the $x$ -axis (degrees)
yangle	real	90	Angle made by interface normal with the $y$ -axis (degrees)
posnL	real	0.1	Point of intersection between the left interface plane and the $x$ -axis
posnR	real	0.9	Point of intersection between the right interface plane and the $x$ -axis

### 35.1.4 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. We provide two different ways to generate the initial conditions:

1. 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

$$p'_0 = \frac{3(\gamma - 1)E}{(\nu + 1)\pi \delta r^\nu}, \quad (35.4)$$

where  $\nu = 2$  for cylindrical geometry and  $\nu = 3$  for spherical geometry.

2. We initialize from a precomputed pseudo-analytical solution by reading in a 1D profile.

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

$$R(t) = C_\nu(\gamma) \left( \frac{Et^2}{\rho_0} \right)^{1/(\nu+2)}. \quad (35.5)$$

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

$$\begin{aligned} \rho &= \rho_1 \equiv \frac{\gamma + 1}{\gamma - 1} \rho_0 \\ p &= p_1 \equiv \frac{2}{\gamma + 1} \rho_0 u^2 \\ v &= v_1 \equiv \frac{2}{\gamma + 1} u, \end{aligned} \quad (35.6)$$

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

$$\begin{aligned} \rho(\xi)/\rho_1 &\propto \xi^{\nu/(\gamma-1)} \\ p(\xi)/p_1 &= \text{constant} \\ v(\xi)/v_1 &\propto \xi. \end{aligned} \quad (35.7)$$

**Figure 35.7** 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  $r \rightarrow 0$ ). 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

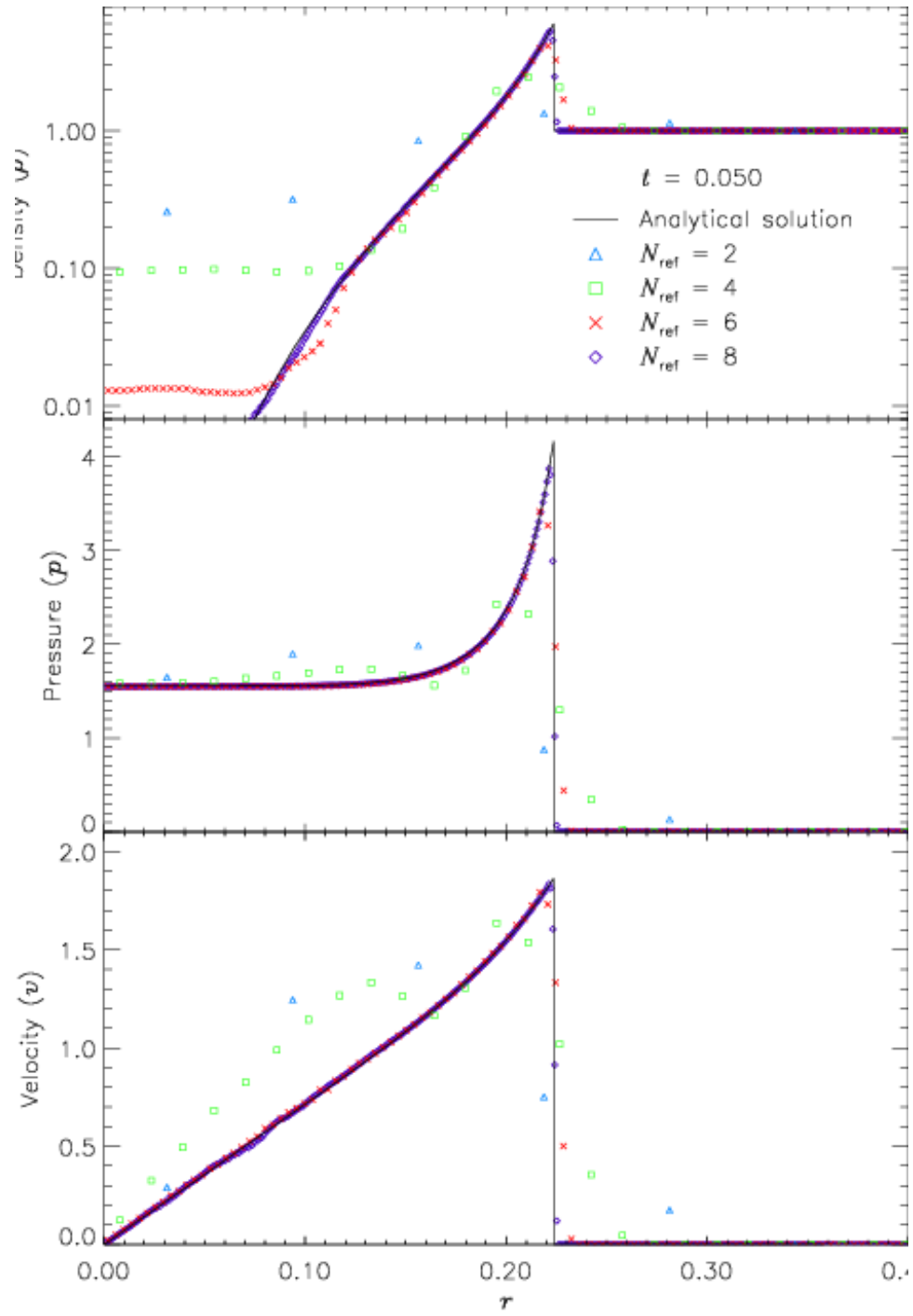


Figure 35.7: Comparison of numerical and analytical solutions to the Sedov problem in two dimensions. Numerical solution values are averages in radial bins at the finest AMR grid resolution  $N_{\text{ref}}$  in each run.

by four cells in the  $N_{\text{ref}} = 4$  case, 17 cells in the  $N_{\text{ref}} = 6$  case, and 69 cells for  $N_{\text{ref}} = 8$ . However, because the solution is self-similar, for any given maximum refinement level, the structure will be four cells wide at a sufficiently early time. The behavior when the shock is under-resolved is to underestimate the peak value of each variable, particularly the density and pressure.

Figure 35.8 shows the pressure field in the 8-level calculation at  $t = 0.05$  together with the block refinement pattern. Note that a relatively small fraction of the grid is maximally refined in this problem. Although the pressure gradient at the center of the grid is small, this region is refined because of the large temperature gradient there. This illustrates the ability of PARAMESH to refine grids using several different variables at once.

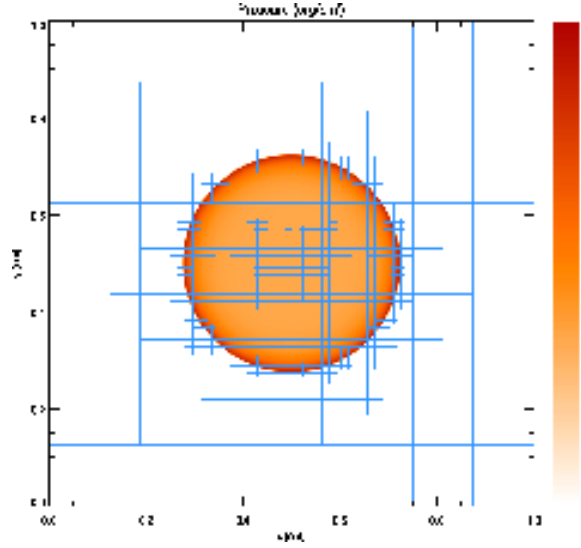


Figure 35.8: Pressure field in the 2D Sedov explosion problem with 8 levels of refinement at  $t = 0.05$ . The outlines of the AMR blocks are overlaid on the pressure colormap.

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 35.9. 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  $1\frac{1}{2}$  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 35.3. 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.

#### 35.1.4.1 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 35.10 and 35.11 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.



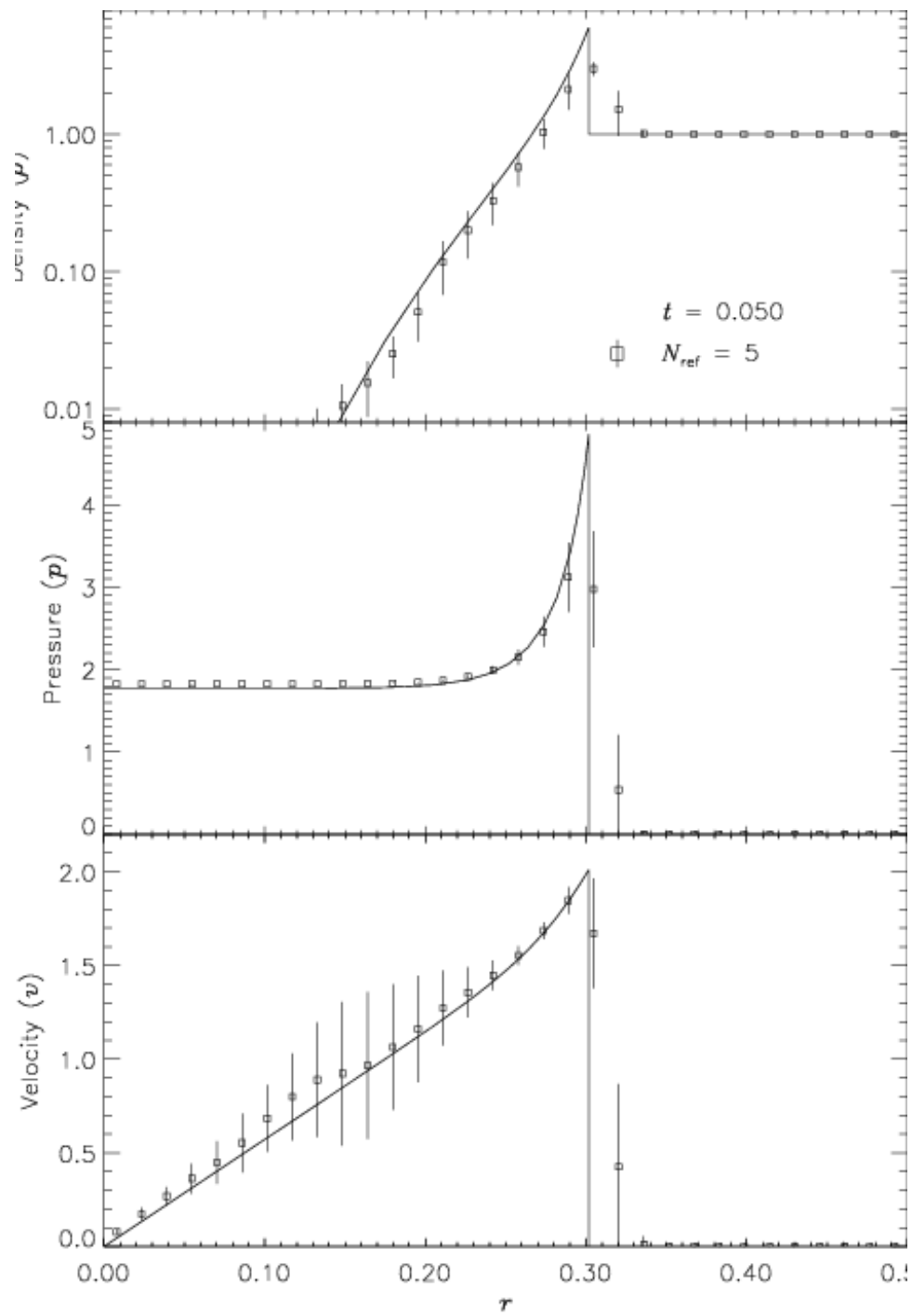
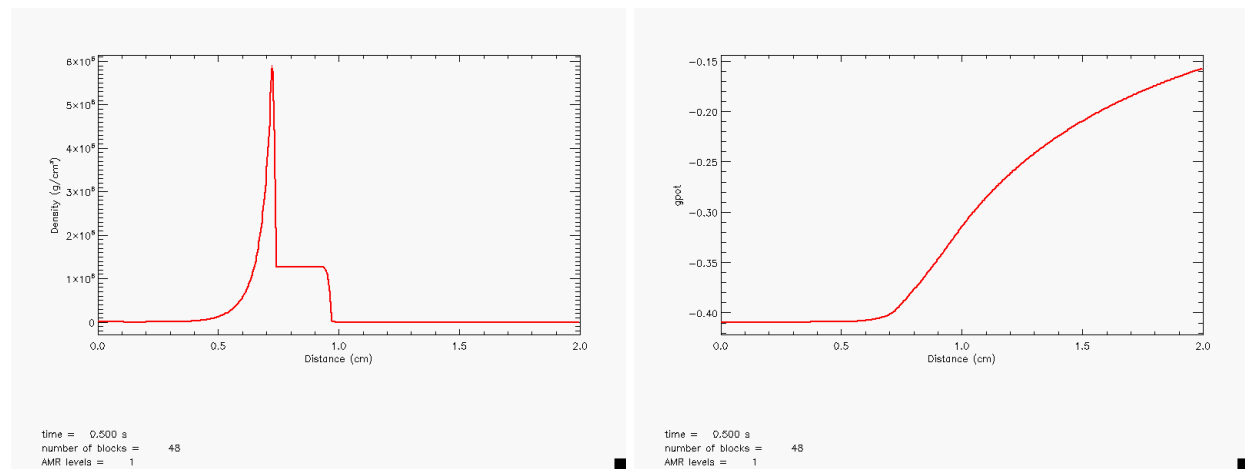
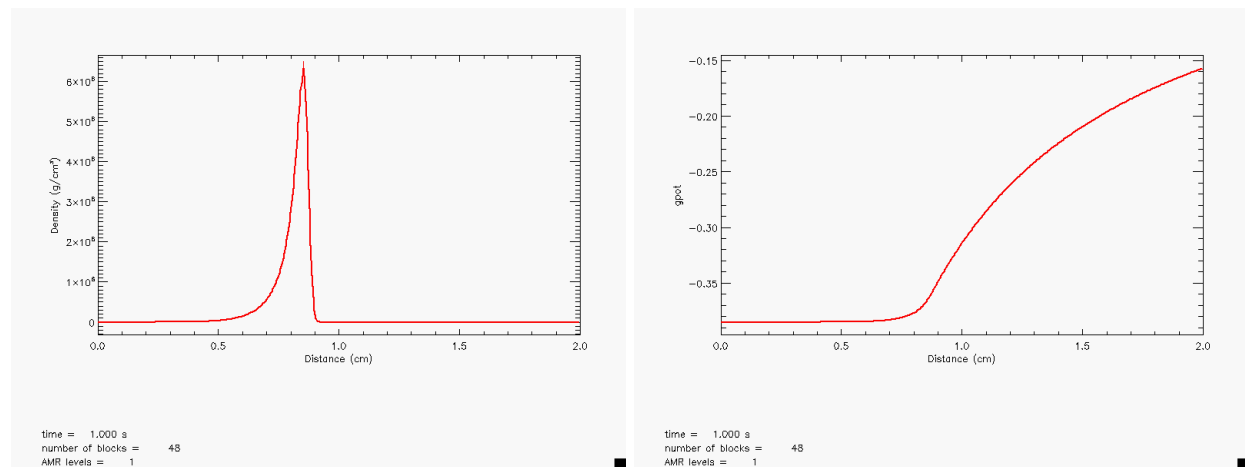


Figure 35.9: Comparison of numerical and analytical solutions versus radius  $r$  to the spherically symmetric Sedov problem. A 3D grid with five levels of refinement is used.

Table 35.3: Runtime parameters used with the Sedov test problem.

Variable	Type	Default	Description
<code>sim_pAmbient</code>	real	$10^{-5}$	Initial ambient pressure ( $p_0$ )
<code>sim_rhoAmbient</code>	real	1	Initial ambient density ( $\rho_0$ )
<code>sim_expEnergy</code>	real	1	Explosion energy ( $E$ )
<code>sim_rInit</code>	real	0.05	Radius of initial pressure perturbation ( $\delta r$ )
<code>sim_xctr</code>	real	0.5	$x$ -coordinate of explosion center
<code>sim_yctr</code>	real	0.5	$y$ -coordinate of explosion center
<code>sim_zctr</code>	real	0.5	$z$ -coordinate of explosion center
<code>sim_nSubZones</code>	integer	7	Number of sub-cells in cells for applying the 1D profile

Figure 35.10: Snapshots of Sedov Self-gravity density profile and gravitational potential at time  $t=0.5$  sec.Figure 35.11: Snapshots of Sedov Self-gravity density profile and gravitational potential at time  $t=1.0$  sec.

### 35.1.5 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 `Grid` runtime parameters `xmin`, `xmax`, `ymin`, and `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

$$\delta u = -y' \frac{\beta}{2\pi} \exp\left(\frac{1-r^2}{2}\right) \quad (35.8)$$

$$\delta v = x' \frac{\beta}{2\pi} \exp\left(\frac{1-r^2}{2}\right) \quad (35.9)$$

$$\delta T^* = -\frac{(\gamma-1)\beta}{8\gamma\pi^2} \exp(1-r^2) \quad (35.10)$$

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

$$T = \frac{T_\infty}{T_\infty^*} T^* \quad (35.11)$$

$$\rho = \rho_\infty \left(\frac{T}{T_\infty}\right)^{\frac{1}{\gamma-1}}. \quad (35.12)$$

At any location in space, the conserved variables (density,  $x$ - and  $y$ -momentum, and total energy) can be computed from the above quantities. The flow-field is initialized by computing cell averages of the conserved variables; in each cell, the average is approximated by averaging over `nx_subint`  $\times$  `ny_subint` subintervals. The runtime parameters for the isentropic vortex problem are listed in [Table 35.4](#).

Table 35.4: Parameters for the `IsentropicVortex` problem.

Variable	Type	Default	Description
<code>p_ambient</code>	real	1.0	Initial ambient pressure ( $p_\infty$ )
<code>rho_ambient</code>	real	1.0	Initial ambient density ( $\rho_\infty$ )
<code>u_ambient</code>	real	1.0	Initial ambient $x$ -velocity ( $u_\infty$ )
<code>v_ambient</code>	real	1.0	Initial ambient $y$ -velocity ( $v_\infty$ )
<code>vortex_strength</code>	real	5.0	Non-dimensional vortex strength
<code>xctr</code>	real	0.0	$x$ -coordinate of vortex center
<code>yctr</code>	real	0.0	$y$ -coordinate of vortex center
<code>nx_subint</code>	integer	10	number of subintervals in $x$ -direction
<code>ny_subint</code>	integer	10	number of subintervals in $y$ -direction

Figure 35.12 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  $\rho_{min} = 0.510287$ . Because the exact solution of the isentropic vortex problem is the initial solution shifted by  $(u_\infty t, v_\infty t)$ , numerical phase (dispersion) and amplitude (dissipation) errors are easy to identify. Dispersive errors distort the shape of the vortex, breaking its symmetry. Dissipative errors smooth the solution and flatten extrema; for the vortex, the minimum in density at the vortex core will increase.

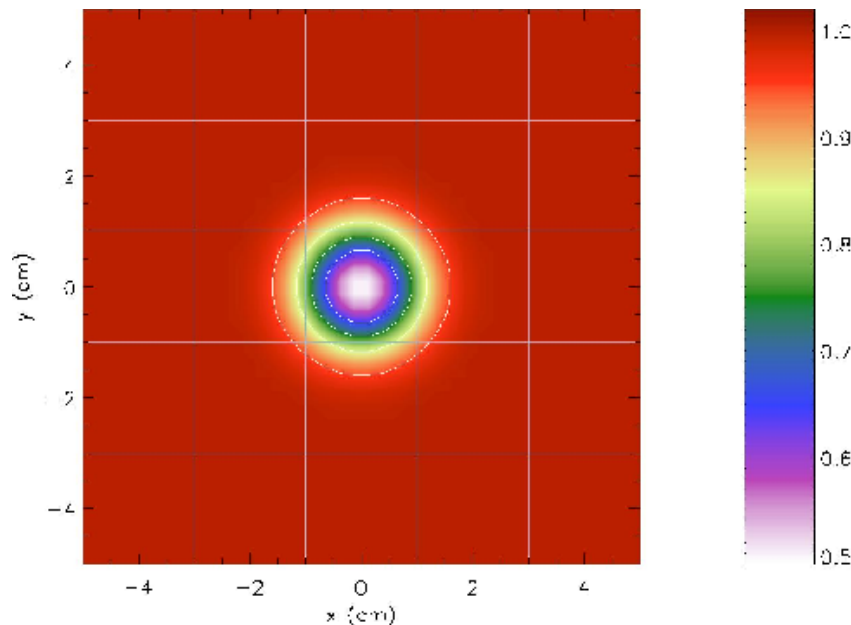


Figure 35.12: Density at  $t = 0.0$  for the isentropic vortex problem. Shown are the initial condition and the exact solution at  $t = 10.0, 20.0, \dots$

A numerical simulation using the PPM scheme was run to illustrate such errors. The simulation used the same grid shown in Figure 35.12 with the same contour levels and color values. The grid is intentionally coarse and the evolution time long to make numerical errors visible. The vortex is represented by approximately 8 grid points in each coordinate direction and is advected diagonally with respect to the grid. At solution times of  $t = 10, 20, \dots$ , *etc.*, the vortex should be back at its initial location.

Figure 35.13 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 35.14), and the minimum density is now 0.601786.

Figure 35.15 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.

### 35.1.6 The double Mach reflection problem

This numerical planar shock test problem introduced by Woodward and Colella (1984) simulates an evolution of an unsteady planar shock that is incident on an oblique surface. Initially, the incident planar shock begins to propagate to the bottom surface at a  $30^\circ$  angle with the shock Mach number of 10. Later, the solution to this problem produces a self-similar wave pattern that corresponds to the double Mach reflection. Following

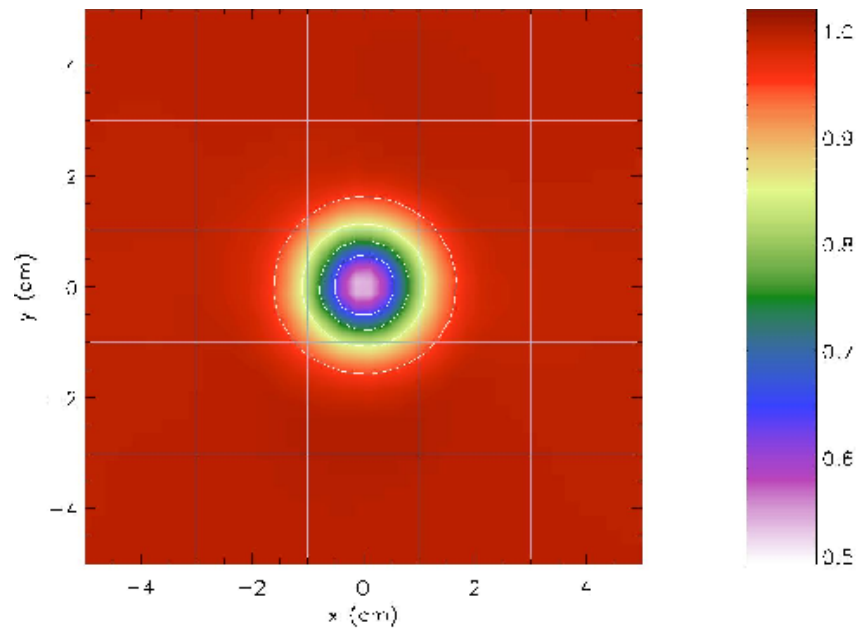


Figure 35.13: Density at  $t = 50.0$  for the isentropic vortex problem.

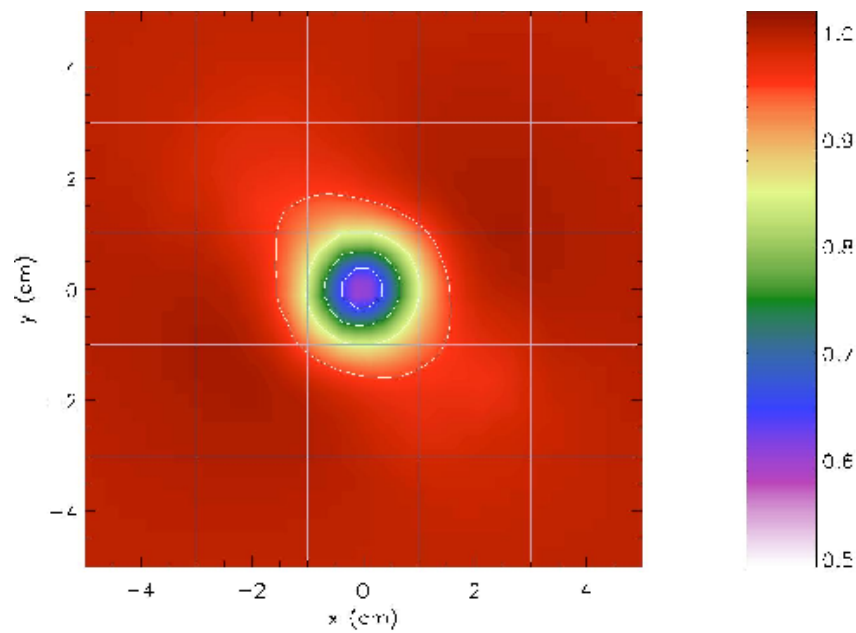


Figure 35.14: Density at  $t = 100.0$  for the isentropic vortex problem.

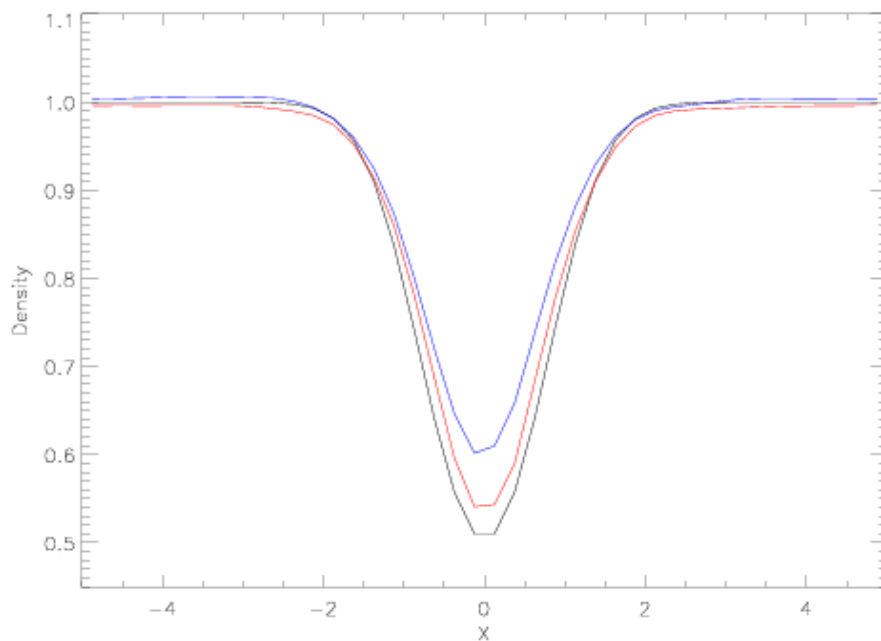


Figure 35.15: Representative density profiles for the isentropic vortex near  $y = 0.0$  at  $t = 0.0$  (black),  $t = 50.0$  (red), and  $t = 100.0$  (blue).

many other numerical setups of this problem, we tilt the incident shock rather than the reflecting wall so as to avoid numerical complications in modeling an oblique physical boundary.

The initial setup involves a Mach 10 shock in air,  $\gamma = 1.4$ , on a rectangular 2D domain of size  $[0, 4] \times [0, 1]$ . The reflecting wall is represented as the bottom surface of the domain, beginning at  $x = 1/6$ . The velocity normal to the incident shock in the post-shock region is 8.25, and the flow is at rest in the pre-shock region. The undisturbed air ahead of the shock has a density of 1.4 and a pressure of 1. See the initial density profile in [Figure 35.16](#) resolved on 6 refinement AMR levels using  $16^2$ -cell block size.

The boundary condition on  $0 \leq x \leq 1/6$  at  $y = 0$  is fixed in time with the initial values so that the reflected shock is attached to the bottom surface. We impose reflecting boundary condition (i.e., negating the  $y$  velocity field  $v$ ) on the rest of the bottom surface. On the top surface of  $y = 1$ , we allow the initial Mach 10 shock proceeds exactly as a function of time in order that the numerical evolution follows the oblique shock propagation without any planar distortion. At  $x = 0$ , we impose a supersonic inflow boundary condition, and the outflow condition at  $x = 4$ .

In later time, the solution develops to form two Mach stems and two contact discontinuities, as shown in [Figure 35.17](#) the density at  $t = 2.5$  sec. Also shown in [Figure 35.18](#) are 30 levels of contour lines of pressure, illustrating the evolved flow discontinuities at  $t = 2.5$  sec. We can also see that the numerical solution is smooth and non-oscillatory in the region bounded by the curved reflected shock, the bottom surface and the second Mach stem (see more discussion in Woodward and Colella, 1984).

The 5th order WENO method in the unsplit hydrodynamics solver is used with a choice of hybrid Riemann solver which selectively adopts HLL only at strong shocks and HLLC otherwise.

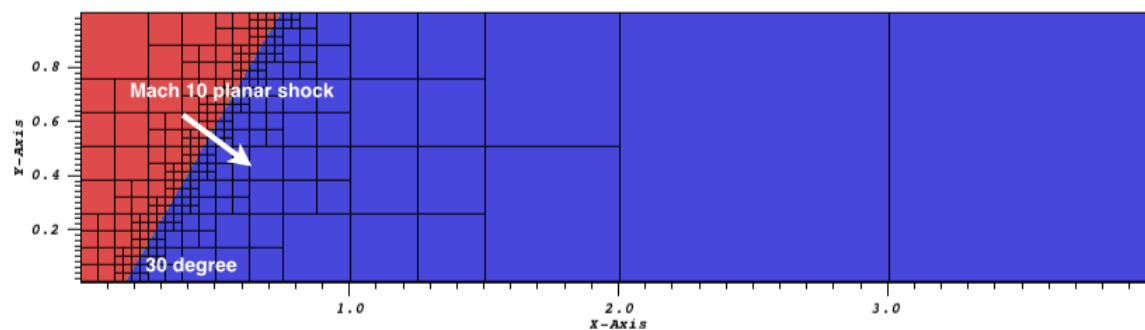


Figure 35.16: The initial density at  $t = 0$  visualized with 6 levels of AMR block structures.

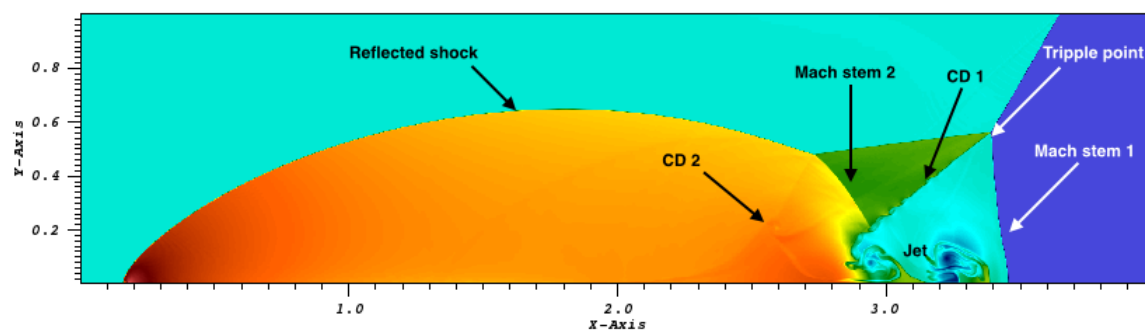


Figure 35.17: Density profile at  $t = 2.5$ . Two contact discontinuities are denoted as "CD", along with two Mach stems, a curved reflected shock, and a jet formulation of the denser fluid along the bottom surface.

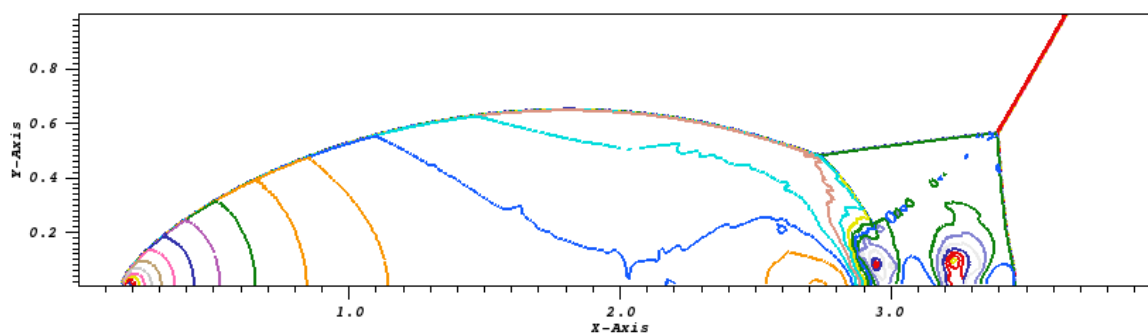


Figure 35.18: The 30 levels of contour lines of pressure at  $t = 2.5$ .

Table 35.5: Runtime parameters used with the `double Mach reflection` test problem.

Variable	Type	Default	Description
<code>sim_rhoLeft</code>	real	8	Initial density to the left of the shock ( $\rho_L$ )
<code>sim_rhoRight</code>	real	1.4	Initial density to the right ( $\rho_R$ )
<code>sim_pLeft</code>	real	116.5	Initial pressure to the left ( $p_L$ )
<code>sim_pRight</code>	real	1	Initial pressure to the right ( $p_R$ )
<code>sim_uLeft</code>	real	7.1447096	Initial $x$ -velocity to the left ( $u_L$ )
<code>sim_uRight</code>	real	0	Initial $x$ -velocity to the right ( $u_R$ )
<code>sim_vLeft</code>	real	-4.125	Initial $y$ -velocity to the left ( $v_L$ )
<code>sim_vRight</code>	real	0	Initial $y$ -velocity to the right ( $v_R$ )
<code>sim_xangle</code>	real	-30	Angle made by shock normal with the $x$ -axis (degrees)
<code>sim_yangle</code>	real	90	Angle made by shock normal with the $y$ -axis (degrees)
<code>sim_posn</code>	real	1/6	Point of intersection between the shock plane and the $x$ -axis

### 35.1.7 Wind Tunnel With a Step

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

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

$$\rho_0 = 1.4, \quad p_0 = 1, \quad u_0 = 3, \quad \gamma = 1.4, \quad (35.13)$$

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

The additional runtime parameters supplied with the `WindTunnel` problem are listed in [Table 35.6](#). This problem is configured to use the perfect-gas equation of state (`Gamma`) with `gamma` set to 1.4. We also set `xmax = 3`, `ymin = 1`, `Nblockx = 15`, and `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 `REFLECTING` boundaries to the obstacle blocks. Finally, we use `xl.boundary.type = "user"` (`USER_DEFINED` condition) and `xr.boundary.type = "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 35.19](#) shows the evolution of density and velocity between  $t = 0$  and  $t = 4$  (the period considered by Woodward and Colella). Immediately, a shock forms directly in front of the step and begins to move slowly away from it. Simultaneously, the shock curves around the corner of the step, extending farther downstream and growing in size until it strikes the upper boundary just after  $t = 0.5$ . The corner of the step becomes a singular point, with a rarefaction fan connecting the still gas just above the step to the shocked gas in front of it. Entropy errors generated in the vicinity of this singular point produce a numerical boundary layer about one cell thick along the surface of the step. Woodward and Colella reduce this effect by resetting the cells immediately behind the corner to conserve entropy and the sum of enthalpy



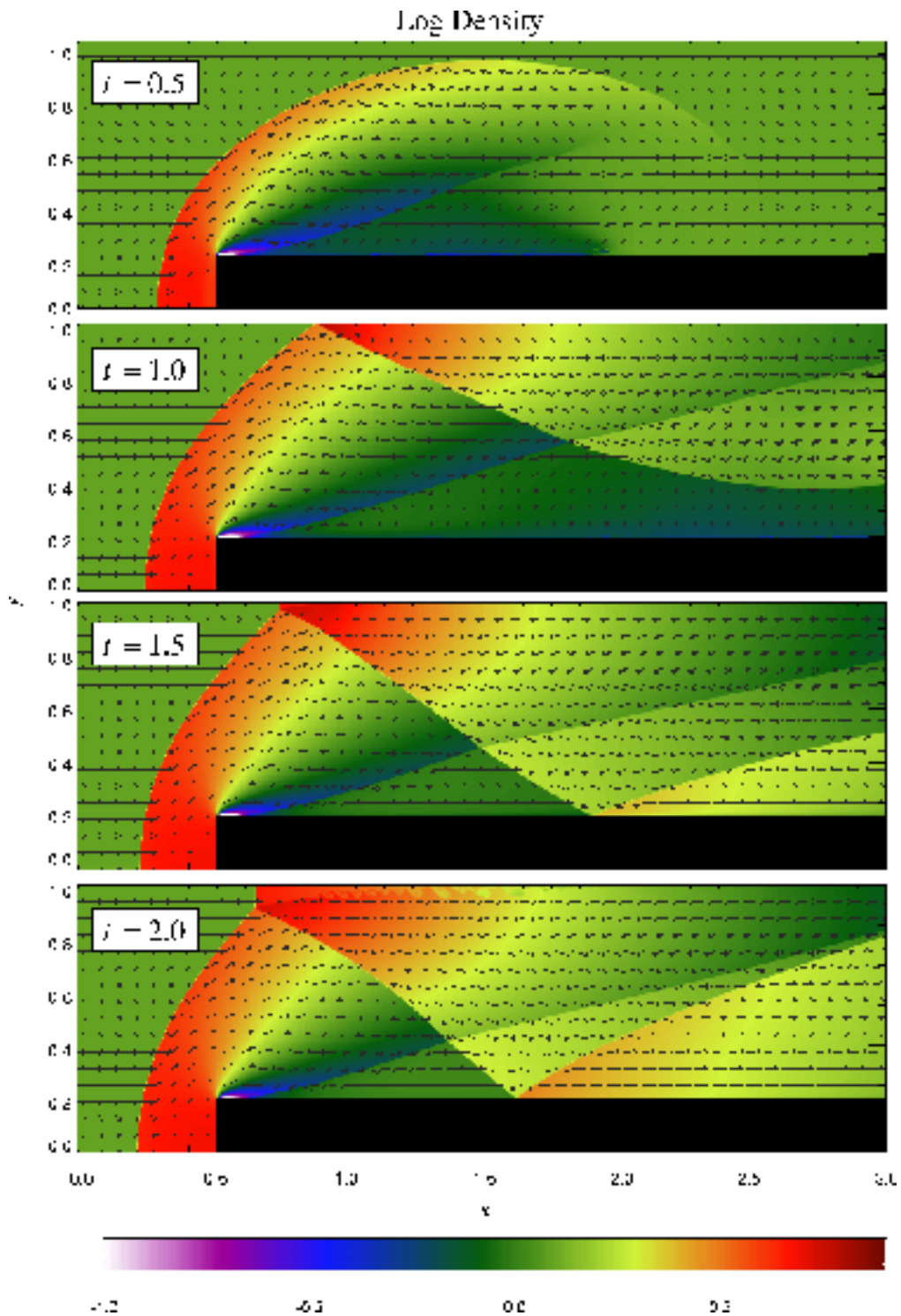


Figure 35.19: Density and velocity in the Emery wind tunnel test problem, as computed with FLASH. A 2D grid with five levels of refinement is used.

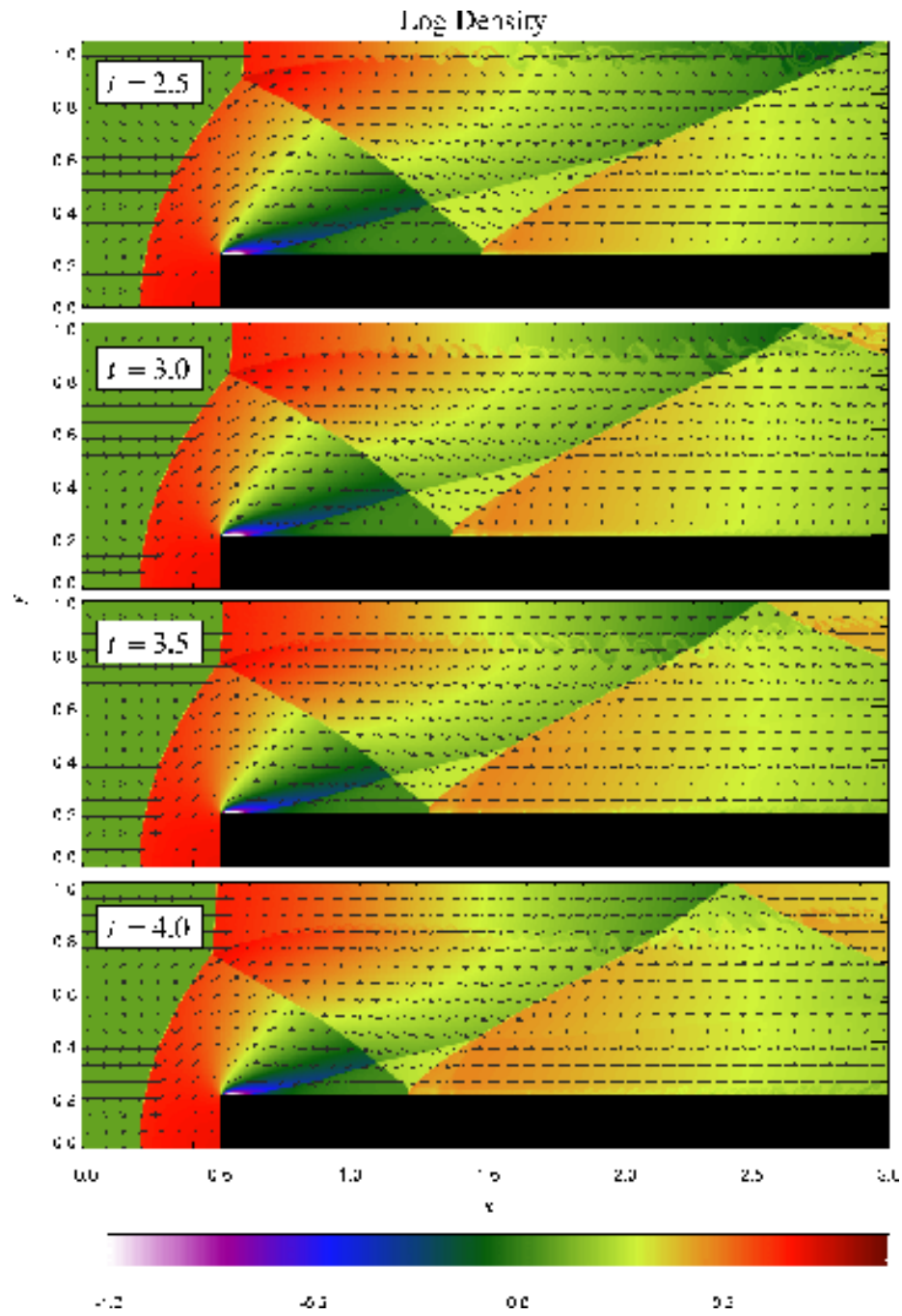


Figure 35.19: Density and velocity in the Emery wind tunnel test problem (continued).

Table 35.6: Runtime parameters used with the `WindTunnel` test problem.

Variable	Type	Default	Description
<code>sim_pAmbient</code>	real	1	Ambient pressure ( $p_0$ )
<code>sim_rhoAmbient</code>	real	1.4	Ambient density ( $\rho_0$ )
<code>sim_windVel</code>	real	3	Inflow velocity ( $u_0$ )

and specific kinetic energy through the rarefaction. However, we are less interested here in reproducing the exact solution than in verifying the code and examining the behavior of such numerical effects as resolution is increased, so we do not apply this additional boundary condition. The errors near the corner result in a slight over-expansion of the gas there and a weak oblique shock where this gas flows back toward the step. At all resolutions we also see interactions between the numerical boundary layer and the reflected shocks that appear later in the calculation.

The shock reaches the top wall at  $t \approx 0.65$ . The point of reflection begins at  $x \approx 1.45$  and then moves to the left, reaching  $x \approx 0.95$  at  $t = 1$ . As it moves, the angle between the incident shock and the wall increases until  $t = 1.5$ , at which point it exceeds the maximum angle for regular reflection ( $40^\circ$  for  $\gamma = 1.4$ ) and begins to form a Mach stem. Meanwhile the reflected shock has itself reflected from the top of the step, and here too the point of intersection moves leftward, reaching  $x \approx 1.65$  by  $t = 2$ . The second reflection propagates back toward the top of the grid, reaching it at  $t = 2.5$  and forming a third reflection. By this time in low-resolution runs, we see a second Mach stem forming at the shock reflection from the top of the step; this results from the interaction of the shock with the numerical boundary layer, which causes the angle of incidence to increase faster than in the converged solution. [Figure 35.20](#) compares the density field at  $t = 4$  as computed by FLASH using several different maximum levels of refinement. Note that the size of the artificial Mach reflection diminishes as resolution improves.

The shear cell behind the first (“real”) Mach stem produces another interesting numerical effect, visible at  $t \geq 3$  — Kelvin-Helmholtz amplification of numerical errors generated at the shock intersection. The waves thus generated propagate downstream and are refracted by the second and third reflected shocks. This effect is also seen in the calculations of Woodward and Colella, although their resolution was too low to capture the detailed eddy structure we see. [Figure 35.21](#) 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.

### 35.1.8 The Shu-Osher problem

The Shu-Osher problem (Shu and Osher, 1989) tests a shock-capturing scheme’s ability to resolve small-scale flow features. It gives a good indication of the numerical (artificial) viscosity of a method. Since it is designed to test shock-capturing schemes, the equations of interest are the one-dimensional Euler equations for a single-species perfect gas.

In this problem, a (nominally) Mach 3 shock wave propagates into a sinusoidal density field. As the shock advances, two sets of density features appear behind the shock. One set has the same spatial frequency as the unshocked perturbations, but for the second set, the frequency is doubled. Furthermore, the second set

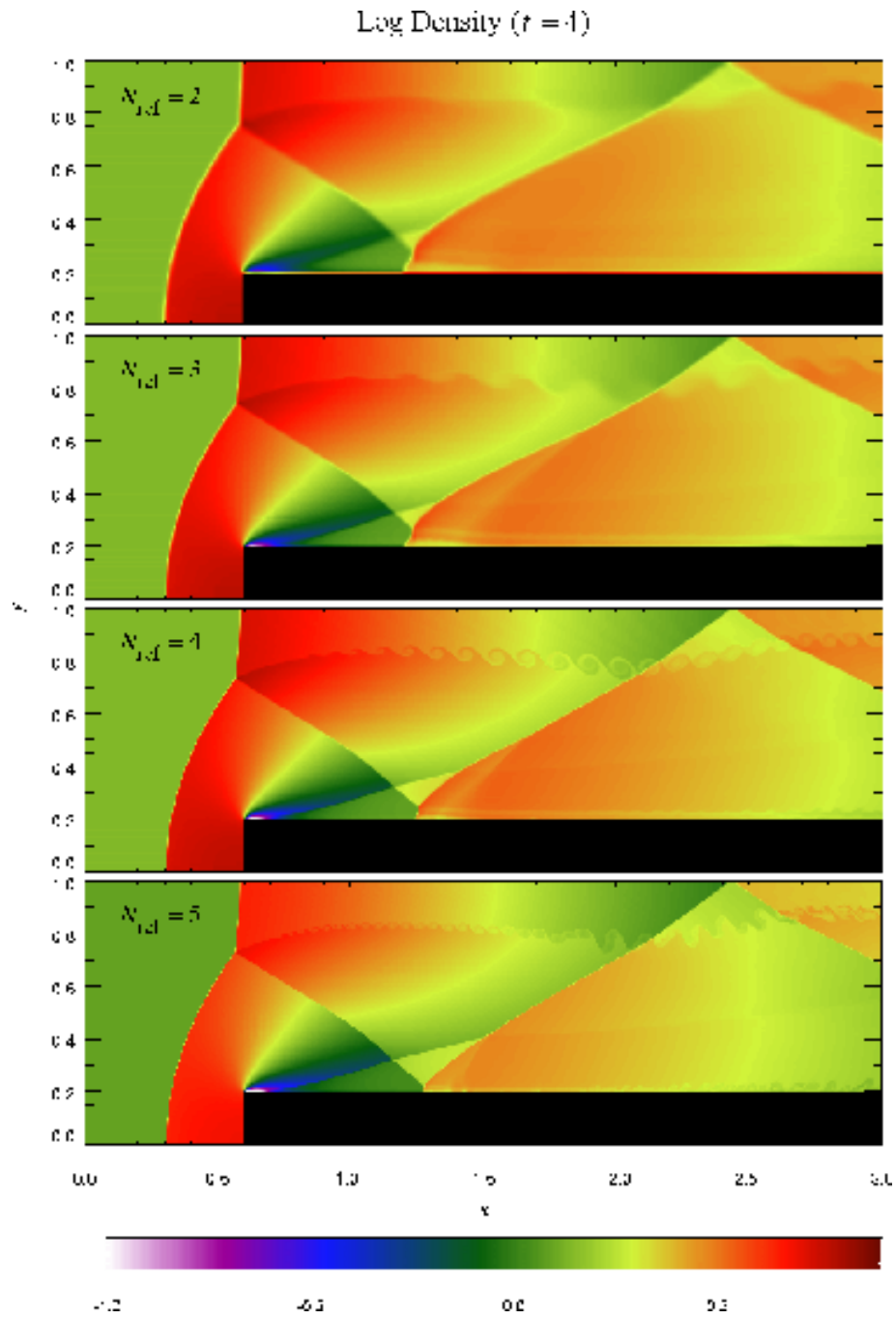


Figure 35.20: Density at  $t = 4$  in the Emery wind tunnel test problem, as computed with FLASH using several different levels of refinement.

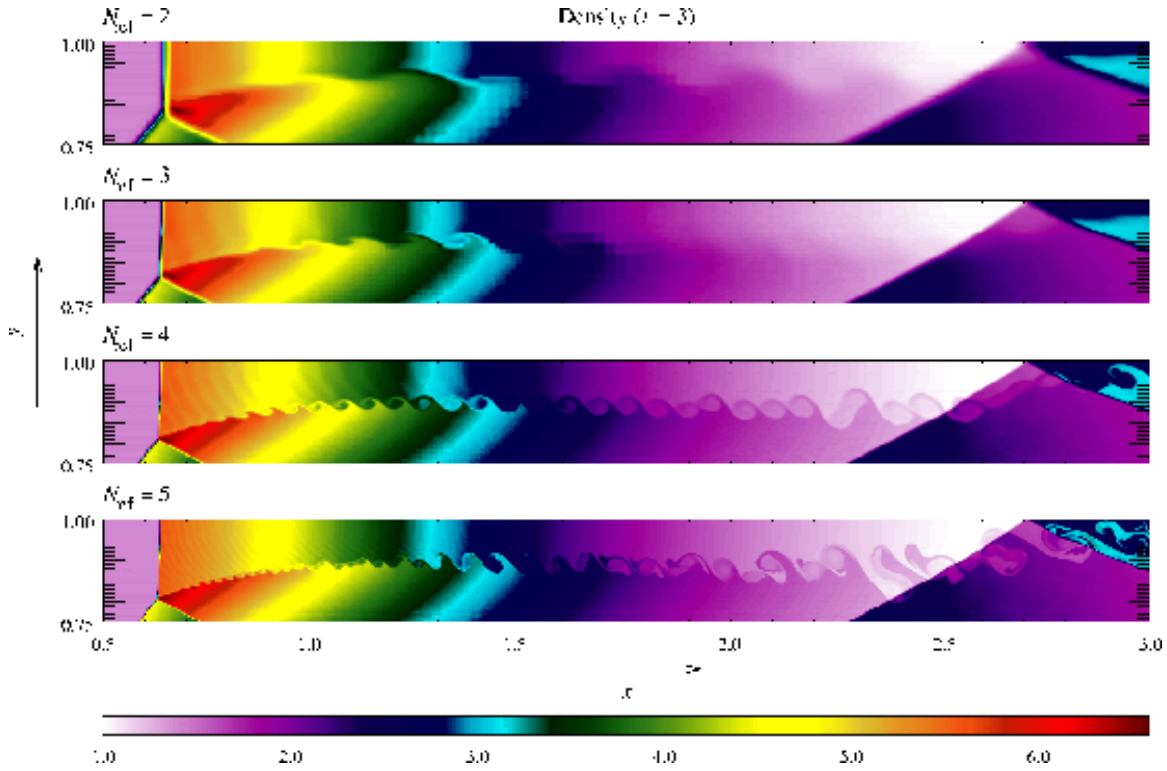


Figure 35.21: Detail of the Kelvin-Helmholtz instability seen at  $t = 3$  in the Emery wind tunnel test problem for several different levels of refinement.

follows more closely behind the shock. None of these features is spurious. The test of the numerical method is to accurately resolve the dynamics and strengths of the oscillations behind the shock.

The `shu_oshier` problem is initialized as follows. On the domain  $-4.5 \leq x \leq 4.5$ , the shock is at  $x = x_s$  at  $t = 0.0$ . On either side of the shock,

$$\begin{array}{ccc}
 & x \leq x_s & x > x_s \\
 \rho(x) & \rho_L & \rho_R (1.0 + a_\rho \sin(f_\rho x)) \\
 p(x) & p_L & p_R \\
 u(x) & u_L & u_R
 \end{array} \tag{35.14}$$

where  $a_\rho$  is the amplitude and  $f_\rho$  is the frequency of the density perturbations. The `gamma` equation of state alternative implementation is used with `gamma` set to 1.4. The runtime parameters and their default values are listed in Table 35.7. The initial density,  $x$ -velocity, and pressure distributions are shown in Figure 35.22.

Table 35.7: Runtime parameters used with the `shu_oshier` test problem.

Variable	Type	Default	Description
<code>posn</code>	real	-4.0	Initial shock location ( $x_s$ )
<code>rho_left</code>	real	3.857143	Initial density to the left of the shock ( $\rho_L$ )
<code>rho_right</code>	real	1.0	Nominal initial density to the right ( $\rho_R$ )
<code>p_left</code>	real	10.33333	Initial pressure to the left ( $p_L$ )
<code>p_right</code>	real	1.0	Initial pressure to the right ( $p_R$ )
<code>u_left</code>	real	2.629369	Initial velocity to the left ( $u_L$ )
<code>u_right</code>	real	0.0	Initial velocity to the right ( $u_R$ )
<code>a_rho</code>	real	0.2	Amplitude of the density perturbations

<code>f_rho</code>	real	5.0	Frequency of the density perturbations
--------------------	------	-----	--

The problem is strictly one-dimensional; building 2-d or 3-d executables should give the same results along each  $x$ -direction grid line. For this problem, special boundary conditions are applied. The initial conditions should not change at the boundaries; if they do, errors at the boundaries can contaminate the results. To avoid this possibility, a boundary condition subroutine was written to set the boundary values to their initial values.

The purpose of the tests is to determine how much resolution, in terms of mesh cells per feature, a particular method requires to accurately represent small scale flow features. Therefore, all computations are carried out on equispaced meshes *without* adaptive refinement. Solutions are obtained at  $t = 1.8$ . The reference solution, using 3200 mesh cells, is shown in [Figure 35.23](#). This solution was computed using PPM at a CFL number of 0.8. Note the shock located at  $x \simeq 2.4$ , and the high frequency density oscillations just to the left of the shock. When the grid resolution is insufficient, shock-capturing schemes underpredict the amplitude of these oscillations and may distort their shape.

[Figure 35.24](#) shows the density field for the same scheme at 400 mesh cells and at 200 mesh cells. With 400 cells, the amplitudes are only slightly reduced compared to the reference solution; however, the shapes of the oscillations have been distorted. The slopes are steeper and the peaks and troughs are broader, which is the result of overcompression from the contact-steepening part of the PPM algorithm. For the solution on 200 mesh cells, the amplitudes of the high-frequency oscillations are significantly underpredicted.

### 35.1.9 Driven Turbulence StirTurb

The driven turbulence problem **StirTurb** simulates homogeneous, isotropic and weakly-compressible turbulence. Because theories of turbulence generally assume a steady state, and because turbulence is inherently a dissipative phenomenon, the fluid must be driven to sustain a steady-state. This driving must be done carefully in order to avoid introducing artifacts into the turbulent flow. We use a relatively sophisticated stochastic driving method originally introduced by Eswaran & Pope (1988). The initial conditions sets up a homogeneous background. The resolution used for this test run was  $32^3$ , and the boundary conditions were periodic. The [Table 35.8](#) shows values the runtime parameters values to control the amount of driving, and the [Figures 35.25](#) and [Figure 35.26](#) show the density and  $x$  velocity profile of an  $xy$  plane in the center of the domain.

### 35.1.10 Relativistic Sod Shock-Tube

The relativistic version of the shock tube problem (**RHD.Sod**) is a simple one-dimensional setup that involves the decay of an initially discontinuous two fluids into three elementary wave structures: a shock, a contact, and a rarefaction wave. As in Newtonian hydrodynamics case, this type of problem is useful in addressing the ability of the Riemann solver to check the code correctness in evolving such simple elementary waves.

We construct the initial conditions for the relativistic shock tube problem as found in Martí & Müller (2003). We use an ideal equation of state with  $\Gamma = 5/3$  for this problem and the left and right states are

Table 35.8: Runtime parameters used with the **Driven Turbulence** test problem.

Variable	Type	Value	Description
<code>st_stirmax</code>	real	25.1327	maximum stirring wavenumber
<code>st_stirmin</code>	real	6.2832	minimum stirring wavenumber
<code>st_energy</code>	real	5.E-6	energy input per mode
<code>st_decay</code>	real	0.5	correlation time for driving
<code>st_freq</code>	integer	1	frequency of stirring

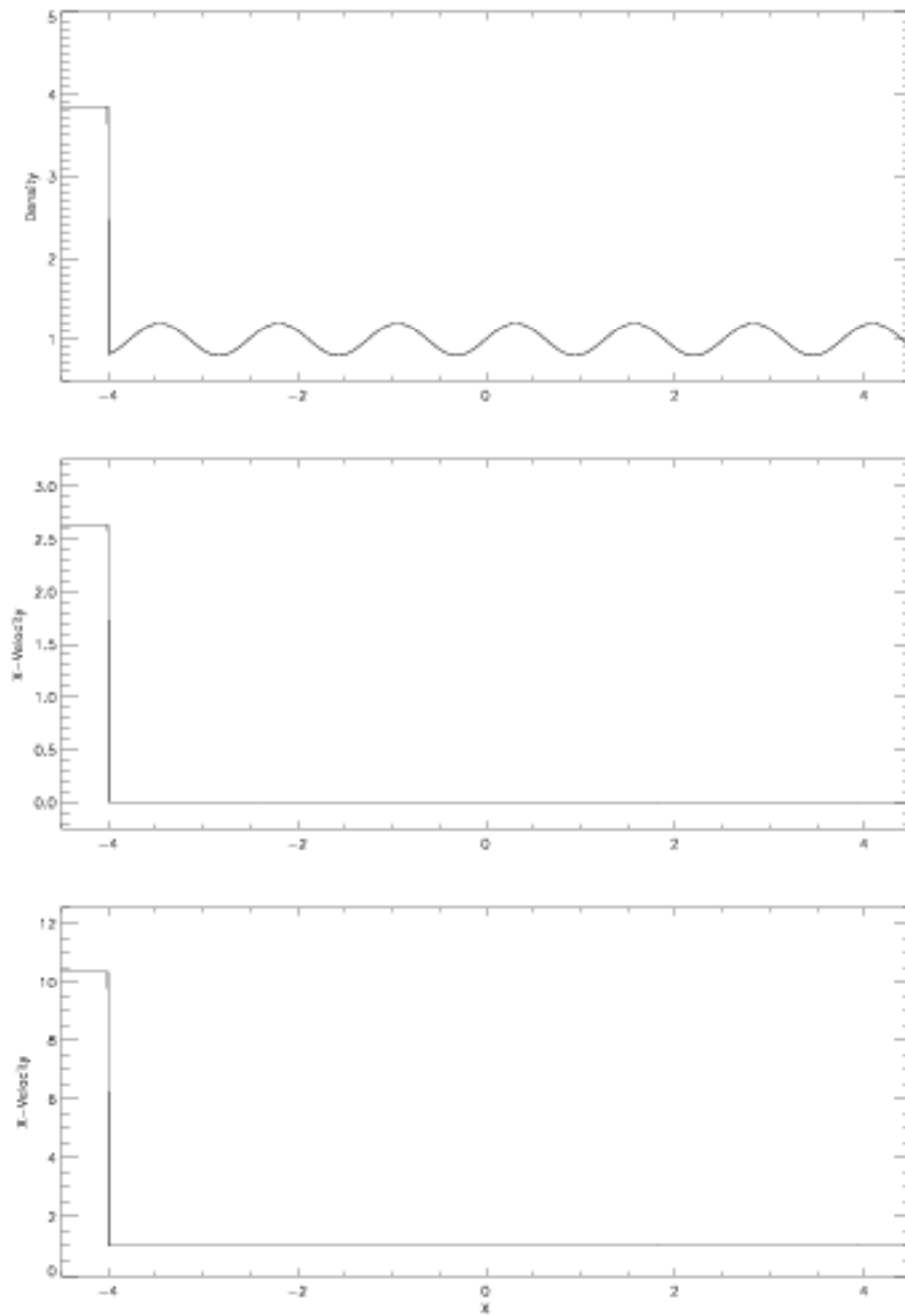


Figure 35.22: Initial density,  $x$ -velocity, and pressure for the Shu-Osher problem.

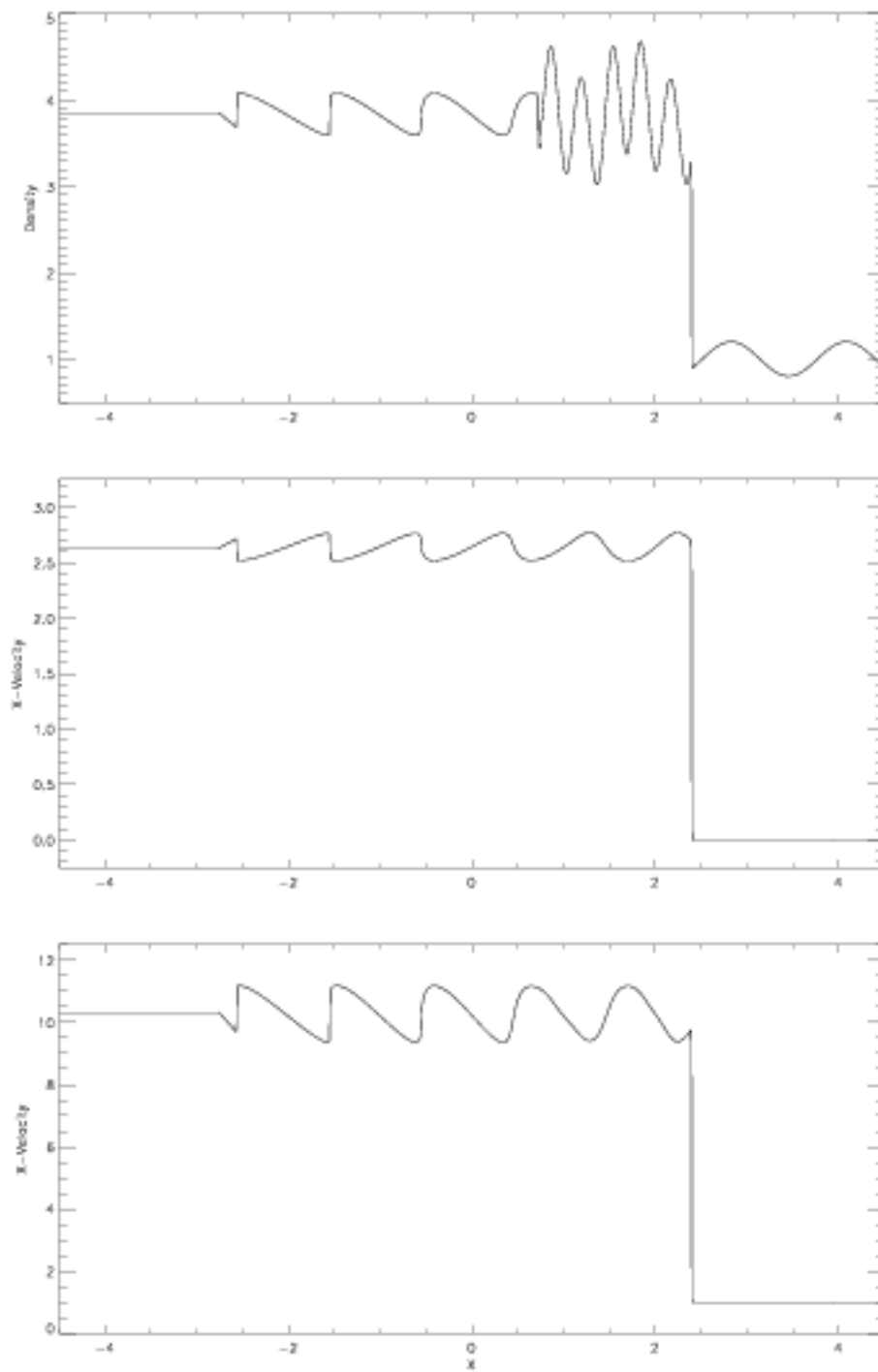


Figure 35.23: Density,  $x$ -velocity, and pressure for the reference solution at  $t = 1.8$ .



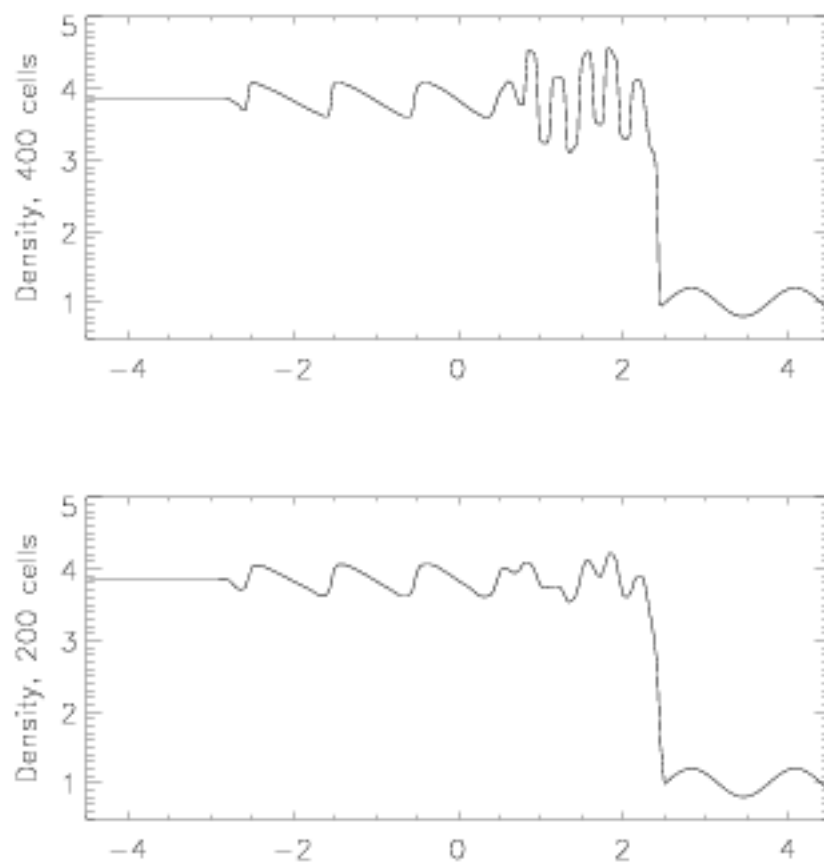


Figure 35.24: Density fields on 400 and 200 mesh cells from the PPM scheme.

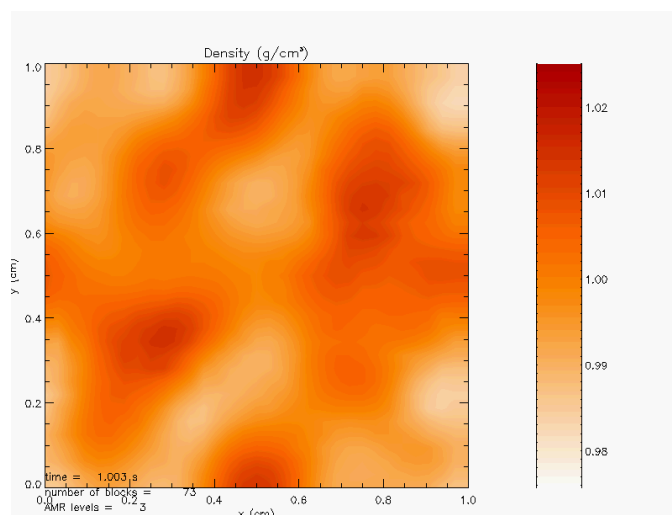


Figure 35.25: Density profile for the **StirTurb** driven turbulence problem.

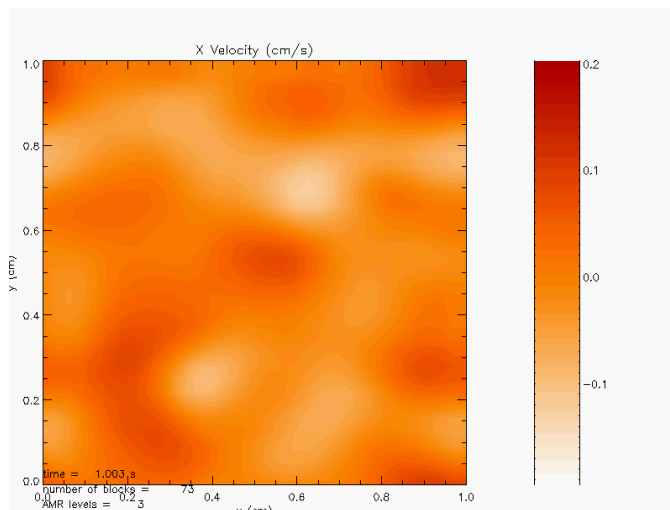


Figure 35.26: velocity along X dimension for the **StirTurb** driven turbulence problem.

given:

$$\begin{aligned} \rho_L &= 10 & \rho_R &= 1.0 \\ p_L &= 40/3 & p_R &= 2/3 \times 10^{-6} \end{aligned} \quad (35.15)$$

and the computational domain is  $0 \leq x \leq 1$ . The initial shock location is at  $x = 0.5$  (halfway across a box with unit dimensions).

In FLASH, the RHD Sod problem (**RHD\_Sod**) uses the runtime parameters listed in [Table 35.9](#):

Table 35.9: Runtime parameters used with the **RHD\_Sod** test problem.

Variable	Type	Default	Description
<b>sim_rhoLeft</b>	real	10	Initial density to the left of the interface ( $\rho_L$ )
<b>sim_rhoRight</b>	real	1.0	Initial density to the right ( $\rho_R$ )
<b>sim_pLeft</b>	real	40/3	Initial pressure to the left ( $p_L$ )
<b>sim_pRight</b>	real	$2/3 \times 10^{-6}$	Initial pressure to the right ( $p_R$ )
<b>sim_uLeft</b>	real	0	Initial velocity (perpendicular to interface) to the left ( $u_L$ )
<b>sim_uRight</b>	real	0	Initial velocity (perpendicular to interface) to the right ( $u_R$ )
<b>sim_xangle</b>	real	0	Angle made by interface normal with the $x$ -axis (degrees)
<b>sim_yangle</b>	real	90	Angle made by interface normal with the $y$ -axis (degrees)
<b>sim_posn</b>	real	0.5	Point of intersection between the interface plane and the $x$ -axis

[Figure 35.27](#), [Figure 35.28](#), and [Figure 35.29](#) show the results of running the RHD Sod problem on a one-dimensional uniform grid of size 400 at simulation time  $t = 0.36$ . In this run the left-going wave is the rarefaction wave, while two right-going waves are the contact discontinuity and the shock wave. This configuration results in mildly relativistic effects that are mainly thermodynamical in nature.

The differences in the relativistic regime, as compared to Newtonian hydrodynamics, can be seen in a curved velocity profile for the rarefaction wave and the narrow constant state (density shell) in between

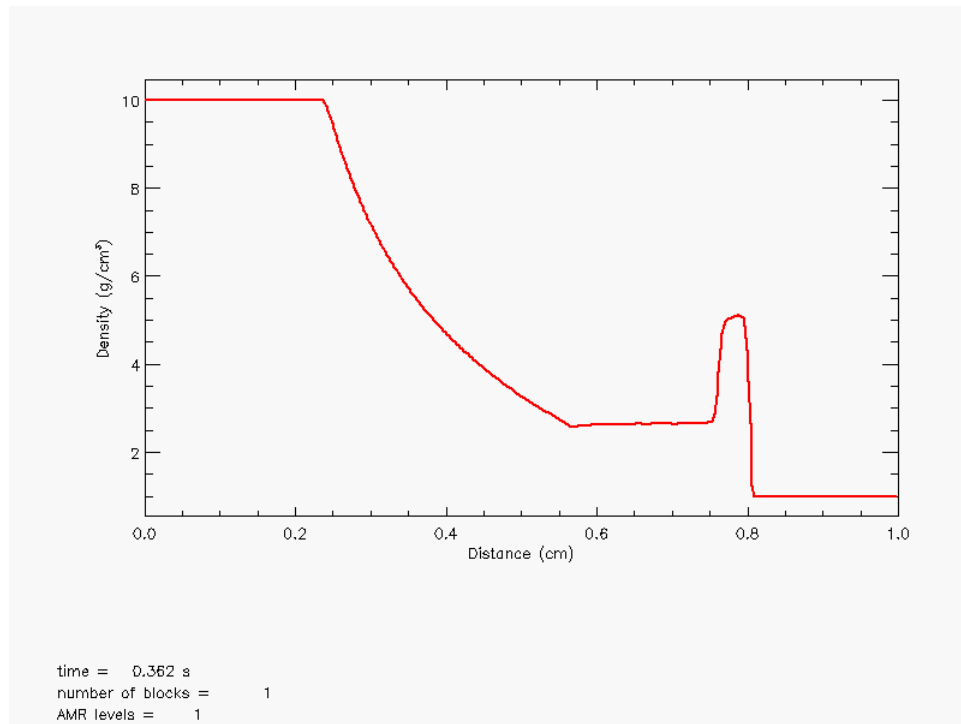


Figure 35.27: Density of numerical solution to the relativistic Sod problem at time  $t = 0.36$ .

the shock wave and contact discontinuity. Numerically, it is particularly challenging to resolve the thin narrow density plateau, which is bounded by a leading shock front and a trailing contact discontinuity (see [Figure 35.27](#)).

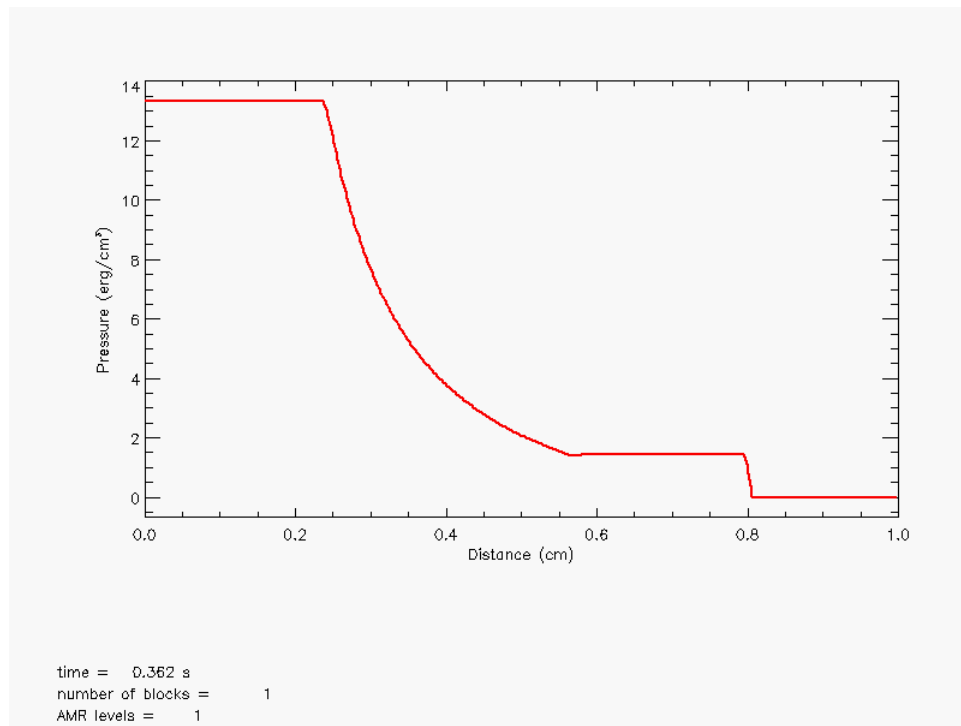


Figure 35.28: Pressure of numerical solution to the relativistic Sod problem at time  $t = 0.36$ .

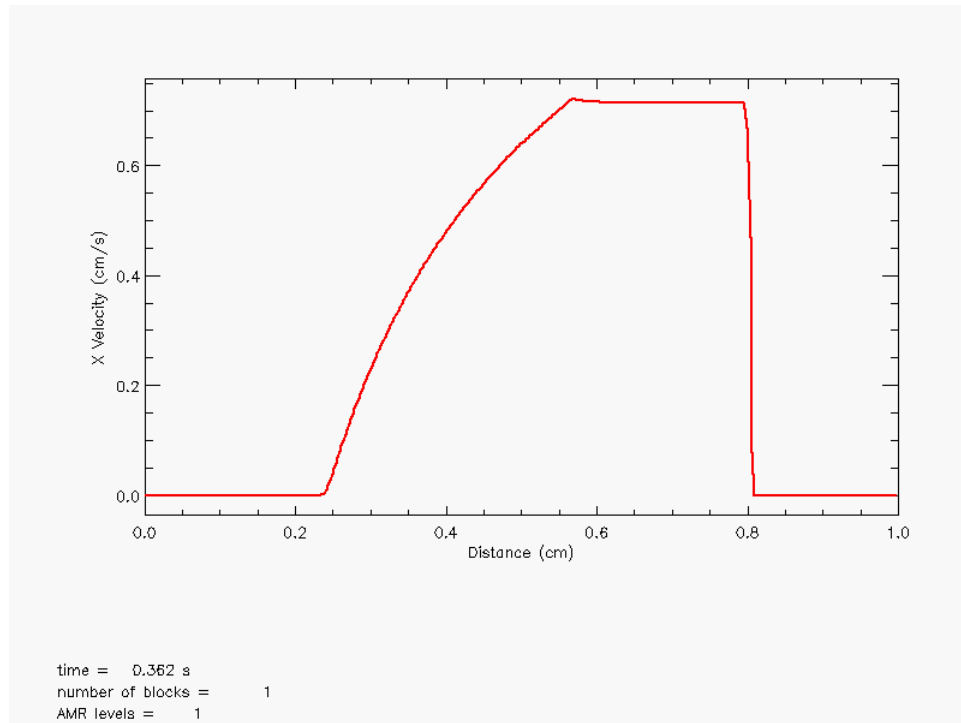


Figure 35.29: Normal velocity of numerical solution to the relativistic Sod problem at time  $t = 0.36$ .

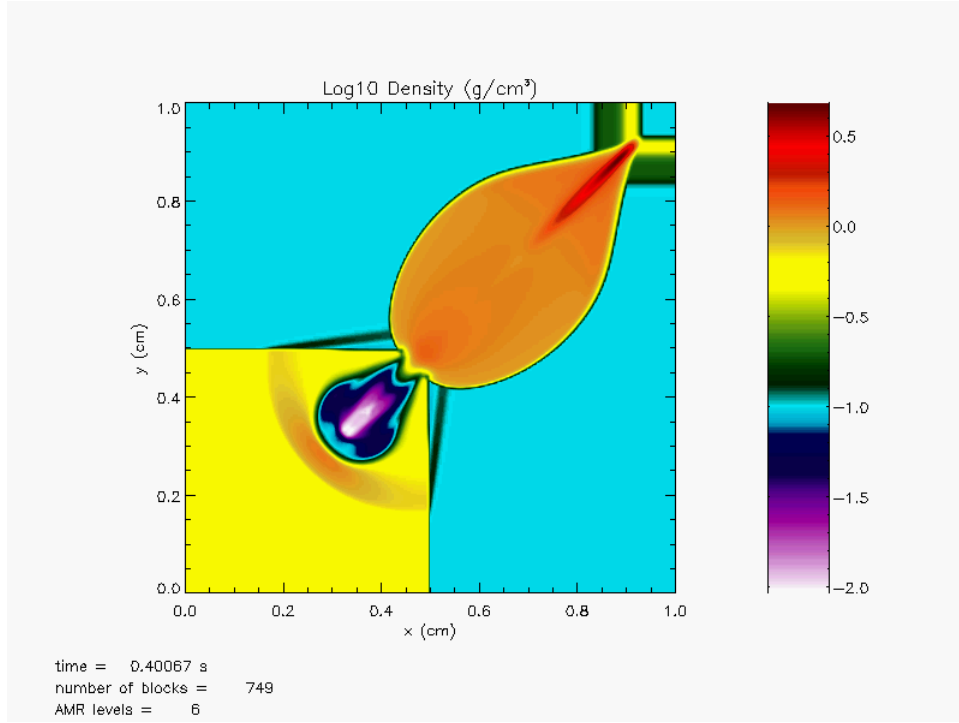


Figure 35.30: Log of density of numerical solution to the relativistic 2D Riemann problem at time  $t = 0.8$ . The solution was resolved on AMR grid with 6 levels of refinements

### 35.1.11 Relativistic Two-dimensional Riemann

The two-dimensional Riemann problem (RHD\_Riemann2D), originally proposed by Schulz et al. (1993), involves studying interactions of four basic waves that consist of shocks, rarefactions, and contact discontinuities. The initial condition provided here is based on Migone et al. (2005) producing these elementary waves at every interface. The setup of the problem is given on a rectangular domain of size  $[-1, 1] \times [-1, 1]$ , which is divided into four constant state subregions as:

$$(\rho, p, v_x, v_y) = \begin{cases} (0.5, 1.0, 0.0, 0.0) & -1.0 \leq x < 0.0, \quad -1.0 \leq y < 0.0 \\ (0.1, 1.0, 0.0, 0.99) & 0.0 \leq x \leq 1.0, \quad -1.0 \leq y < 0.0 \\ (0.1, 1.0, 0.99, 0.0) & -1.0 \leq x < 0.0, \quad 0.0 \leq y \leq 1.0 \\ (\rho_1, p_1, 0.0, 0.0) & 0.0 \leq x \leq 1.0, \quad 0.0 \leq y \leq 1.0 \end{cases}, \quad (35.16)$$

where  $\rho_1 = 5.477875 \times 10^{-3}$  and  $p_1 = 2.762987 \times 10^{-3}$ . An ideal EOS is used with the specific heat ratio  $\Gamma = 5/3$ .

The solution obtained at  $t = 0.8$  in Figure 35.30 shows that the symmetry of the problem is well maintained. The two shocks are propagated from the upper left and the lower right regions to the upper right region, yielding continuous collisions of shocks at the upper right corner. The curved shock fronts are transmitted and formed in the diagonal region of the domain. The lower left region is bounded by contact discontinuities. By the time  $t = 0.8$  most of regions are filled with shocked gas, whereas there are still two unperturbed regions in the lower left and upper right regions.

### 35.1.12 Flow Interactions with Stationary Rigid Body

The stationary rigid body is only implemented and tested in the unsplit hydro solver [Section 15.1.3](#). It is possible that the unsplit staggered mesh MHD solver [Section 15.3.3](#) can support the rigid body but we have not tested yet.

#### 35.1.12.1 NACA Airfoil

Flow simulations over a series of NACA airfoils (or a simple flat plate) can be obtained using the implementation of a stationary rigid body in the unsplit hydro solver described in [Section 15.1.3.1](#). In this example, the cambered NACA2412 airfoil is initialized with positive unity values indicating the part of the domain that belongs to a stationary rigid body. The rest of the domain is assigned negative unity values to indicate it as a flow region for the unsplit hydro solver. At the surface of the rigid body, a reflecting boundary condition is applied in order to represent the fact that there is no flow penetrating the rigid object.

Plots in [35.31\(a\)](#) – [35.33\(b\)](#) illustrate Mach number and pressure plots over the airfoil with the three different initial Mach numbers, 0.65, 0.95 and 1.2 at  $t = 1.8$ . By this time, the flow conditions have reached their steady states. For Mach number = 0.65, the critical Mach number has not yet been obtained and the flow over the airfoil is all subsonic as shown in [35.31\(a\)](#). Since the airfoil is asymmetric and cambered, we see there are pressure gradients across the top and bottom surfaces even at zero angle of attack in [35.31\(b\)](#). These pressure gradients (higher pressure at the bottom than the top) generate a lift force by which an airplane can fly defying gravity.

At Mach number reaching 0.95 as shown in [35.32\(a\)](#) there are local points that are supersonic. This indicates that the critical Mach number for the airfoil is between 0.65 and 0.95. In fact, one can show that the critical Mach number is around 0.7 for the NACA2412 airfoil. We see that there is a development of a bow shock formation in front of the airfoil. A formation of a subsonic region between the bow shock and the nose of the airfoil is visible in the Mach number plot. Inside the bow shock, a sonic line at which the local flow speed becomes the sound speed makes an oval shape together with the bow shock. In both the Mach number and pressure plots, a strong wake forms starting from the top and bottom of the surfaces near the trailing edge. The wake is hardly visible for Mach number 0.65 in [35.31\(a\)](#) and [35.31\(b\)](#). Normal shock waves have formed stemming from the trailing edge as seen in [35.32\(a\)](#) and [35.32\(b\)](#).

At Mach number 1.2 the flow becomes supersonic everywhere. In this case, the shape of the bow shock becomes narrower and there are much larger supersonic pockets developed on the top and bottom surfaces with a smaller subsonic region between the bow shock and the nose region.

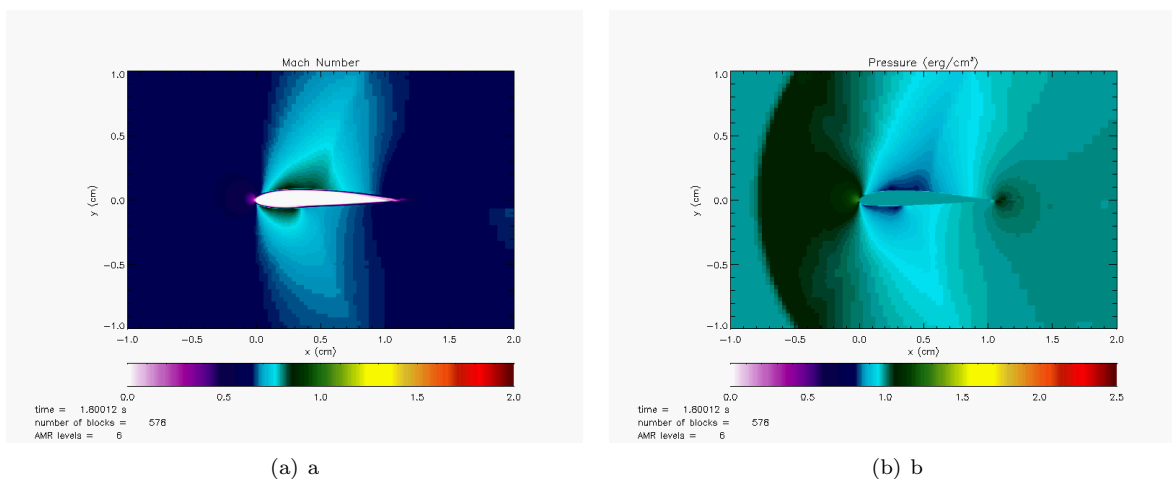


Figure 35.31: NACA2412 in Mach number 0.65 flow at 0 degree angle of attack problem at  $t = 1.8$  (a) Mach number (b) Pressure

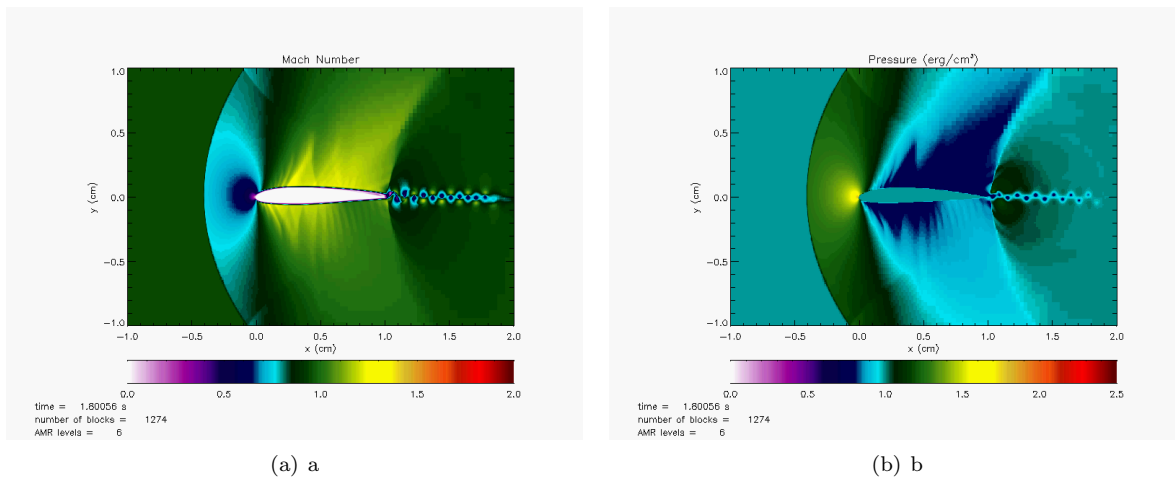


Figure 35.32: NACA2412 in Mach number 0.95 flow at 0 degree angle of attack problem at  $t = 1.8$  (a) Mach number (b) Pressure

### 35.1.12.2 Solid Objects in Sedov Explosion

Another problem for testing a stationary rigid body in a simulation is to consider the Sedov-like explosion in a chamber surrounded by a solid wall with holes. The wall is shown as red blocks with white boundary in [35.34\(a\)](#) and [35.34\(b\)](#). The simulation was done on a uniform Cartesian grid with 300 cells on each direction. Three holes in the wall subdivide it into four different stationary solid bodies in a square computational domain  $[0, 1.5] \times [0, 1.5]$ . The explosion goes off at the origin and generate shock waves inside the chamber. In later time, when the shock waves pass through the three holes in the wall, turbulence effects are triggered from the interaction between the fluid and the wall and enhance vortical fluid motions.

One important thing in this problem is to keep the given symmetry throughout the simulation. The flow symmetry across the diagonal direction is well preserved in [35.34\(a\)](#) and [35.34\(b\)](#).

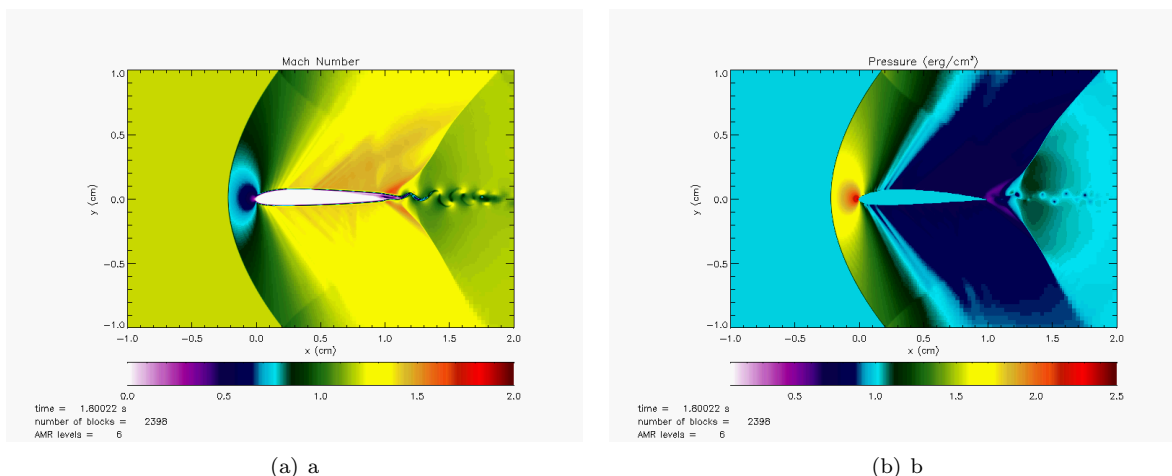


Figure 35.33: NACA2412 in Mach number 1.2 flow at 0 degree angle of attack problem at  $t = 1.8$  (a) Mach number (b) Pressure

## 35.2 Magnetohydrodynamics Test Problems

The magnetohydrodynamics (MHD) test problems provided in this release can be found in `source/-Simulation/SimulationMain/magnetoHD/`. In order to set up an MHD problem, users need to specify the `magnetoHD` path in a setup script. For instance, the `BrioWu` problem can be configured by typing `./setup magnetoHD/BrioWu -auto -1d`.

### 35.2.1 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 35.1.1). 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 35.35, Figure 35.36, Figure 35.37, Figure 35.38, and Figure 35.39. The figures show the distribution of density, normal and tangential velocity components, tangential magnetic field component and pressure at  $t = 0.1$  (in non-dimensional units). As can be seen, the code accurately and sharply resolves all waves present in the solution. There is a small undershoot in the solution at  $x \approx 0.44$ , which results from a discontinuity-enhancing monotized 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 `8Wave` MHD solver with a second-order MUSCL-Hancock scheme (setup with `+8wave`) was used for the results shown in this simulation. The `StaggeredMesh` MHD solver (setup with `+usm`) can also be used for this Brio-Wu problem in one- and two-dimensions. However, in the latter case, the `StaggeredMesh` solver only supports non-rotated setups for which a shock normal is parallel to the  $x$ -axis that initially intersects that axis at  $x = 0.5$  (halfway across a box with unit dimensions). This limitation occurs in the `StaggeredMesh` scheme because the currently released version of the FLASH code does not truly support physically correct boundary conditions for this rotated shock geometry.



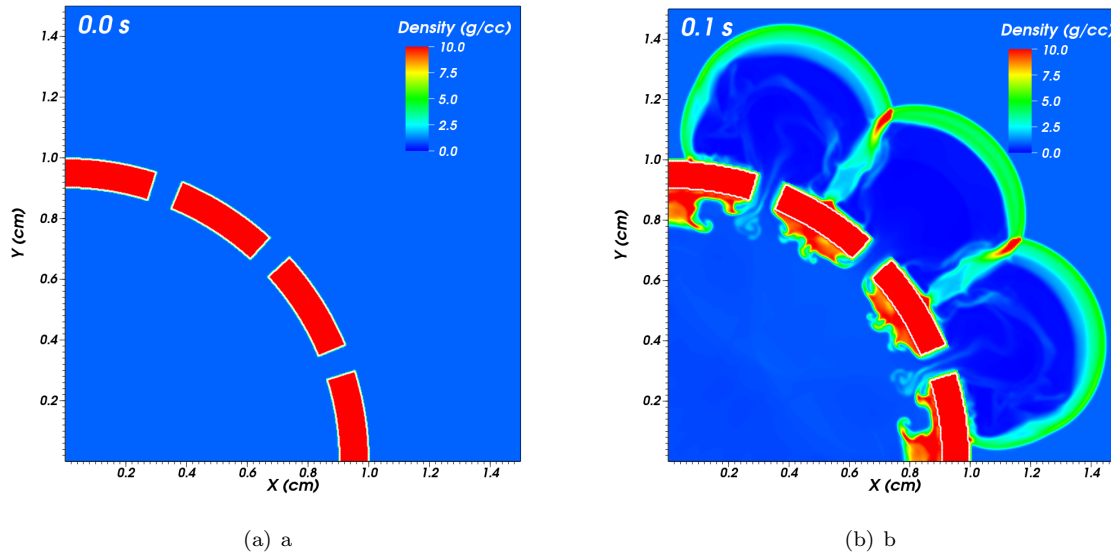


Figure 35.34: The Sedov explosion in a chamber surrounded by a wall with holes. (a) Density plot at  $t = 0.0$  sec. (b) Density plot at  $t = 0.1$  sec.

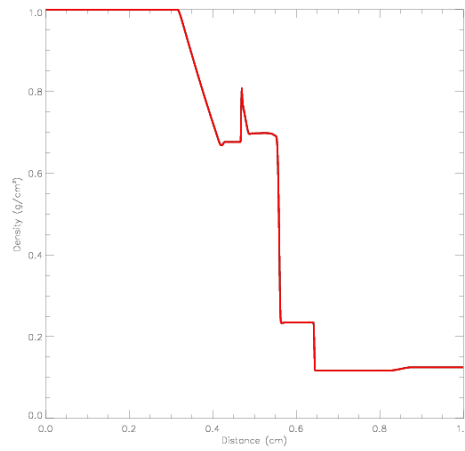


Figure 35.35: Density profile for the Brio-Wu shock tube problem.

### 35.2.1.1 Slowly moving shocks (SMS) issues in the Brio-Wu problem

Figure 35.40 clearly demonstrates that the conventional PPM reconstruction method fails to preserve monotonicities, shedding oscillations especially in the plateau near strong discontinuities such as the contact and right going slow MHD shock. In Fig. 35.41, Mach numbers are plotted with varying strengths of the transverse field  $B_y$ . The oscillations increase with an increase of  $B_y$ ; the reason being that stronger  $B_y$  introduces more transverse effect that resists shock propagation in the  $x$  direction causing the shock to move slowly. This effect is clearly seen in the locations of the shock fronts (right going slow MHD shocks in this case), which remain closer to the initial location  $x = 0.5$  when  $B_y$  is stronger.

Results from using the upwind biased slope limiter for PPM are illustrated in Figures 35.42 – 35.45. The oscillation shedding found in the conventional PPM (e.g., Fig. 35.40) are significantly reduced in both density, and Mach number profiles. The overall qualitative solution behavior of using the upwind PPM

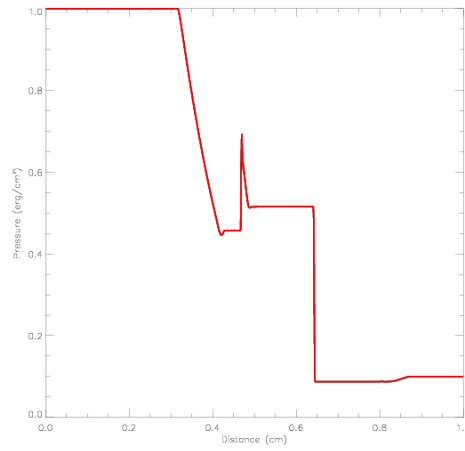


Figure 35.36: Pressure profile for the Brio-Wu shock tube problem.

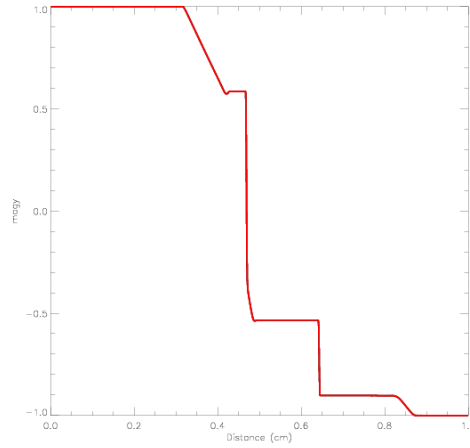


Figure 35.37: Tangential magnetic field profile for the Brio-Wu shock tube problem.

approach, shown in Fig. 35.42, is very much similar to those from the 5th order WENO method as illustrated in Fig. 35.43. As shown in the closeup views in Fig. 35.44 and 35.45, the solutions with the upwind PPM slope limiter (blue curves) outperforms the conventional PPM method (red curves), and compares well with the WENO scheme (green curves). In fact, the upwind approach shows the most flat density plateau of all methods. Note that the SMS issue can be observed regardless of the choice of Riemann solvers and the dissipation mechanism in PPM (e.g., even with flattening on). 400 grid points were used.

### 35.2.2 Orszag-Tang MHD Vortex

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

$$\mathbf{V} = V_0 (-\sin(2\pi y), \sin(2\pi x), 0), \quad \mathbf{B} = B_0 (-\sin(2\pi y), \sin(4\pi x), 0), \quad (x, y) \in [0, 1]^2, \quad (35.17)$$

where  $B_0$  is chosen so that the ratio of the gas pressure to the RMS magnetic pressure is equal to  $2\gamma$ . In this setup the initial density, the speed of sound and  $V_0$  are set to unity; therefore, the initial pressure  $p_0 = 1/\gamma$

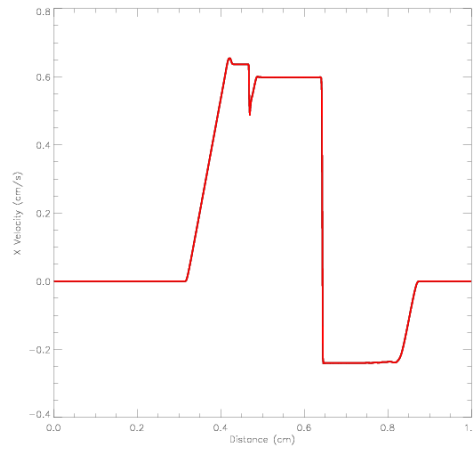


Figure 35.38: Normal velocity profile for the Brio-Wu shock tube problem.

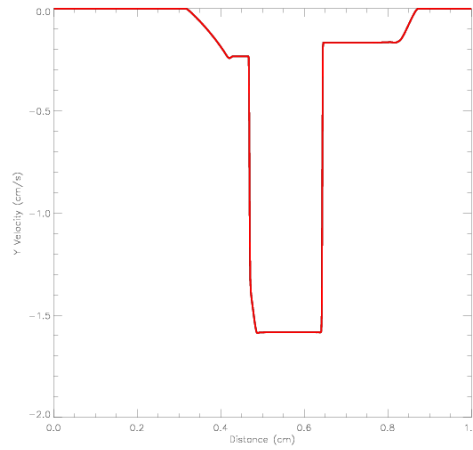


Figure 35.39: Tangential velocity profile for the Brio-Wu shock tube problem.

and  $B_0 = 1/\gamma$ .

As the evolution time increases, the vortex flow pattern becomes increasingly complicated due to the nonlinear interactions of waves. A highly resolved simulation of this problem should produce two-dimensional MHD turbulence. [Figure 35.46](#) and [Figure 35.47](#) shows density and magnetic field contours at  $t = 0.5$ . As one can observe, the flow pattern at this time is already quite complicated. A number of strong waves have formed and passed through each other, creating turbulent flow features at all spatial scales.

The results were obtained using the directionally splitting **8Wave** MHD solver for this Orszag-Tang problem.

The 3D version are also shown in the below solved using the unsplit staggered mesh MHD solver.

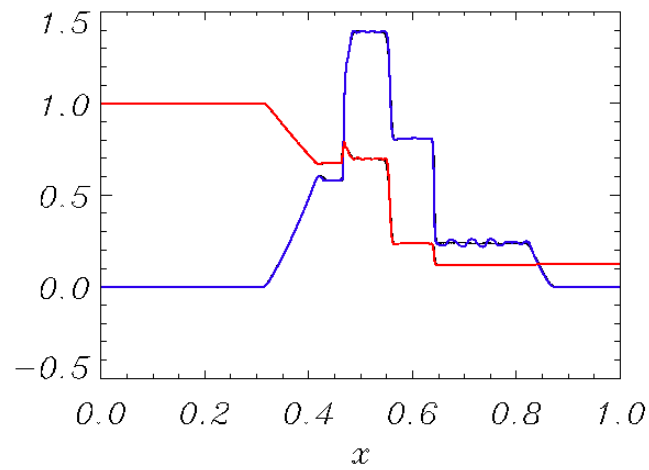


Figure 35.40: Density (thick red) and Mach number (thick blue) at  $t = 0.1$  of the Brio-Wu test with the conventional PPM's MC limiter. Thin black curves represent reference solutions using the PLM of MUSCL-Hancock scheme. Severe numerical oscillations are evident in the solution using the conventional PPM reconstruction on 400 grid points.

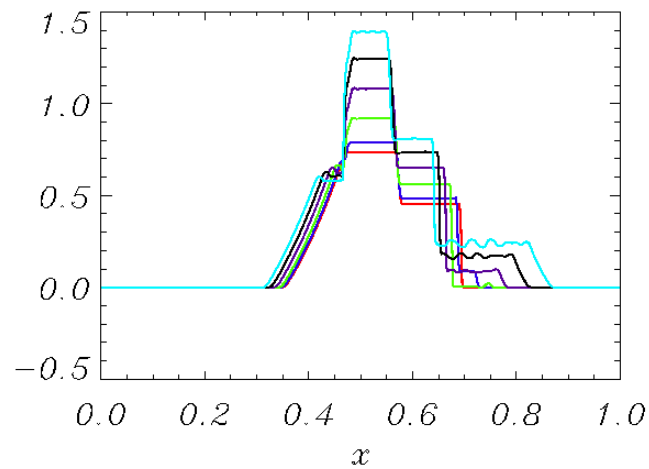


Figure 35.41: Mach numbers at  $t = 0.1$  with varying  $B_y$  from 0 to 1 with the conventional PPM's MC limiter. Curves in red, blue, green, purple, black, and cyan represent  $B_y = 0, 0.2, 0.4, 0.6, 0.8$ , and 1, respectively. Severe numerical oscillations are evident in the solution using the conventional PPM reconstruction on 400 grid points.

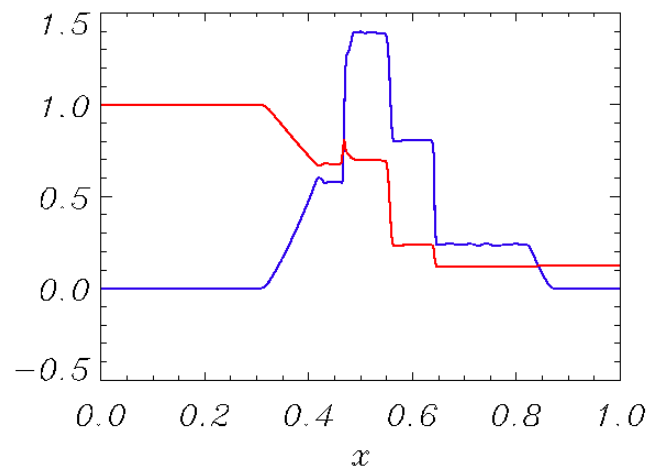


Figure 35.42: Density (red) and Mach number (blue) at  $t = 0.1$  of the Brio-Wu test using the upwind biased PPM limiter.

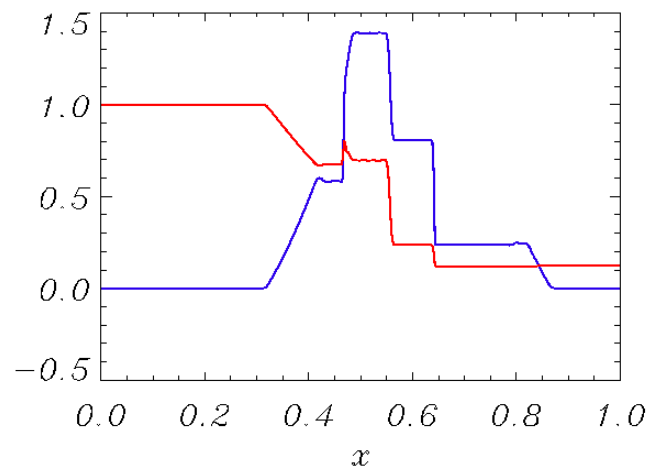


Figure 35.43: Density (red) and Mach number (blue) at  $t = 0.1$  of the Brio-Wu test using the 5th order WENO scheme.

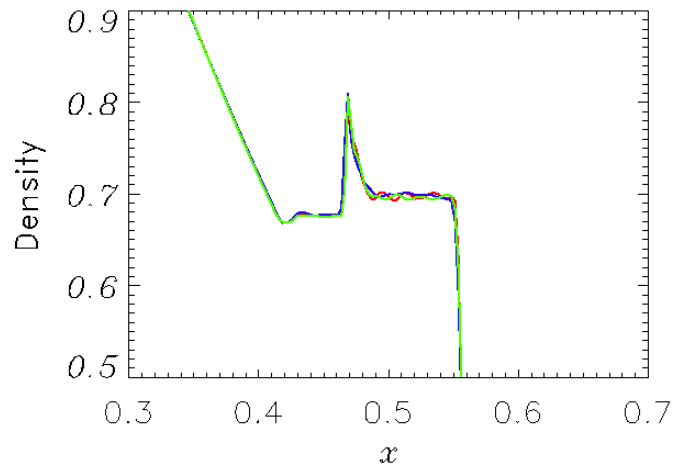


Figure 35.44: Density closeup at  $t = 0.1$  of the Brio-Wu test. The conventional PPM in red curve, the upwind PPM in blue curve, and the WENO scheme in green curve.

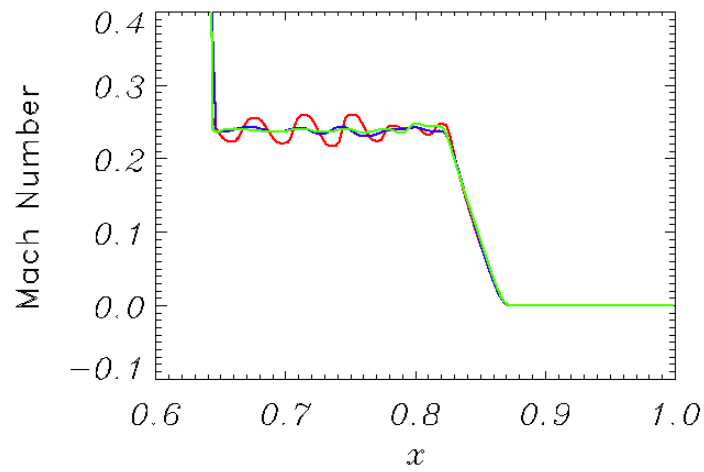


Figure 35.45: Mach number closeup at  $t = 0.1$  of the Brio-Wu test. The conventional PPM in red curve, the upwind PPM in blue curve, and the WENO scheme in green curve.

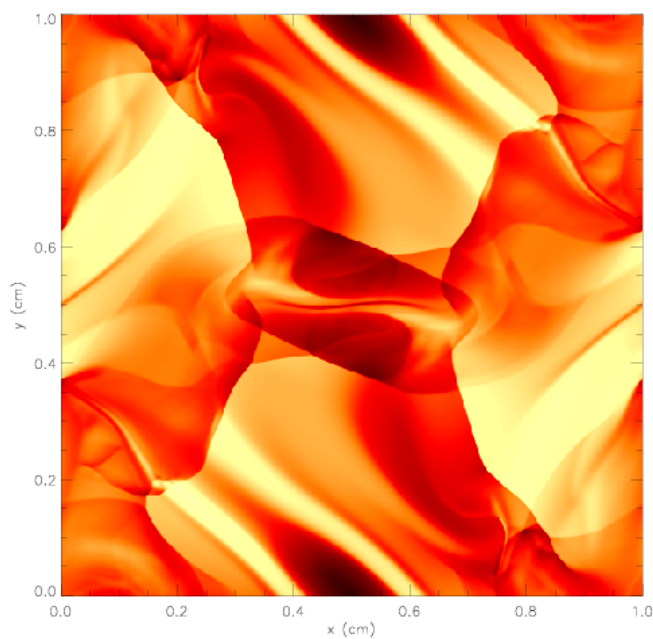


Figure 35.46: Density contours in the Oszag-Tang MHD vortex problem at  $t = 0.5$ .

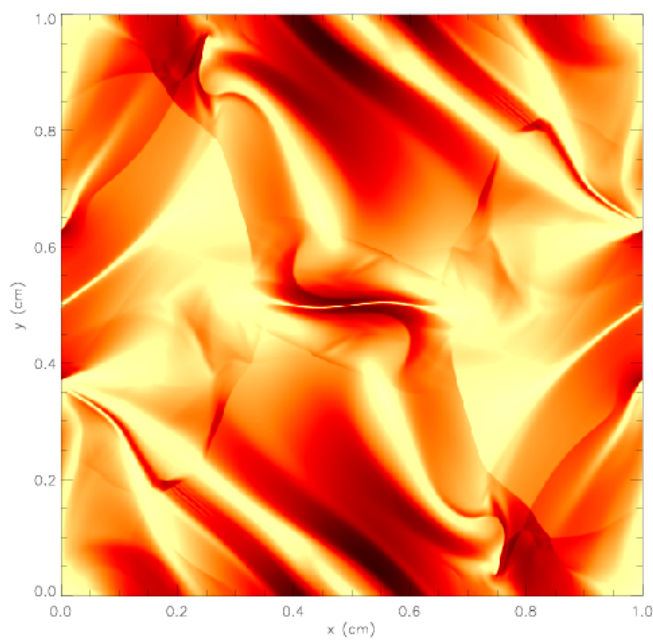


Figure 35.47: Magnetic field contours in the Oszag-Tang MHD vortex problem at  $t = 0.5$ .

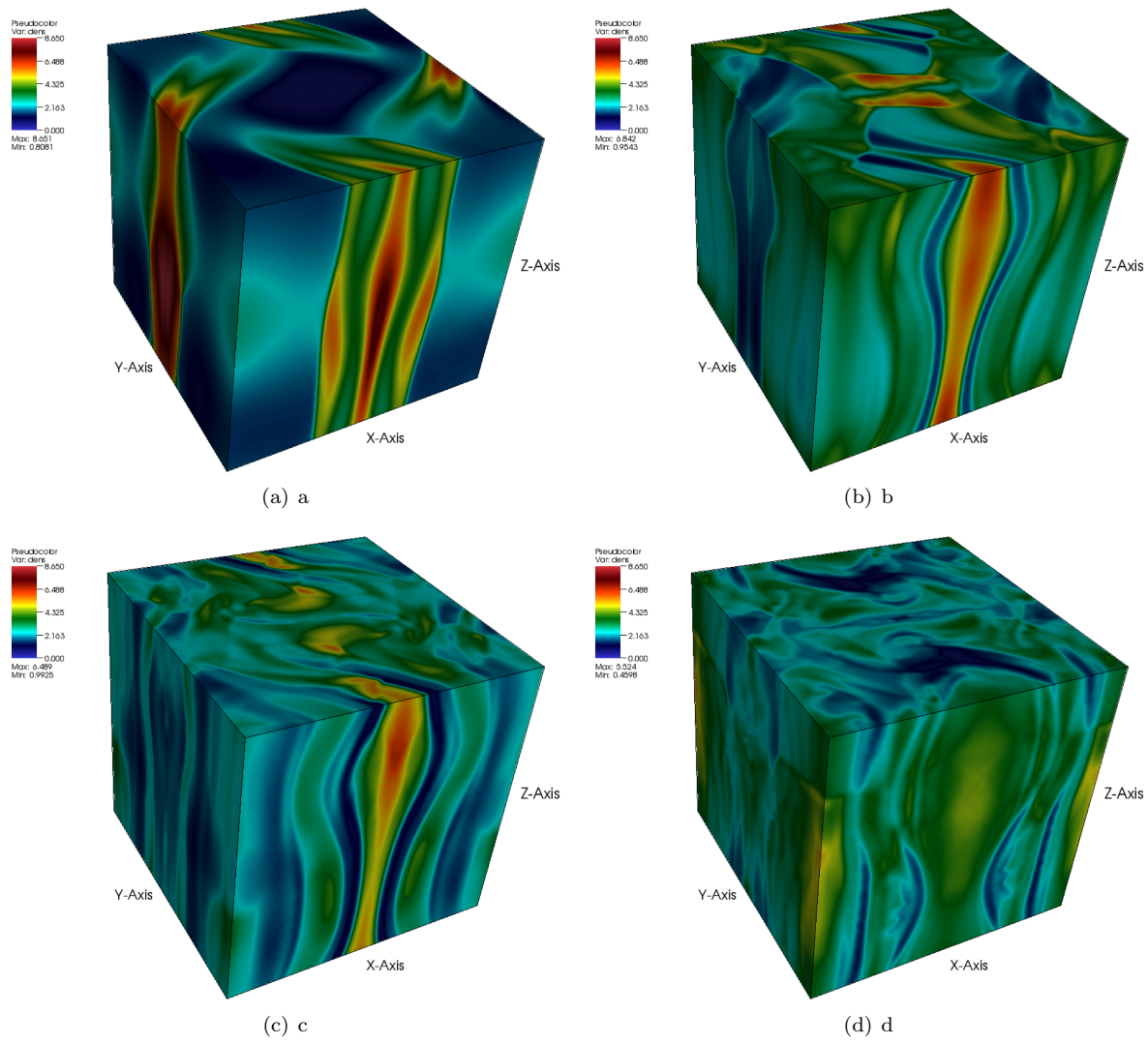


Figure 35.48: Density plots of a 3D version of the Orszag-Tang problem on a  $128^3$  uniform grid resolution. (a) Density at  $t = 0.2$  (b) Density at  $t = 0.5$  (c) Density at  $t = 0.7$  (d) Density at  $t = 1.0$ .



### 35.2.3 Magnetized Accretion Torus

The magnetized accretion torus problem is based on the global magneto-rotational instability (MRI) simulations of Hawley (2000). It can be found under `magnetoHD/Torus`. We consider a magnetized torus of constant angular momentum ( $\Omega \propto r^{-2}$ ), inside a normalized Paczyński-Wiita pseudo-Newtonian gravitational potential (Paczyński&Wiita, 1980) of the form  $\Phi = -1/(R - 1)$ , where  $R = \sqrt{r^2 + z^2}$ .

The cylindrical computational domain, with lengths normalized at  $r_0$ , extends from  $1.5 r_0$  to  $15.5 r_0$  in the radial direction and from  $-7 r_0$  to  $7 r_0$  in the  $z$  direction. For this specific simulation we use seven levels of refinement, linear reconstruction with vanLeer limiter and the HLLC Riemann solver. The boundary conditions are set to outflow, except for the leftmost side where a diode-like condition is applied.

Assuming an adiabatic equation of state, the initial density profile of the torus is given by

$$\frac{\Gamma P}{(\Gamma - 1)\rho} = C - \Phi - \frac{l_K^2}{2r^2}, \quad (35.18)$$

where the specific heats ratio is  $\Gamma = 5/3$ ,  $l_K$  is the Keplerian angular momentum at the pressure maximum ( $r_{Pmax} = 4.7r_0$ ) and  $C$  is an integration constant that specifies the outer surface of the torus, given its inner radius ( $r_{in} = 3r_0$ ). The initial poloidal magnetic field is computed using the  $\phi$  component of the vector potential,  $A_\phi \propto \max(\rho - 5, 0)$ , and normalized so as the initial minimum value of the plasma  $\beta = 2P/B^2$  is equal to  $10^2$ . The resulting field follows the contours of density, i.e. torus-like nested loops, and is embedded well within the torus.

We allow the system to evolve for  $t = 150 t_0$ . The torus is MRI unstable and after approximately one revolution accretion sets in. The strong shear generates an azimuthal field component and the angular momentum is redistributed. Due to the instability, filamentary structures form at the torus surface that account for its rich morphology (Figure 35.49). These results can be promptly compared to those in Hawley (2000) and Mignone et al. (2007).

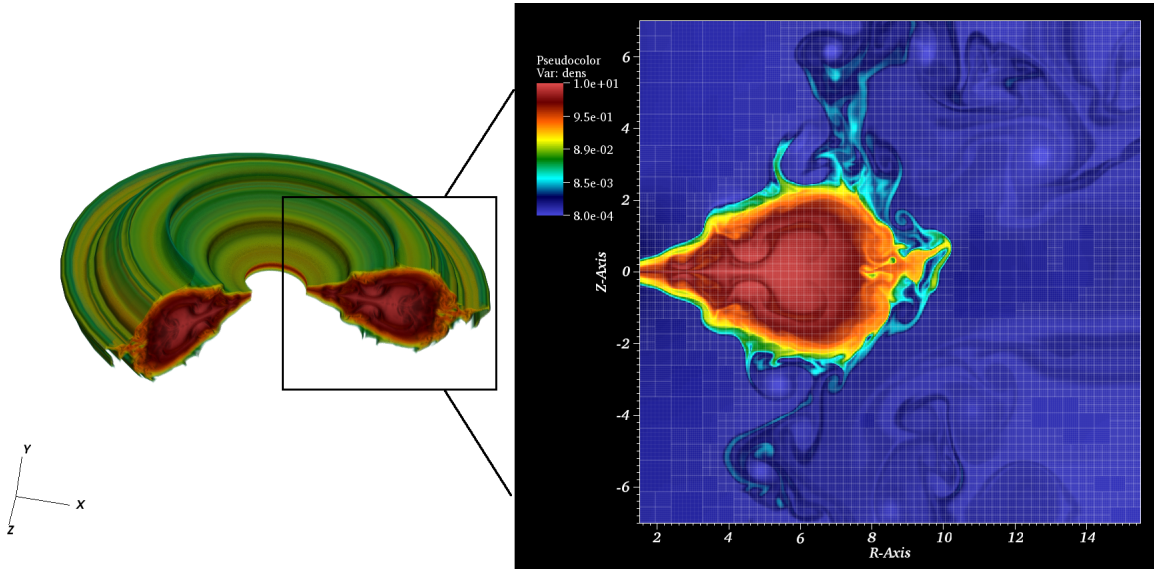


Figure 35.49: Left: 3D rendering of the axisymmetric torus evolution. Right: Density logarithm of the magnetized accretion torus after  $150 t_0$ . Superposed are the AMR levels and the mesh.

### 35.2.4 Magnetized Noh Z-pinch

In this test we consider the magnetized version of the classical Noh problem (Noh, 1987). It consists of a cylindrically symmetric implosion of a pressure-less gas: the gas stagnates at the symmetry axis and creates an outward moving accretion shock which propagates at a constant velocity. The self-similar analytic solution of this problem has been widely used as benchmark test for hydrodynamic codes, especially those targeting implosion.

Recently, Velikovich et al. (2012) have extended the original test problem to include an embedded azimuthal magnetic field, in accordance with the Z-pinch physics. In their study, they present a family of exact analytic solutions for which the values of primitive variables are finite everywhere, providing an excellent benchmark for Z-pinch and general MHD codes.

We perform this test in both cartesian and cylindrical geometries. The tests can be found respectively in `magnetoHD/Noh` and `magnetoHD/NohCylindrical`. For brevity we describe only the cylindrical initialization. The cartesian can be easily recovered by projecting the solution onto the X-Y plane. A 3T MHD version of this test can also be found under `magnetoHD/unitTest/NohCylindricalRagelike`.

The simulation is initialized in a computational box that spans  $[0, 3]$  cm in the  $r$  and  $z$  directions, in cylindrical ( $r - z$ ) geometry. The leftmost boundary condition is set to axisymmetry, whereas the remaining boundaries are set to outflow (zero gradient). The initial condition in primitive variables is defined as

$$\begin{aligned}\rho &= 3.1831 \times 10^{-5} r^2 \text{ g/cm}^3, \\ \mathbf{v} &= (-3.24101 \times 10^7, 0, 0) \text{ cm/s}, \\ \mathbf{B} &= (0, 6.35584 \times 10^5 r, 0) \text{ gauss}, \\ P &= C \mathbf{B}^2.\end{aligned}\tag{35.19}$$

Since Godunov-type codes cannot run with zero pressure, we initialize  $P$  by choosing  $C = P/\mathbf{B}^2 = 10^{-6}$ , ensuring a magnetically dominated plasma.

The simulations are evolved for 30 ns, utilizing the unsplit solver and a Courant number of 0.8. We use 6 levels of refinement, corresponding to an equivalent resolution of  $256 \times 256$  zones. The reconstruction is piecewise-linear (second order) with characteristic limiting, whereas the Riemann solvers employed are the HLLD and Roe.

The resulting density profile is shown in [Figure 35.50](#). The refinement closely follows the propagation of the discontinuity which reaches the analytically predicted location ( $r = 0.3$  cm) at  $t = 30$  ns. The lineouts for HLLD (green dots) and Roe (blue dots) shown on the right, display good agreement with the analytic solution (red line) and the discontinuity is sharply captured (resolved on two points). Our [Figure 35.50](#) can be directly compared to Figures 2a and 3a of Velikovich et al. (2012).

### 35.2.5 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

$$\rho(x, y) = \begin{cases} 10 & r \leq r_0 \\ 1 + 9f(r) & r_0 < r < r_1 \\ 1 & r \geq r_1 \end{cases}\tag{35.20}$$

$$u(x, y) = \begin{cases} -f(r)u_0(y - 0.5)/r_0 & r \leq r_0 \\ -f(r)u_0(y - 0.5)/r & r_0 < r < r_1 \\ 0 & r \geq r_1 \end{cases}\tag{35.21}$$

$$v(x, y) = \begin{cases} f(r)u_0(x - 0.5)/r_0 & r \leq r_0 \\ f(r)u_0(x - 0.5)/r & r_0 < r < r_1 \\ 0 & r \geq r_1 \end{cases}\tag{35.22}$$

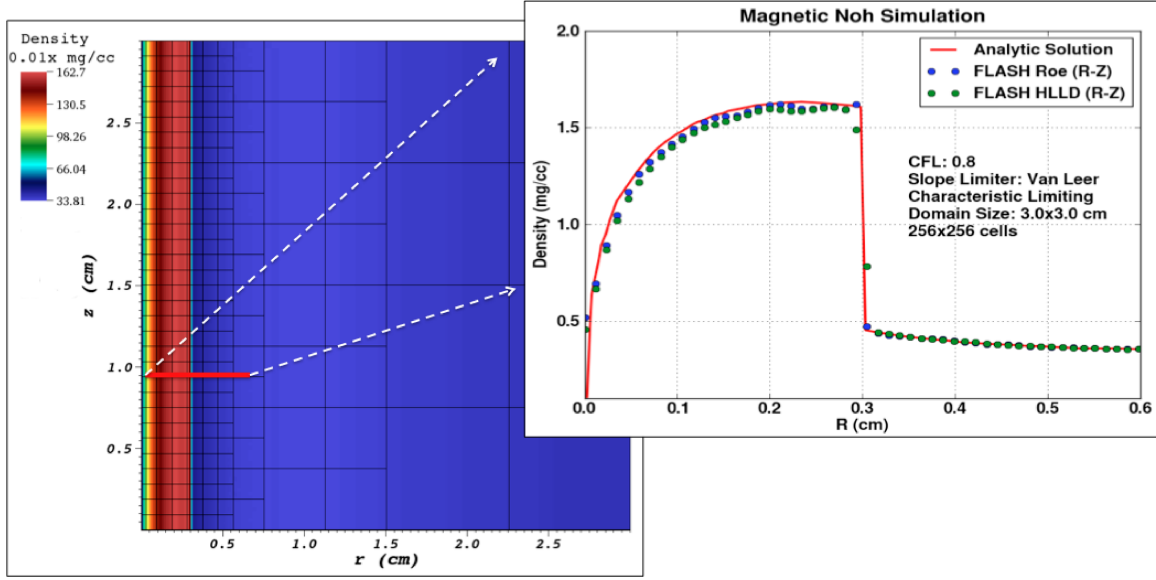


Figure 35.50: Magnetized Noh problem in cylindrical coordinates. Left: density snapshot along with AMR levels after 30 ns for the HLLD Riemann solver. Right: Lineouts of density close to the origin,  $r = [0, 0.6]$  cm, for HLLD and Roe, superposed on the analytic solution.

$$p(x, y) = 1 \quad (35.23)$$

$$B_x(x, y) = \frac{5}{\sqrt{4\pi}} \quad (35.24)$$

$$B_y(x, y) = 0, \quad (35.25)$$

where  $r_0 = 0.1, r_1 = 0.115, r = \sqrt{(x - 0.5)^2 + (y - 0.5)^2}, w = B_z = 0$  and a taper function  $f(r) = (r_1 - r)/(r - r_0)$ . The value  $\gamma = 1.4$  is used. The initial set-up is occupied by a dense rotating disk at the center of the domain, surrounded by the ambient flow at rest with uniform density and pressure. The rapidly spinning rotor is not in an equilibrium state due to the centrifugal forces. As the rotor spins with the given initial rotating velocity, the initially uniform magnetic field in  $x$ -direction will wind up the rotor. The rotor will be wrapped around by the magnetic field, and hence start launching torsional Alfvén waves into the ambient fluid. The angular momentum of the rotor will be diminished in later times as the evolution time increases. The circular rotor will be progressively compressed into an oval shape by the build-up of the magnetic pressure around the rotor. The results shown in Figure 35.51 were obtained using the **StaggeredMesh** MHD solver using 6 refinement levels. The divergence free evolution of the magnetic fields are well preserved as illustrated in Figure 35.52.

### 35.2.6 MHD Current Sheet

The two-dimensional current sheet problem, **magnetoHD/CurrentSheet**, has recently been studied by Gardiner and Stone (2005) in ideal MHD regime. The two current sheets are initialized and therefore magnetic reconnections are inevitably driven. In the regions the magnetic reconnection takes place the magnetic flux approaches vanishingly small values, and the loss in the magnetic energy is converted into heat (thermal energy). This phenomenon changes the overall topology of the magnetic fields and hence affects the global magnetic configuration.

The square computational domain is given as  $[0, 2] \times [0, 2]$  with periodic boundary conditions on all four

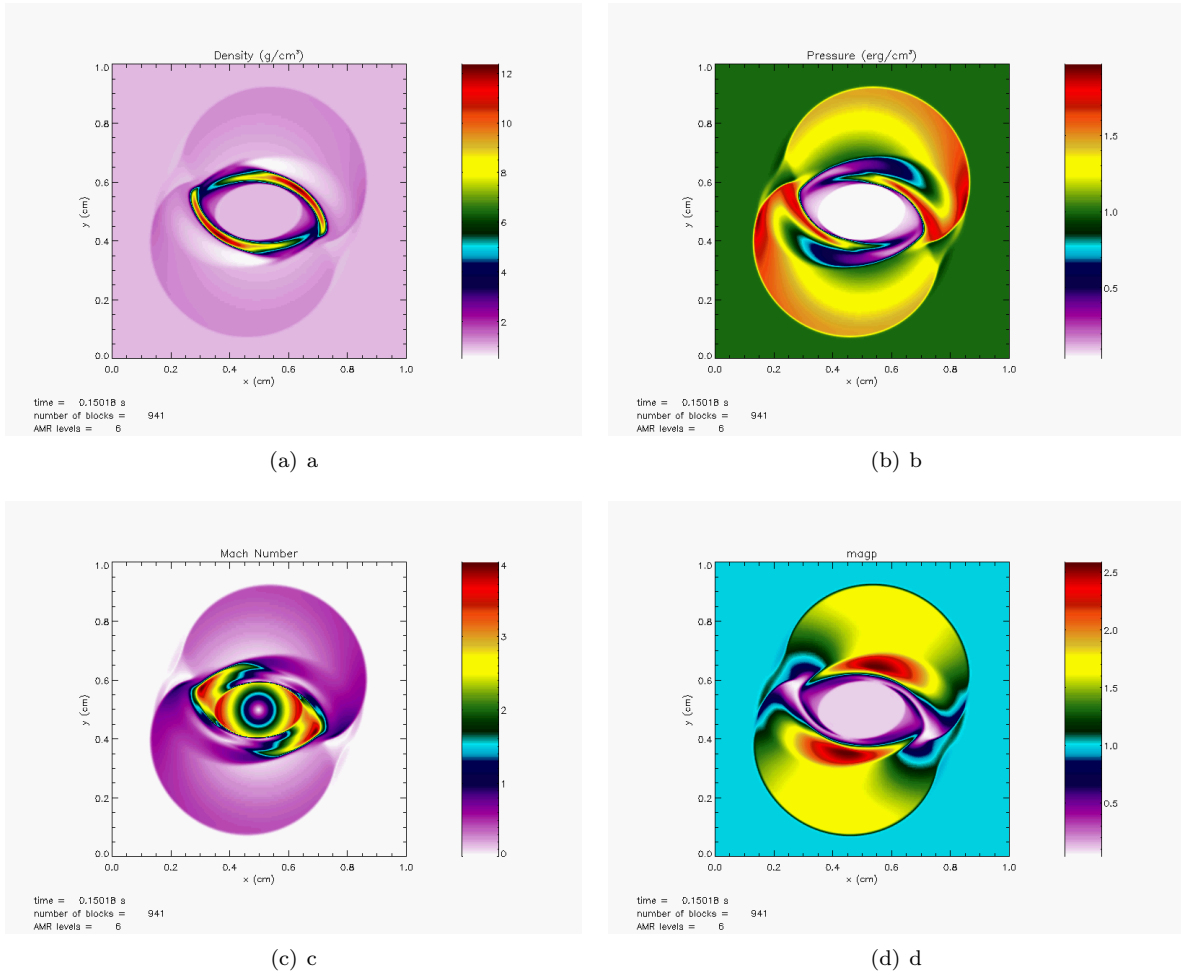


Figure 35.51: The Rotor problem at  $t = 0.15$  (a) Density (b) Pressure (c) Mach number (d) Magnetic pressure.

sides. We initialize two current sheets in the following:

$$B_y = \begin{cases} B_0 & 0.0 \leq x < 0.5 \\ -B_0 & 0.5 \leq x < 1.5 \\ B_0 & 1.5 \leq x \leq 2.0 \end{cases}, \quad (35.26)$$

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 35.53(a) – 35.53(f) on a  $256 \times 256$  uniform grid. In Figure 35.54 the same problem is resolved on an AMR grid with 6 refinement levels, showing the current density  $j_z$  along with the AMR block structures at  $t = 4.0$ . The **StaggeredMesh** solver was used for this problem.

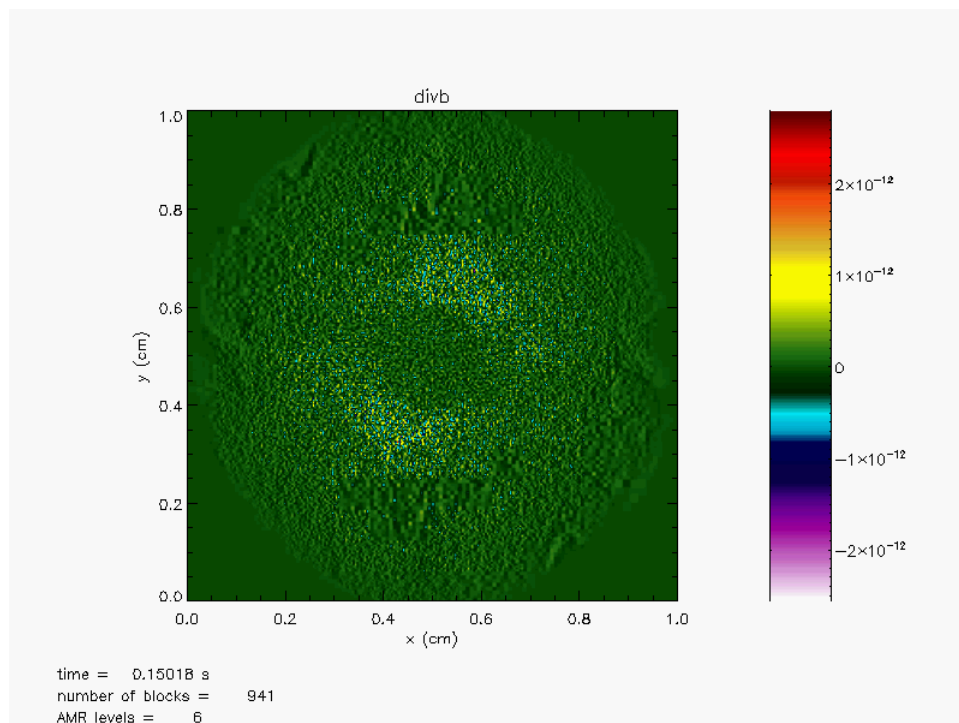


Figure 35.52: Divergence of magnetic fields using the **StaggeredMesh** solver at  $t = 0.15$  for the Rotor problem.

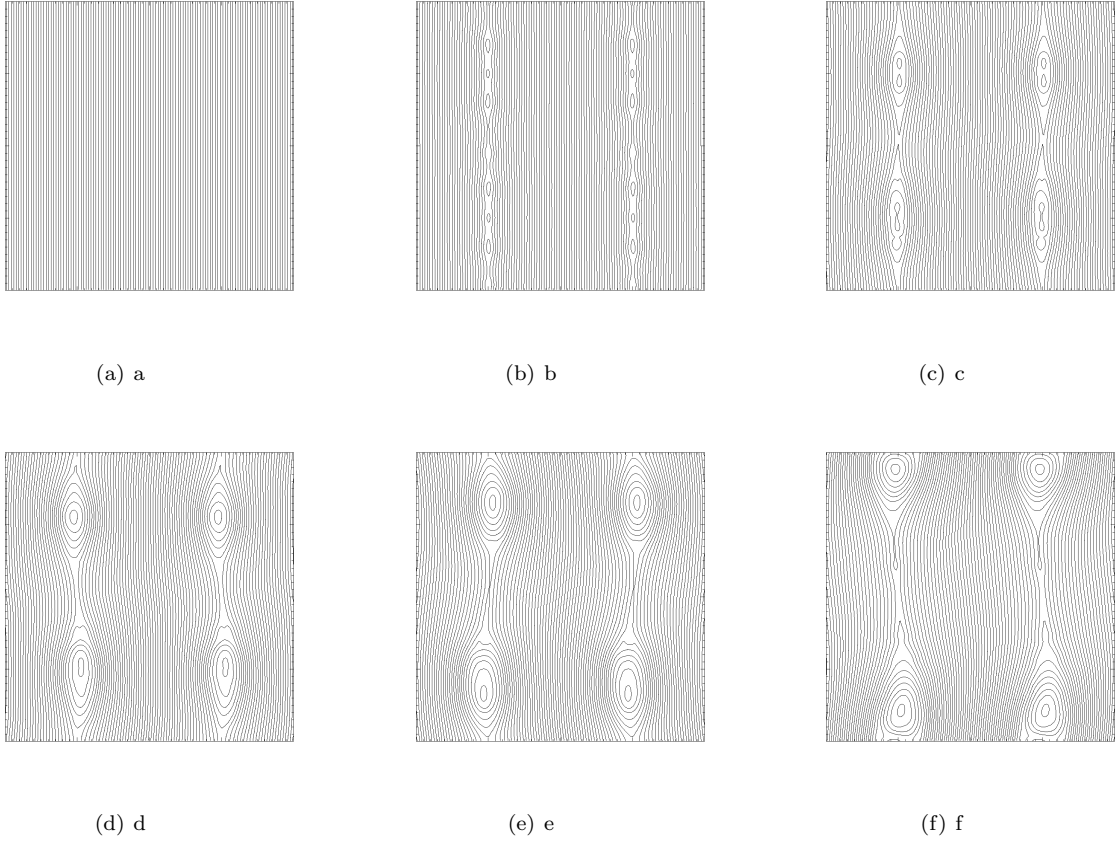


Figure 35.53: The temporal evolutions of field lines for the MHD **CurrentSheet** problem. Equally spaced 60 contour lines are shown at time (a)  $t = 0.0$  (b)  $t = 1.0$  (c)  $t = 2.0$  (d)  $t = 3.0$  (e)  $t = 4.0$  (f)  $t = 5.0$ .

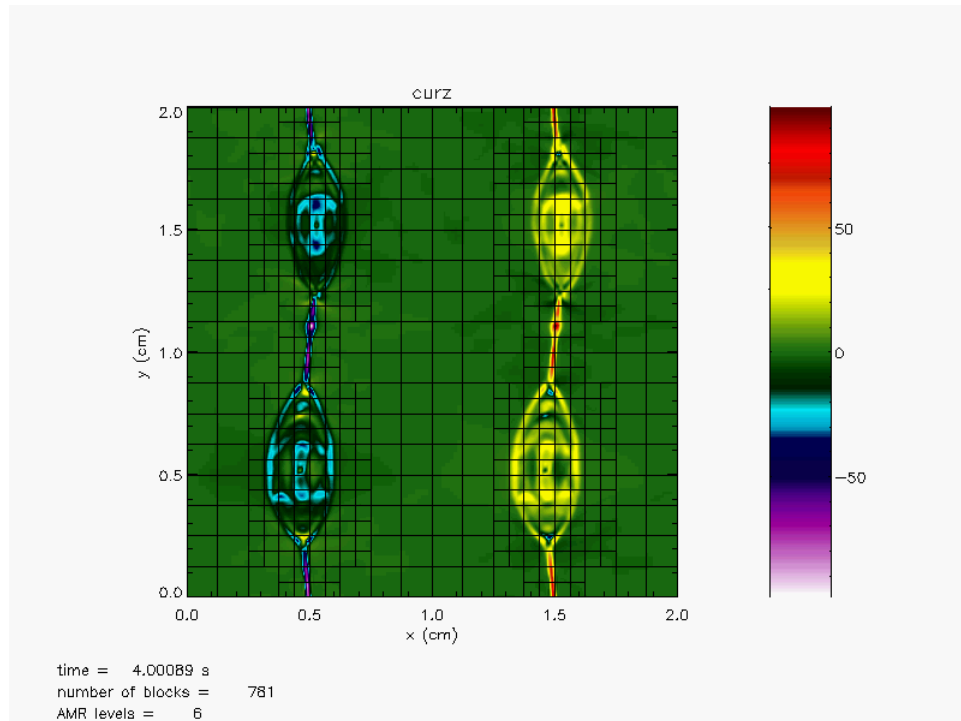


Figure 35.54: Current density at  $t = 4.0$  using the `StaggeredMesh` solver for the MHD `CurrentSheet` problem.

### 35.2.7 Field Loop

The 2D and 3D field loop advection problems (`magnetoHD/FieldLoop`) are known to be stringent test cases in multidimensional MHD. In this test problem we consider a 2D advection of a weakly magnetized field loop traversing the computational domain diagonally. Details of the problem has been described in Gardiner and Stone (2005).

The computational domain is  $[-1, 1] \times [-0.5, 0.5]$ , with a grid resolution  $256 \times 148$ , and doubly-periodic boundary conditions. With this rectangular grid cell, the flow is not symmetric in  $x$  and  $y$  directions because the field loop does not advect across each *grid cell* diagonally and hence the resulting fluxes are different in  $x$  and  $y$  directions. The density and pressure are unity everywhere and  $\gamma = 5/3$ . The velocity fields are defined as,

$$\mathbf{U} = u_0(\cos\theta, \sin\theta, 1) \quad (35.27)$$

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  $A_z$ ,

$$B_x = \frac{\partial A_z}{\partial y}, \quad B_y = -\frac{\partial A_z}{\partial x}, \quad (35.28)$$

where

$$A_z = \begin{cases} A_0(R - r) & r \leq R \\ 0 & \text{otherwise.} \end{cases}, \quad (35.29)$$

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 (35.29) are  $A_0 = 10^{-3}$  and a field loop radius  $R = 0.3$ . This initial condition results in a very high plasma beta  $\beta = p/B_p = 2 \times 10^6$  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 35.55, the USM scheme maintains the circular shape of the loop extremely well to the final time step. The scheme successfully retains the initial circular symmetry and does not develop severe oscillations.

A variant 3D version of this problem (Gardiner and Stone, 2008) is also available, and is illustrated in in Fig. 35.56. This problem is considered to be a particularly challenging test because the correct solution requires inclusion of the multidimensional MHD terms to preserve the in-plane dynamics. Otherwise, the failure in preserving the in-plane dynamics (i.e., growth in the out-of-plane component) results in erroneous behavior of the field loop. As shown in Fig. 35.56, the USM solver successfully maintains the circular shape of the field loop and maintains the out-of-plane component of the magnetic field to very small values over the domain. This figure compares very well with Fig. 2 in Gardiner and Stone, 2008.

### 35.2.8 3D MHD Blast

A 2D version of the MHD blast problem was studied by Zachary *et al.* (Zachary, Malagoli, and Colella, 1994) and we consider a variant 3D version of the MHD spherical blast wave problem here. This problem



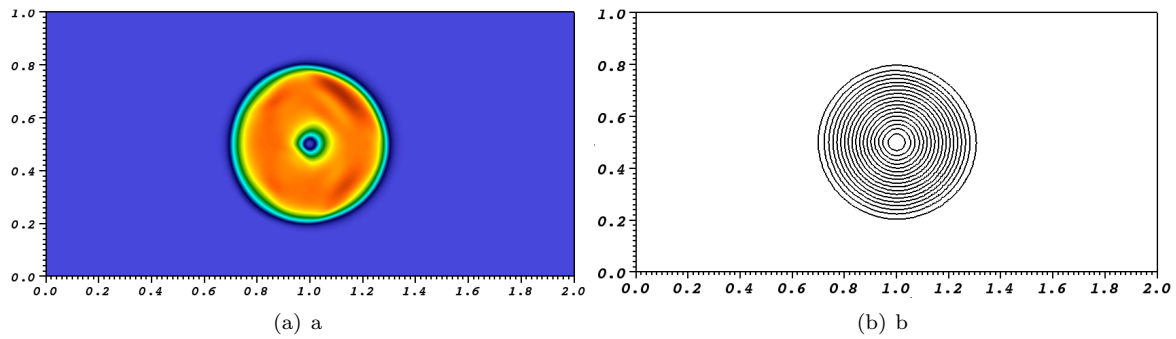


Figure 35.55: The field loop advection problem using the `StaggeredMesh` solver at time  $t = 2$  with the Roe Riemann solver. (a)  $B_p$  with the MEC at  $t = 2$ . The color scheme between  $2.32 \times 10^{-25}$  and  $7.16 \times 10^{-7}$  was used. (b) Magnetic field lines with the MEC at  $t = 2$ . 20 contour lines of  $A_z$  between  $-2.16 \times 10^{-6}$  and  $2.7 \times 10^{-4}$  are shown.

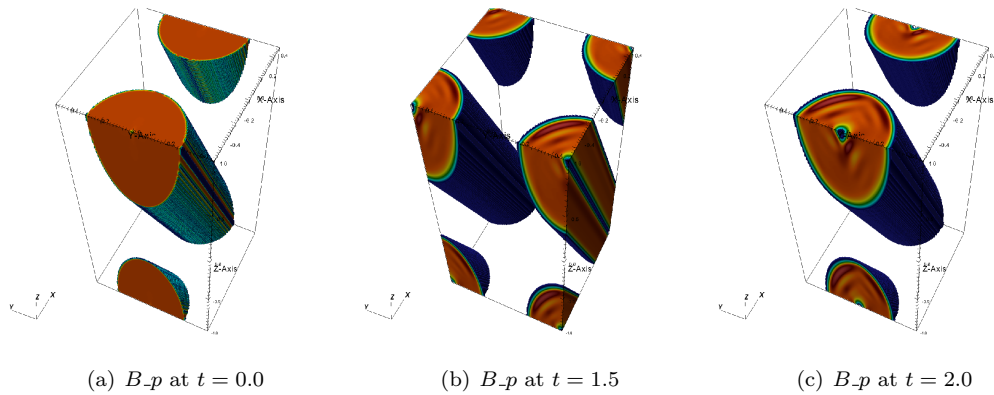


Figure 35.56: Thresholded images of the field loop advection problem at times  $t = 0.0, 1.5$ , and  $2.0$  using a uniform grid size  $128 \times 128 \times 256$ .

leads to the formation and propagation of strong MHD discontinuities, relevant to astrophysical phenomena where the magnetic field energy has strong dynamical effects. With a numerical scheme that fails to preserve the divergence-free constraint, unphysical states could be obtained involving negative gas pressure because the background magnetic pressure increases the strength of magnetic monopoles.

This problem can be computed in various magnetized flow regimes by considering different magnetic field strengths. The computational domain is a square  $[-0.5, 0.5] \times [-0.5, 0.5] \times [-0.5, 0.5]$  with a maximum refinement level 4. The explosion is driven by an over-pressurized circular region at the center of the domain with a radius  $r = 0.1$ . The initial density is unity everywhere, and the pressure of the ambient gas is 0.1, whereas the pressure of the inner region is 1000. The strength of a uniform magnetic field in the  $x$ -direction are 0,  $50/\sqrt{4\pi}$  and  $100/\sqrt{4\pi}$ . This initial condition results in a very low- $\beta$  ambient plasma state,  $\beta = 2.513 \times 10^{-4}$ . Through this low- $\beta$  ambient state, the explosion emits almost spherical fast magneto-sonic shocks that propagate with the fastest wave speed. The flow has  $\gamma = 1.4$ .

With this strong magnetic field strength,  $B_x = 100/\sqrt{4\pi}$ , shown in Figure 35.57, the explosion now becomes highly anisotropic as shown in the pressure plot in Figure 35.57. The Figure shows that the displacement of gas in the transverse  $y$ -direction is increasingly inhibited and hydrodynamical shocks propagate in both positive and negative  $x$ -directions parallel to  $B_x$ . This process continues until total pressure equilibrium is reached in the central region. This problem is also available in 2D setup.

## 35.3 Gravity Test Problems

### 35.3.1 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

$$\begin{aligned}\rho(\mathbf{x}) &= \rho_0 [1 + \delta \cos(\mathbf{k} \cdot \mathbf{x})] \\ p(\mathbf{x}) &= p_0 [1 + \gamma \delta \cos(\mathbf{k} \cdot \mathbf{x})] \\ \mathbf{v}(\mathbf{x}) &= 0 ,\end{aligned}\tag{35.30}$$

where the perturbation amplitude  $\delta \ll 1$ . The stability of the perturbation is determined by the relationship between the wavenumber  $k \equiv |\mathbf{k}|$  and the Jeans wavenumber  $k_J$ , where  $k_J$  is given by

$$k_J \equiv \frac{\sqrt{4\pi G \rho_0}}{c_0} ,\tag{35.31}$$

and where  $c_0$  is the unperturbed adiabatic sound speed

$$c_0 = \sqrt{\frac{\gamma p_0}{\rho_0}}\tag{35.32}$$

(Chandrasekhar 1961). If  $k > k_J$ , the perturbation is stable and oscillates with frequency

$$\omega = \sqrt{c_0^2 k^2 - 4\pi G \rho_0} ;\tag{35.33}$$

otherwise, it grows exponentially, with a characteristic timescale given by  $\tau = (i\omega)^{-1}$ .

We checked the dispersion relation (35.33) for stable perturbations with  $\gamma = 5/3$  by fixing  $\rho_0$  and  $p_0$  and performing several runs with different  $k$ . We followed each case for roughly five oscillation periods using a uniform grid in the box  $[0, L]^2$ . We used  $\rho_0 = 1.5 \times 10^7$  g cm $^{-3}$  and  $p_0 = 1.5 \times 10^7$  dyn cm $^{-2}$ , yielding

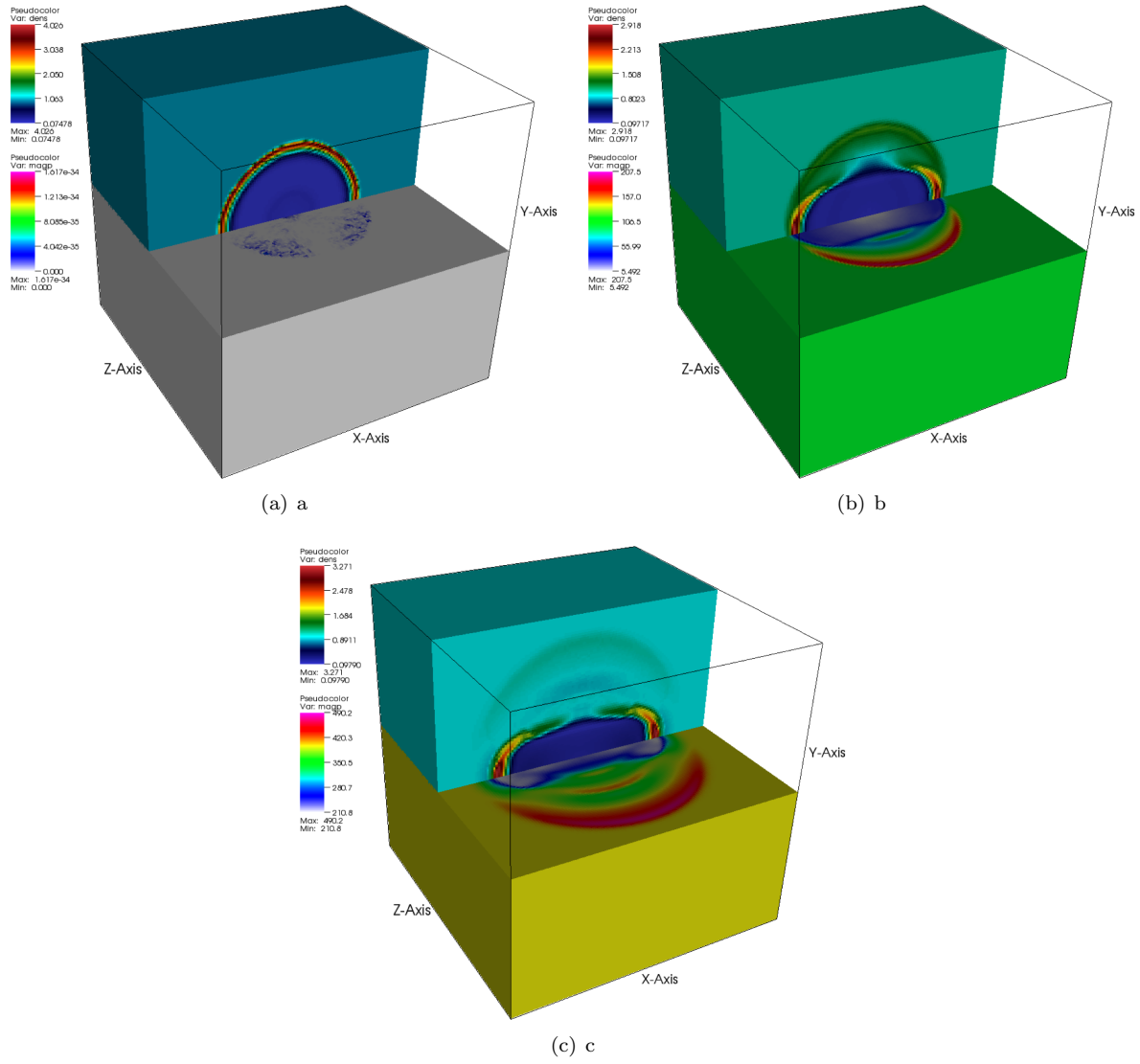


Figure 35.57: The MHD blast test at time  $t = 0.01$  using the unsplit staggered mesh MHD solver. Density (top half) and magnetic pressure (bottom half) plots for three different strengths of  $B_x$ . (a)  $B_x = 0$  (b)  $B_x = 50/\sqrt{4\pi}$  (c)  $B_x = 100/\sqrt{4\pi}$ .

$k_J = 2.747 \text{ cm}^{-1}$ . 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

$$L = \frac{1}{2} \sqrt{\frac{\pi \gamma p_0}{G \rho_0^2}}. \quad (35.34)$$

This prevents roundoff errors at wavenumbers less than  $k_J$  from being amplified by the physical Jeans instability. We used wavevectors  $\mathbf{k}$  parallel to and at 45 degrees to the  $x$ -axis. Each test calculation used the multigrid Poisson solver together with its default settings.

The resulting kinetic, thermal, and potential energies as functions of time for one choice of  $\mathbf{k}$  are shown

in Figure 35.58 together with the analytic solution, which is given in two dimensions by

$$\begin{aligned}
 T(t) &= \frac{\rho_0 \delta^2 |\omega|^2 L^2}{8k^2} [1 - \cos(2\omega t)] \\
 U(t) - U(0) &= -\frac{1}{8} \rho_0 c_0^2 \delta^2 L^2 [1 - \cos(2\omega t)] \\
 W(t) &= -\frac{\pi G \rho_0^2 \delta^2 L^2}{2k^2} [1 + \cos(2\omega t)] .
 \end{aligned} \tag{35.35}$$

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 35.59 compares the resulting dispersion relation to (35.33). It can be seen from this plot that FLASH correctly reproduces (35.33). At the highest wave number ( $k = 100$ ), each wavelength is resolved using only about 14 cells on a six-level uniform grid, and the average timestep (which depends on  $c_0$ ,  $\Delta x$ , 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%.

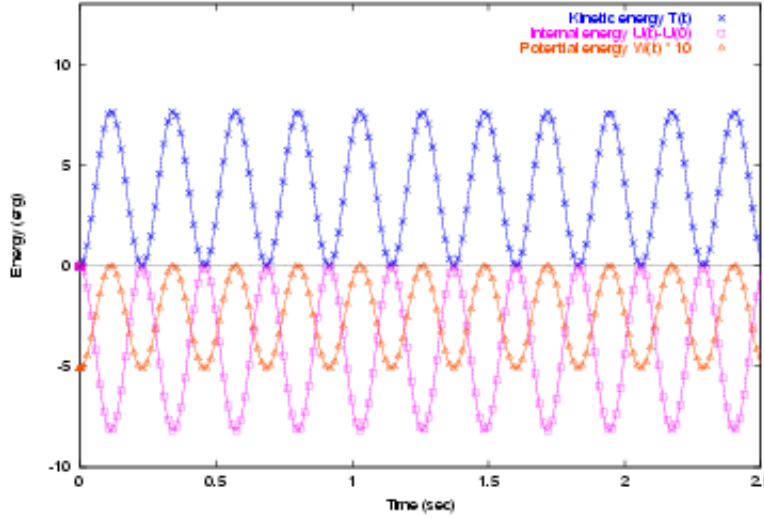


Figure 35.58: Kinetic, internal, and potential energy versus time for a stable Jeans mode with  $k = 10.984$ . Points indicate numerical values found using FLASH 3.0 with a fixed four-level adaptive grid. The analytic solution for each form of energy is shown using a solid line.

The additional runtime parameters supplied with the **Jeans** problem are listed in Table 35.10. This problem is configured to use the perfect-gas equation of state (**gamma**) with **gamma** set to 1.67 and is run in a two-dimensional unit box. The refinement marking routine (**Grid\_markRefineDerefine.F90**) supplied with this problem refines blocks whose mean density exceeds a given threshold. Since the problem is not spherically symmetric, the multigrid Poisson solver should be used.

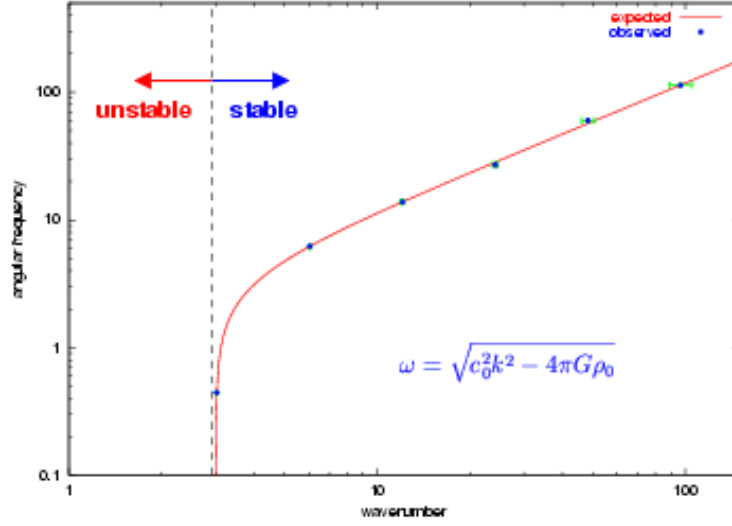


Figure 35.59: Computed versus expected Jeans dispersion relation (for stable modes) found using FLASH 1.62 with a six-level uniform grid.

Table 35.10: Runtime parameters used with the **Jeans** test problem.

Variable	Type	Default	Description
<code>rho0</code>	real	$1.5 \times 10^7$	Initial unperturbed density ( $\rho_0$ )
<code>p0</code>	real	$1.5 \times 10^7$	Initial unperturbed pressure ( $p_0$ )
<code>amplitude</code>	real	0.001	Perturbation amplitude ( $\delta$ )
<code>lambdax</code>	real	0.572055	Perturbation wavelength in $x$ direction ( $\lambda_x = 2\pi/k_x$ )
<code>lambday</code>	real	$1.0 \times 10^{10}$	Perturbation wavelength in $y$ direction ( $\lambda_y = 2\pi/k_y$ )
<code>lambdaz</code>	real	$1.0 \times 10^{10}$	Perturbation wavelength in $z$ direction ( $\lambda_z = 2\pi/k_z$ )
<code>delta_ref</code>	real	0.01	Refine a block if the maximum density contrast relative to $\rho_{\text{ref}}$ is greater than this
<code>delta_deref</code>	real	-0.01	Derefine a block if the maximum density contrast relative to $\rho_{\text{ref}}$ is less than this
<code>reference_density</code>	real	$1.5 \times 10^7$	Reference density for grid refinement ( $\rho_{\text{ref}}$ ). Density contrast is used to determine which blocks to refine; it is defined as

$$\max_{\text{block}} \left\{ \left| \frac{\rho_{ijk}}{\rho_{\text{ref}}} - 1 \right| \right\}$$

### 35.3.2 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

$$p_0 \ll \frac{4\pi G}{\gamma} \rho_0^2 r_0^2. \quad (35.36)$$

We refer to such a nearly pressureless fluid as ‘dust’. A perfect-gas equation of state is used, but the value of  $\gamma$  is not significant. Outflow boundary conditions are used for the gas, while isolated boundary conditions are used for the gravitational field.

The collapse of the dust sphere is self-similar; the cloud should remain spherical with uniform density as it collapses. The radius of the cloud,  $r(t)$ , should satisfy

$$\left(\frac{8\pi G}{3}\rho_0\right)^{1/2} t = \left(1 - \frac{r(t)}{r_0}\right)^{1/2} \left(\frac{r(t)}{r_0}\right)^{1/2} + \sin^{-1}\left(1 - \frac{r(t)}{r_0}\right)^{1/2} \quad (35.37)$$

(Colgate & White 1966). Thus, we expect to test three things with this problem: the ability of the code to maintain spherical symmetry during an implosion (in particular, no block boundary effects should be evident); the ability of the code to keep the density profile constant within the cloud; and the ability of the code to obtain the correct collapse factor. The second of these is particularly difficult, because the edge of the cloud is very sharp and because the Cartesian grid breaks spherical symmetry most dramatically at the center of the cloud, which is where all of the matter ultimately ends up.

Results of a `DustCollapse` run using FLASH 3.0 appear in [Figure 35.60](#), which shows plots of density and the X component of velocity in menacing color scheme. The values are plotted at the end of the run from an X-Y plane in the center of the physical domain; density is in logarithmic scale. This run used a resolution of  $128^3$ , and the results were compared against a similar run using FLASH 2.5. We have also included figures from an earlier higher resolution run using FLASH2 which used  $4^3$  top-level blocks and seven levels of refinement, for an effective resolution of  $2048^3$ . In both the runs, the multipole Poisson solver was used with a maximum multipole moment  $\ell = 0$ . The initial conditions used  $\rho_0 = 10^9 \text{ g cm}^{-3}$  and  $r_0 = 6.5 \times 10^8 \text{ cm}$ . In [Figure 35.61a](#), 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  $7.30 \times 10^9 \text{ cm s}^{-1}$ , respectively. In [Figure 35.61b](#) 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 (35.37). Overall our results, including the numerical effects, agree well with those of Mönchmeyer and Müller (1989).

This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.67 and is run in a three-dimensional box. The problem uses the specialized refinement marking routine supplied under the Grid interface of `Grid_markRefineSpecialized` which refines blocks containing the center of the cloud.

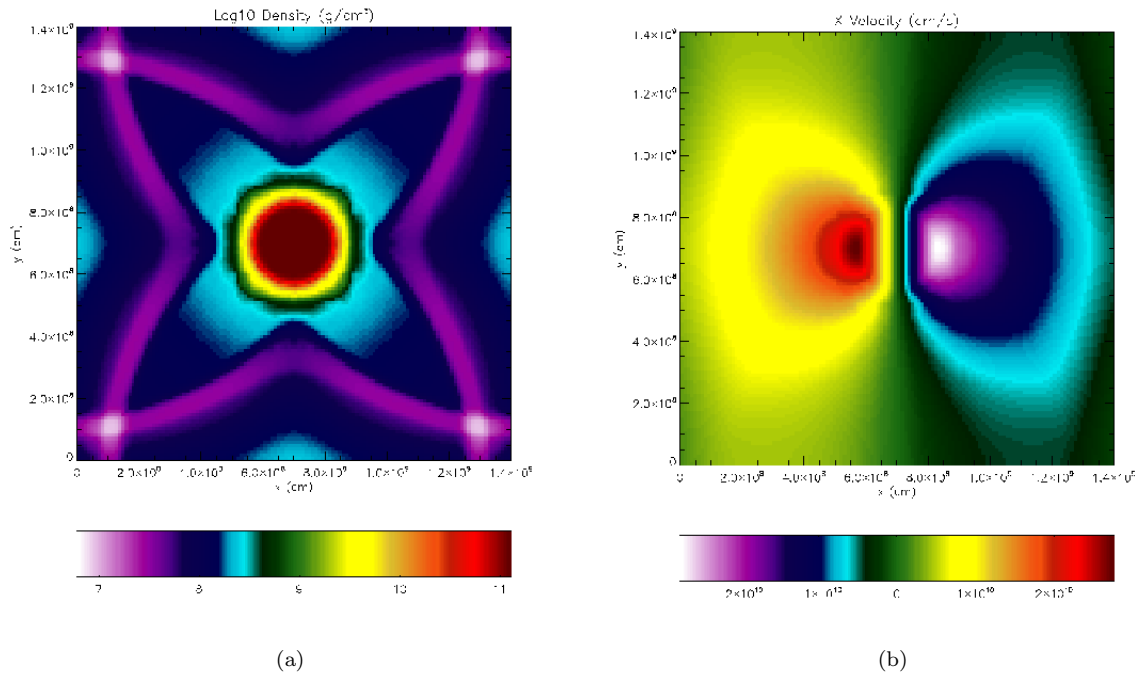


Figure 35.60: XY plane of Density (a) and X component of Velocity (b) are shown at the center of the domain for the DustCollapse problem. The velocity is in normal scale, while density is logscale.

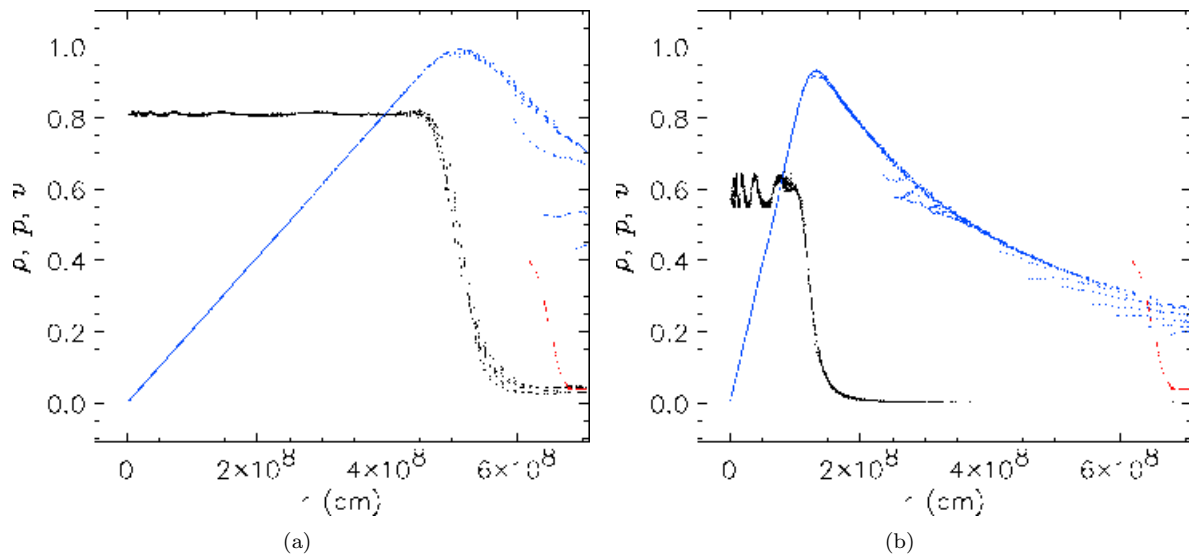


Figure 35.61: Density (black), pressure (red), and velocity (blue) profiles in the homologous dust collapse problem at (a)  $t = 0.0368$  sec and (b)  $t = 0.0637$  sec. The density, pressure, and velocity are scaled as discussed in the text.

### 35.3.3 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

$$\rho(x, y) = \sum_{i=1}^{13} e^{-\sigma_i[(x-x_i)^2+(y-y_i)^2]} , \quad (35.38)$$

where the constants  $\sigma_i$ ,  $x_i$ , and  $y_i$  are given in [Table 35.11](#). The very large range of widths and ellipticities of these peaks forces the mesh structure to be highly refined in some places. The density field and block structure are shown for a 14-level mesh in [Figure 35.62](#).

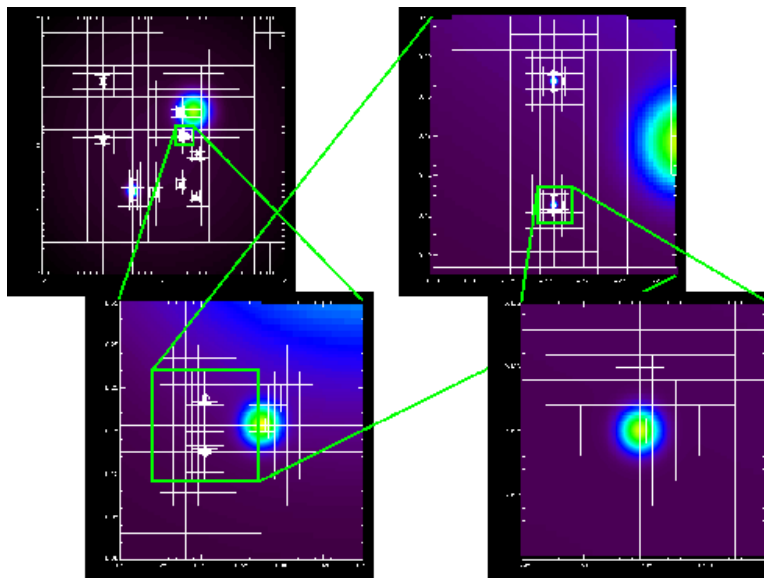


Figure 35.62: Density field and block structure for a 14-level mesh applied to the Huang-Greengard PoissonTest problem. The effective resolution of the mesh is  $65,536^2$ .

Table 35.11: Constants used in the **poistest** problem.

$i$	1	2	3	4	5	6	7
$x_i$	0	-1	-1	0.28125	0.5	0.3046875	0.3046875
$y_i$	0	0.09375	1	0.53125	0.53125	0.1875	0.125
$\sigma_i$	0.01	4000	20000	80000	16	360000	400000
$i$	8	9	10	11	12	13	
$x_i$	0.375	0.5625	-0.5	-0.125	0.296875	0.5234375	
$y_i$	0.15625	-0.125	-0.703125	-0.703125	-0.609375	-0.78125	
$\sigma_i$	2000	18200	128	49000	37000	18900	

The **PoisTest** problem uses one additional runtime parameters **sim.smlRho**, the smallest allowed value of density. Runtime parameters from the **Multigrid** unit (both **Gravity** and **GridSolvers**) are relevant; see [Section 8.10.2.6](#).



### 35.3.4 MacLaurin

The gravitational potential at the surface of, and inside a homogeneous spheroid called a “MacLaurin spheroid” is expressible in terms of analytical functions. This handy result was first determined by MacLaurin (1801), and later summarized by, amongst others, Chandrasekhar (1989). These properties allow validation of the FLASH4 gravitational solvers against the analytical solutions.

As a test case, an oblate ( $a_1 = a_2 > a_3$ ) MacLaurin spheroid, of a constant density  $\rho = 1$  in the interior, and  $\rho = \epsilon \rightarrow 0$  outside (in FLASH4 `smrho` is used). The spheroid is motionless and in hydrostatic equilibrium. The gravitational potential of such object is analytically calculable, and is:

$$\phi(\mathbf{x}) = \pi G \rho [2A_1 a_1^2 - A_1(x^2 + y^2) + A_3(a_3^2 - z^2)] , \quad (35.39)$$

for a point inside the spheroid. Here

$$A_1 = \frac{\sqrt{1-e^2}}{e^3} \sin^{-1} e - \frac{1-e^2}{e^2} , \quad (35.40)$$

$$A_3 = \frac{2}{e^2} - \frac{2\sqrt{1-e^2}}{e^3} \sin^{-1} e , \quad (35.41)$$

where  $e$  is the ellipticity of a spheroid:

$$e = \sqrt{1 - \left(\frac{a_3}{a_1}\right)^2} . \quad (35.42)$$

For a point outside the spheroid, potential is:

$$\phi(\mathbf{x}) = \frac{2a_3}{e^2} \pi G \rho \left[ a_1 e \tan^{-1} h - \frac{1}{2} \left( (x^2 + y^2) \left( \tan^{-1} h - \frac{h}{1+h^2} \right) + 2z^2(h - \tan^{-1} h) \right) \right] , \quad (35.43)$$

where

$$h = \frac{a_1 e}{\sqrt{a_3^2 + \lambda}} , \quad (35.44)$$

and  $\lambda$  is the positive root of the equation

$$\frac{x^2}{a_1^2 + \lambda} + \frac{y^2}{a_2^2 + \lambda} + \frac{z^2}{a_3^2 + \lambda} = 1 . \quad (35.45)$$

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 35.3.4](#). 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  $l_{max}$ , specified by the user as runtime parameter `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:

$$\epsilon = \left| \frac{\phi_{\text{analytical}} - \phi_{\text{FLASH}}}{\phi_{\text{analytical}}} \right| \quad (35.46)$$

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  $\ell$  in (8.15). In [Figure 35.64–35.64](#) the gravitational potential for the MacLaurin spheroid, the FLASH4 solution, and relative errors for several  $l_{max}$ ’s are shown. A similar figure produced for  $l_{max} = 1$  shows no difference from [Figure 35.64](#), indicating that the last dipole term in the multipole expansion does not contribute to the accuracy of the solution but does increase computational

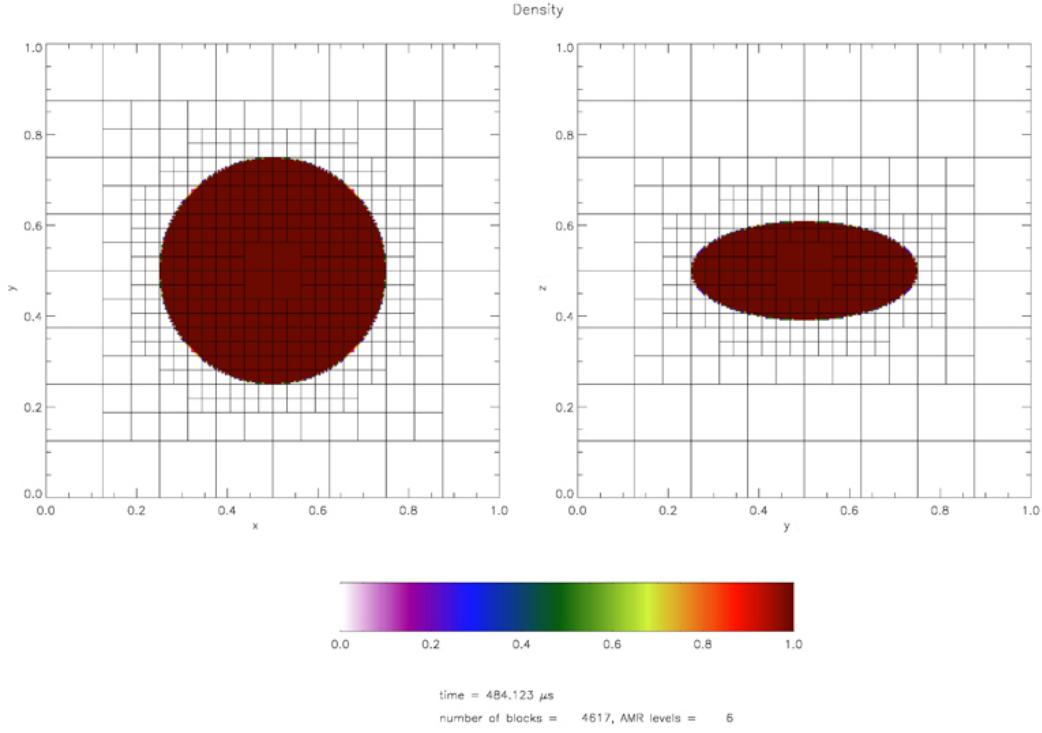


Figure 35.63: Density of the MacLaurin spheroid (left X–Y plane, right Y–Z plane) with ellipticity  $e = 0.9$ . The FLASH4 block structure is shown on top.

$l_{max}$	$\min(\epsilon)$	$\max(\epsilon)$	relative $L_{in}^2$ norm	relative $L_{out}^2$ norm	approx. time [s]
0	$4.5 \cdot 10^{-6}$	0.182	$7.1 \cdot 10^{-2}$	$6.8 \cdot 10^{-2}$	9.8
1	$4.5 \cdot 10^{-6}$	0.182	$7.1 \cdot 10^{-2}$	$6.8 \cdot 10^{-2}$	14.5
2	$9.8 \cdot 10^{-6}$	0.062	$1.4 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	34.7
4	$1.0 \cdot 10^{-8}$	0.026	$4.0 \cdot 10^{-3}$	$5.0 \cdot 10^{-3}$	55.4
6	$6.1 \cdot 10^{-9}$	0.013	$1.6 \cdot 10^{-3}$	$2.5 \cdot 10^{-3}$	134.9
8	$7.8 \cdot 10^{-9}$	0.007	$8.7 \cdot 10^{-4}$	$1.2 \cdot 10^{-3}$	210.2
10	$6.7 \cdot 10^{-9}$	0.004	$5.5 \cdot 10^{-4}$	$7.0 \cdot 10^{-4}$	609.7

Table 35.12: Minimal and maximal relative error in all zones of the simulation, calculated using (35.46). Last row is approximate time for one full timestep (gravity only).

cost. Because gravity sources are all of the same sign, and the symmetry of the problem, all odd- $l$  moments are zero: reasonable, physically motivated values for setting `mpole_lmax` should be an even number.

In the X–Y plane, where the solution is radially symmetric, the first monopole term is enough to qualitatively capture the correct potential. As expected, the error is the biggest on the spheroid boundary, and decreases both outwards and inwards. Increasing the maximum included moment reduces errors. However, in other non-symmetric planes, truncating the potential to certain  $l_{max}$  leads to an error whose leading term will be the spherical harmonic of order  $l_{max} + 2$ , as can be nicely seen in the lower right sections of Figure 35.64 – 35.66. Increasing  $l_{max}$  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  $l_{max} = 4$ , 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.

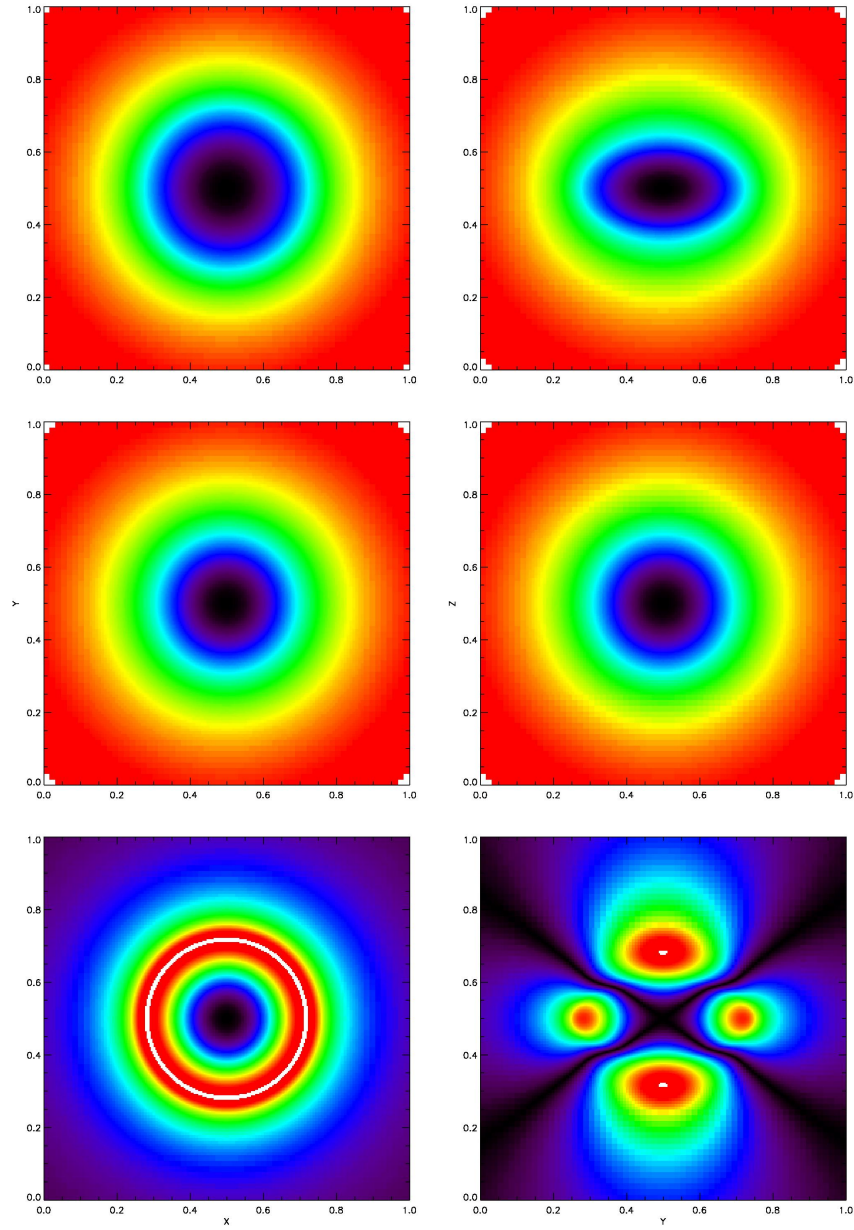


Figure 35.64: Maclaurin spheroid:  $l_{max} = 0$ , 6 refinement levels. Left column is X-Y plane, cut through  $z=0.5$ , right column is Y-Z plane cut through  $x=0.5$ . From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.

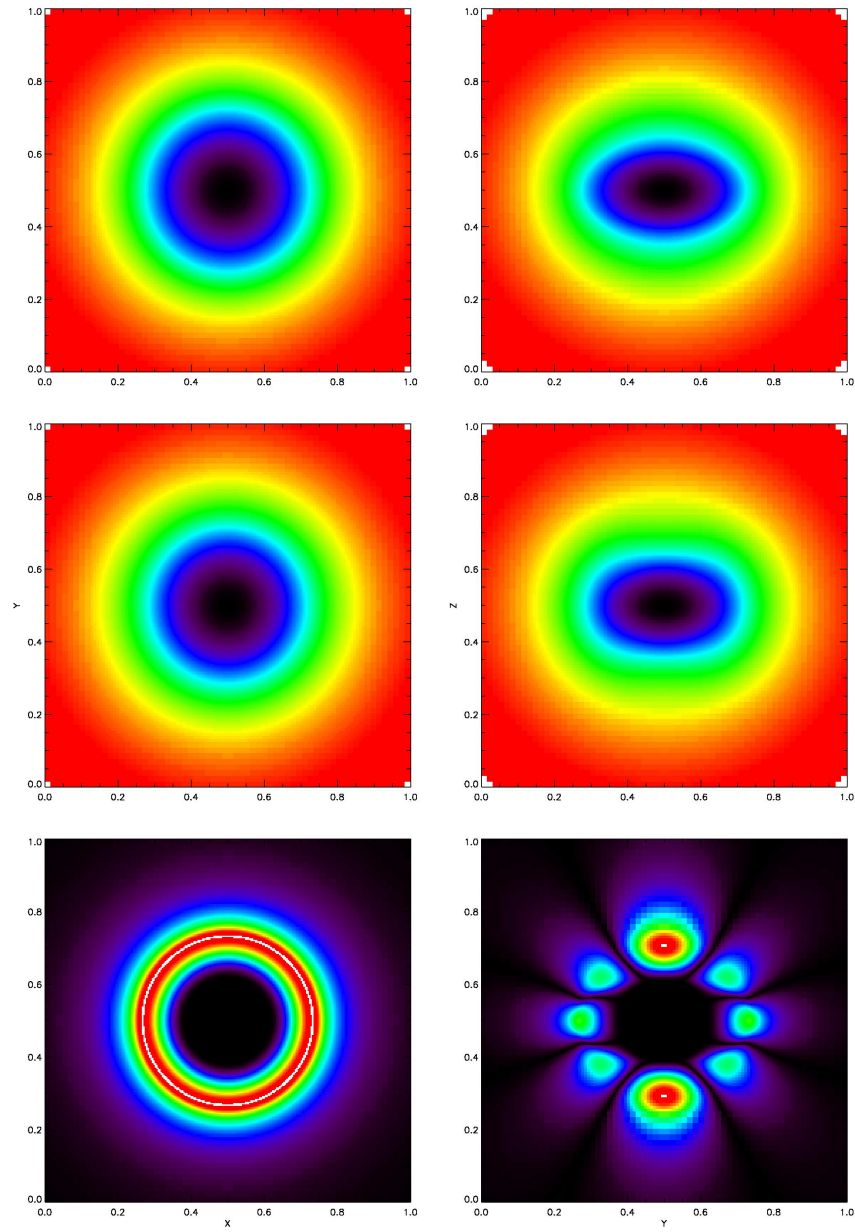


Figure 35.65: Maclaurin spheroid:  $l_{max} = 2$ , 6 refinement levels. Left column is X-Y plane, cut through  $z=0.5$ , right column is Y-Z plane cut through  $x=0.5$ . From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.

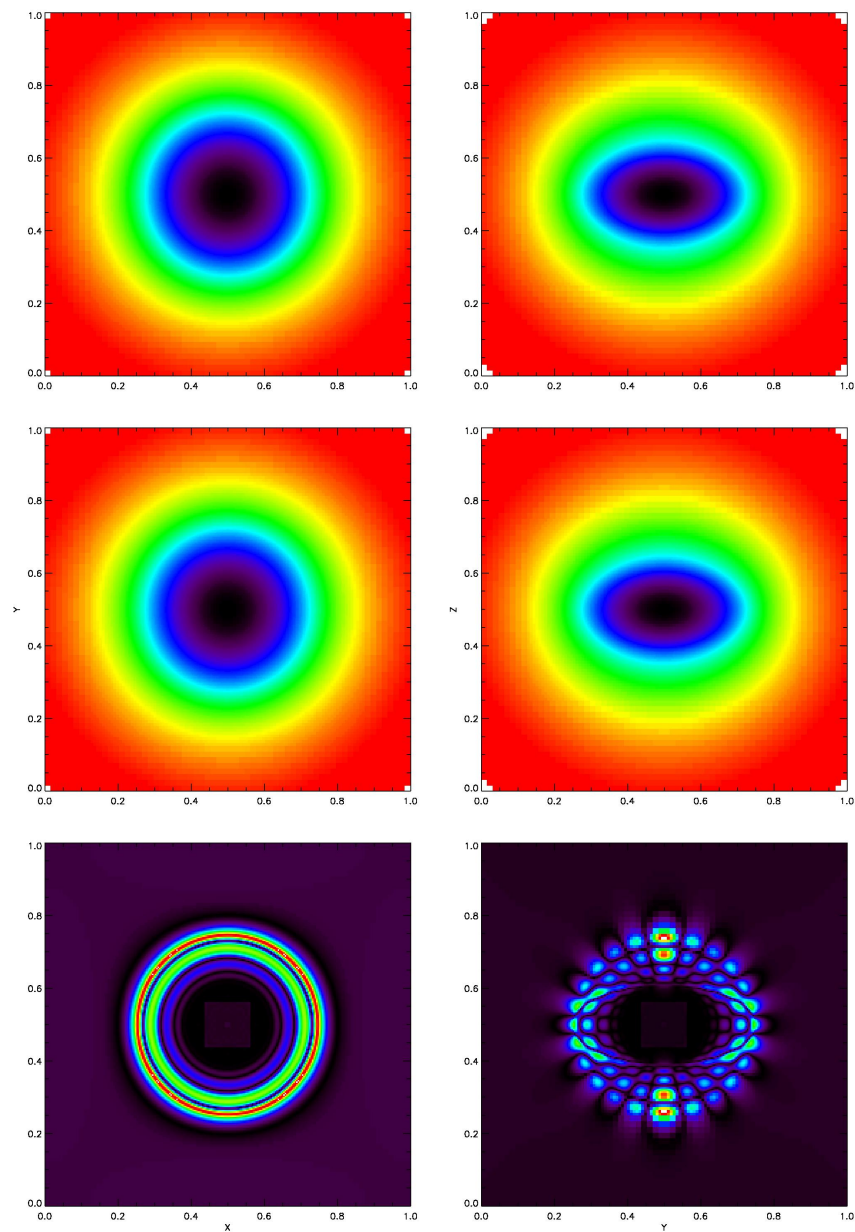


Figure 35.66: Maclaurin spheroid:  $l_{max} = 10$ , 6 refinement levels. Left column is X-Y plane, cut through  $z=0.5$ , right column is Y-Z plane cut through  $x=0.5$ . From top to bottom: analytical solution for the gravitational potential introduced on FLASH grid; solution of FLASH multipole solver; relative error.

## 35.4 Particles Test Problems

These problems are primarily designed to test the functioning of the particle tracking routines within FLASH4.

### 35.4.1 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

$$|v| = \sqrt{\frac{2GM}{r_0}}, \quad (35.47)$$

if a constant gravitational field due to a point mass  $M$  at  $(0.5L_x, 0.5L_y, 0.5L_z)$  is employed, or

$$|v| = \frac{1}{2} \sqrt{\frac{2G}{r_0}}, \quad (35.48)$$

if the particles are self-gravitating. The correct behavior is for the particles to orbit the center of the grid in a circle with constant velocity. Figure 35.67 shows a typical pair of particle trajectories for this problem

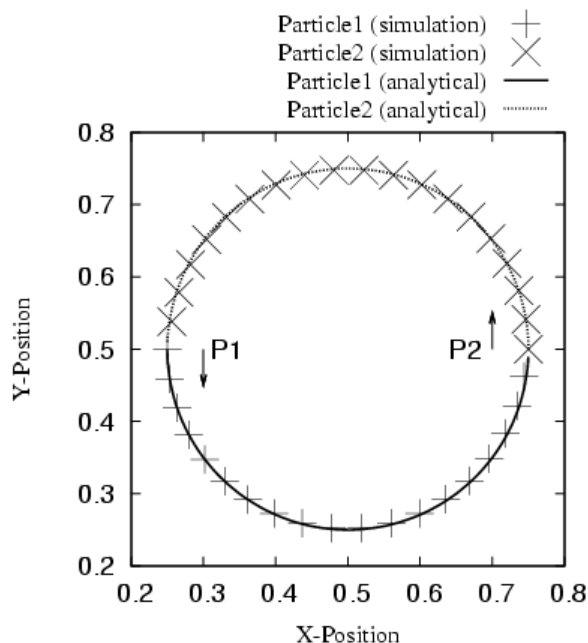


Figure 35.67: Particle trajectories in the `Orbit` test problem for a 3D grid at a fixed refinement level of 2. There is no motion along the  $z$ -axis.

No specific gravity unit is required by the problem configuration file, because the problem is intended to be run with either a fixed external field or the particles' own field. If the particles are to orbit in an external field (`ext_field = .true.`), the field is assumed to be a central point-mass field (`physics/Gravity/GravityMain/PointMass`), and the parameters for that unit should be assigned appropriate values. If the particles are self-gravitating (`ext_field = .false.`), the `physics/Gravity/GravityMain/Poisson`



unit should be included in the code, and a Poisson solver that supports isolated boundary conditions should be used (`grav_boundary_type = "isolated"`).

In either case, long-range forces for the particles must be turned on, or else they will not experience any accelerations at all. This can be done using the particle-mesh method by including the unit `Particles/ParticlesMain/active/longRange/gravity/ParticleMesh`.

#### FLASH Transition

Although the Multipole solver can work with the Orbit problem, the solutions are very poor. We strongly recommend the use of Multigrid solver with this problem.

As of FLASH 2.1 both the multigrid and multipole solvers support isolated boundary conditions. This problem should be run in three dimensions.

#### Grid interpolation

The FLASH2 user guide recommends that this problem be run with conservative, quadratic interpolants (such as `mesh/amr/paramesh2.0/quadratic_cartesian`) and monotonicity enforcement turned off (`monotone = .false.`). In FLASH4, 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 35.13](#) in addition to the regular ones supplied with the code.

Table 35.13: Runtime parameters used in the `orbit` test problem.

Variable	Type	Default	Description
<code>separation</code>	real	0.5	Initial particle separation ( $r_0$ )
<code>ext_field</code>	logical	<code>.false.</code>	Whether to make the particles self-gravitating or to have them orbit in an external potential. In the former case <code>GravityMain/Poisson</code> should be used; in the latter, <code>GravityMain/PointMass</code> .

### 35.4.2 Zel'dovich Pancake

The cosmological pancake problem (Zel'dovich 1970), **Pancake**, provides a good simultaneous test of the hydrodynamics, particle dynamics, Poisson solver, and cosmological expansion modules. Analytic solutions well into the nonlinear regime are available for both  $N$ -body and hydrodynamical codes (Anninos & Norman 1994), permitting an assessment of the code's accuracy. After caustic formation the problem provides a stringent test of the code's ability to track thin, poorly resolved features and strong shocks using most of the basic physics needed for cosmological problems. Also, as pancakes represent single-mode density perturbations, coding this test problem is useful as a basis for creating more complex cosmological initial conditions.

We set the initial conditions for the pancake problem in the linear regime using the analytic solution given by Anninos and Norman (1994). In a universe with  $\Omega_0 = 1$  at redshift  $z$ , a perturbation of wavenumber

$k$  which collapses to a caustic at redshift  $z_c < z$  has comoving density and velocity given by

$$\rho(x_e; z) = \bar{\rho} \left[ 1 + \frac{1+z_c}{1+z} \cos(kx_\ell) \right]^{-1} \quad (35.49)$$

$$v(x_e; z) = -H_0(1+z)^{1/2}(1+z_c) \frac{\sin kx_\ell}{k}, \quad (35.50)$$

where  $\bar{\rho}$  is the comoving mean density. Here  $x_e$  is the distance of a point from the pancake midplane, and  $x_\ell$  is the corresponding Lagrangian coordinate, found by iteratively solving

$$x_e = x_\ell - \frac{1+z_c}{1+z} \frac{\sin kx_\ell}{k}. \quad (35.51)$$

The temperature solution is determined from the density under the assumption that the gas is adiabatic with ratio of specific heats  $\gamma$ :

$$T(x_e; z) = (1+z)^2 \bar{T}_{\text{fid}} \left[ \left( \frac{1+z_{\text{fid}}}{1+z} \right)^3 \frac{\rho(x_e; z_{\text{fid}})}{\rho(x_e; z)} \right]^{\gamma-1}. \quad (35.52)$$

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  $x_e$  are computed using (35.51). Particle velocities are assigned using (35.50).

At caustic formation ( $z = z_c$ ), planar shock waves form in the gas on either side of the pancake midplane and begin to propagate outward. A small region at the midplane is left unshocked. Immediately behind the shocks, the comoving density and temperature vary approximately as

$$\rho(x_e; z) \approx \bar{\rho} \frac{18}{(kx_\ell)^2} \frac{(1+z_c)^3}{(1+z)^3} \quad (35.53)$$

$$T(x_e; z) \approx \frac{\mu H_0^2}{6k_B k^2} (1+z_c) (1+z)^2 (kx_\ell)^2.$$

At the midplane, which undergoes adiabatic compression, the comoving density and temperature are approximately

$$\begin{aligned} \rho_{\text{center}} &\approx \bar{\rho} \left[ \frac{1+z_{\text{fid}}}{1+z} \right]^3 \left[ \frac{3H_0^2 \mu}{k_B \bar{T}_{\text{fid}} k^2} \frac{(1+z_c)^4 (1+z)^3}{1+z_{\text{fid}}} \right]^{1/\gamma} \\ T_{\text{center}} &\approx \frac{3H_0^2 \mu}{k_B k^2} (1+z)^2 (1+z_c)^4 \frac{\bar{\rho}}{\rho_{\text{center}}}. \end{aligned} \quad (35.54)$$

An example FLASH calculation of the post-caustic gas solution appears in Figure 35.68.

Because they are collisionless, the particles behave very differently than the gas. As particles accelerate toward the midplane, their phase profile develops a backwards “S” shape. At caustic formation the velocity becomes multivalued at the midplane. The region containing multiple streams grows in size as particles pass through the midplane. At the edges of this region (the caustics, or the inflection points of the “S”), the particle density is formally infinite, although the finite force resolution of the particles keeps the height of these peaks finite. Some of the particles that have passed through the midplane fall back and form another pair of caustics, twisting the phase profile again. Because each of these secondary caustics contains five streams of particles rather than three, the second pair of density peaks are higher than the first pair. This caustic formation process repeats arbitrarily many times in the analytic solution. In practice, the finite number of particles and the finite force resolution limit the number of caustics that are observed. An example FLASH calculation of the post-caustic particle solution appears in Figure 35.69.

The 2D **pancake** problem in FLASH4 uses the runtime parameters listed in Table 35.14 in addition to the regular ones supplied with the code.

This problem uses periodic boundary conditions and is intrinsically one-dimensional, but it can be run using Cartesian coordinates in 1D, 2D, or 3D, with the pancake midplane tilted with respect to the coordinate axes if desired.



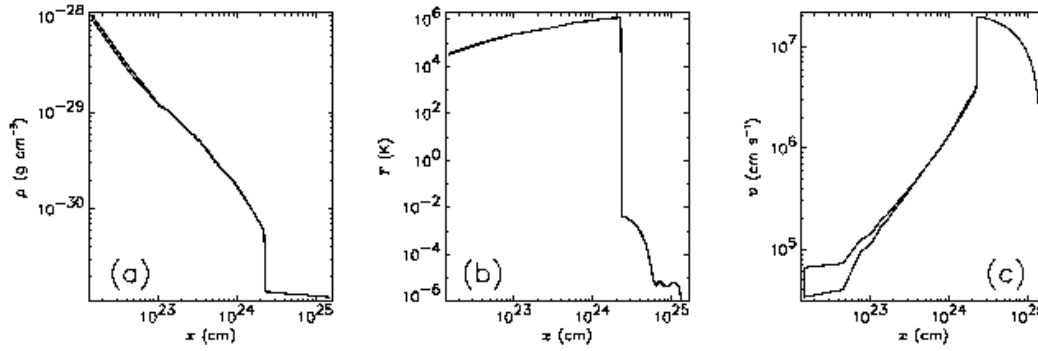


Figure 35.68: Example solution for the gas in a mixed particle/gas Zel’dovich **Pancake** problem. A comoving wavelength  $\lambda = 10$  Mpc, caustic redshift  $z_c = 5$ , fiducial redshift  $z_{\text{fid}} = 200$ , and fiducial temperature  $T_{\text{fid}} = 550$  K were used together with a Hubble constant of  $50 \text{ km s}^{-1} \text{ Mpc}^{-1}$ . 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.

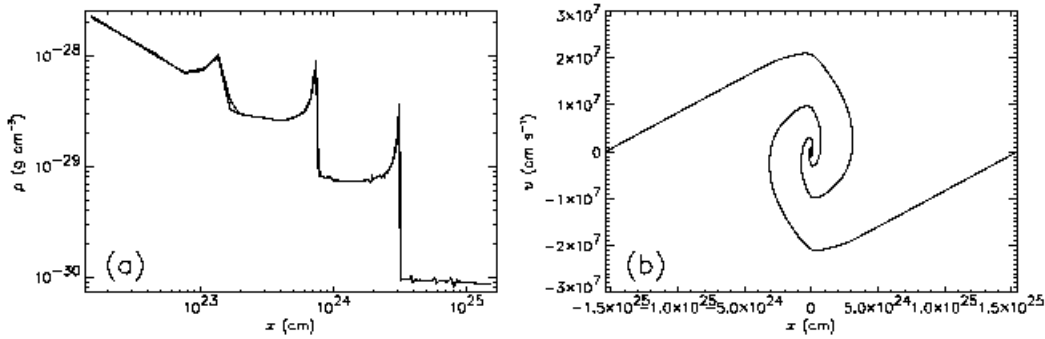


Figure 35.69: Example solution for the dark matter in a mixed particle/gas Zel’dovich **pancake**. Perturbation and cosmological parameters were the same as in [Figure 35.68](#). Results are shown for redshift  $z = 0$ . An adaptive mesh with an effective resolution of 1024 cells was used. The number of particles used was 8192. Other parameters for this run were as described in the text. Distance  $x$  is measured from the pancake midplane. (a) Dark matter density. (b) Dark matter phase diagram showing particle positions  $x$  and velocities  $v$ .

Table 35.14: Runtime parameters used with the 2D **pancake** test problem.

Variable	Type	Default	Description
lambda	real	$3.0857 \times 10^{25}$	Wavelength of the initial perturbation ( $2\pi/k$ )
zcaustic	real	5.	Redshift at which pancake forms a caustic ( $z_c$ )
Tfiducial	real	550.	Fiducial gas temperature ( $T_{\text{fid}}$ )
zfiducial	real	200.	Redshift at which gas temperature is $T_{\text{fid}}$ ( $z_{\text{fid}}$ )
xangle	real	0.	Angle made by pancake normal with the $x$ -axis (degrees)
yangle	real	90.	Angle made by pancake normal with the $y$ -axis (degrees)
pt_numX	integer	128	Number of particles along $x$ -side of initial particle “grid”

<code>pt_numY</code>	integer	128	Number of particles along $y$ -side of initial particle “grid”
<code>pt_numZ</code>	integer	1	Number of particles along $z$ -side of initial particle “grid”

### 35.4.3 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 35.3.3](#). 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 35.15](#) specify that the grid should refine until at most 5 particles exist per block. This creates a refined grid similar to the `Poistest` problem.

Table 35.15: Runtime parameters used with the `PoisParticles` test problem.

Variable	Type	Value	Description
<code>refine_on_particle_count</code>	logical	<code>.true.</code>	On/Off flag for refining the grid according to particle count.
<code>max_particles_per_blk</code>	integer	5	Grid refinement criterion which specifies maximum number of particles per block.

## 35.5 Burn Test Problem

### 35.5.1 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  $^4\text{He}$ ,  $^{12}\text{C}$ ,  $^{16}\text{O}$ ,  $^{20}\text{Ne}$ ,  $^{24}\text{Mg}$ ,  $^{28}\text{Si}$ ,  $^{32}\text{S}$ ,  $^{36}\text{Ar}$ ,  $^{40}\text{Ca}$ ,  $^{44}\text{Ti}$ ,  $^{48}\text{Cr}$ ,  $^{52}\text{Fe}$ , and  $^{56}\text{Ni}$ . It is essential, however, to include  $(\alpha, p)(p, \gamma)$  and  $(\gamma, p)(p, \alpha)$  links in order to obtain reasonably accurate energy generation rates and abundance levels when the temperature exceeds  $\sim 2.5 \times 10^9$  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  $^{27}\text{Al}$ ,  $^{31}\text{P}$ ,  $^{35}\text{Cl}$ ,  $^{39}\text{K}$ ,  $^{43}\text{Sc}$ ,  $^{47}\text{V}$ ,  $^{51}\text{Mn}$ , and  $^{55}\text{Co}$  by assuming steady state proton flows.

The two-dimensional calculations are performed in a planar geometry of size 256.0 cm by 25.0 cm. The initial conditions consist of a constant density of  $10^7$  g cm $^{-3}$ , temperature of  $2 \times 10^8$  K, composition of pure carbon  $X(^{12}\text{C})=1$ , 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

$\text{cm}^{-3}$ , temperature of  $4.423 \times 10^9$  K, and material velocity of  $v_x = 2.876 \times 10^8$   $\text{cm s}^{-1}$ . 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 35.16.

Table 35.16: Runtime parameters used with the Cellular test problem.

Variable	Type	Default	Description
xhe4	real	0.0	Initial mass fraction of He4
xc12	real	1.0	Initial mass fraction of C12
xo16	real	0.0	Initial mass fraction of O16
rhoAmbient	real	$1 \times 10^7$	Density of cold upstream material.
tempAmbient	real	$2 \times 10^8$	Temperature of cold upstream material.
velxAmbient	real	0.0	X-velocity of cold upstream material.
rhoPerturb	real	$4.236 \times 10^7$	Density of the post shock material.
tempPerturb	real	$4.423 \times 10^9$	Temperature of the post shock material.
velxPerturb	real	$2.876 \times 10^8$	X-velocity of the post shock material.
radiusPerturb	real	25.6	Distance below which the perturbation is applied.
xCenterPerturb	real	0.0	X-position of the origin of the perturbation
yCenterPerturb	real	0.0	Y-position of the origin of the perturbation
zCenterPerturb	real	0.0	Z-position of the origin of the perturbation
usePseudo1d	logical	.false.	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.
noiseAmplitude	real	$1.0 \times 10^{-2}$	Amplitude of the white noise added to the perturbation.
noiseDistance	real	5.0	The distance above the starting radius to which white noise is added.

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 35.70 shows the pressure field of the detonation after  $1.26 \times 10^{-7}$  s. The interacting transverse wave structures are particularly vivid, and extend about 25 cm behind the shock front. Figure 35.71 shows a close up of this traverse wave region. Periodic boundary conditions are used at the walls parallel to the y-axis while reflecting boundary conditions were used for the walls parallel to the x-axis.

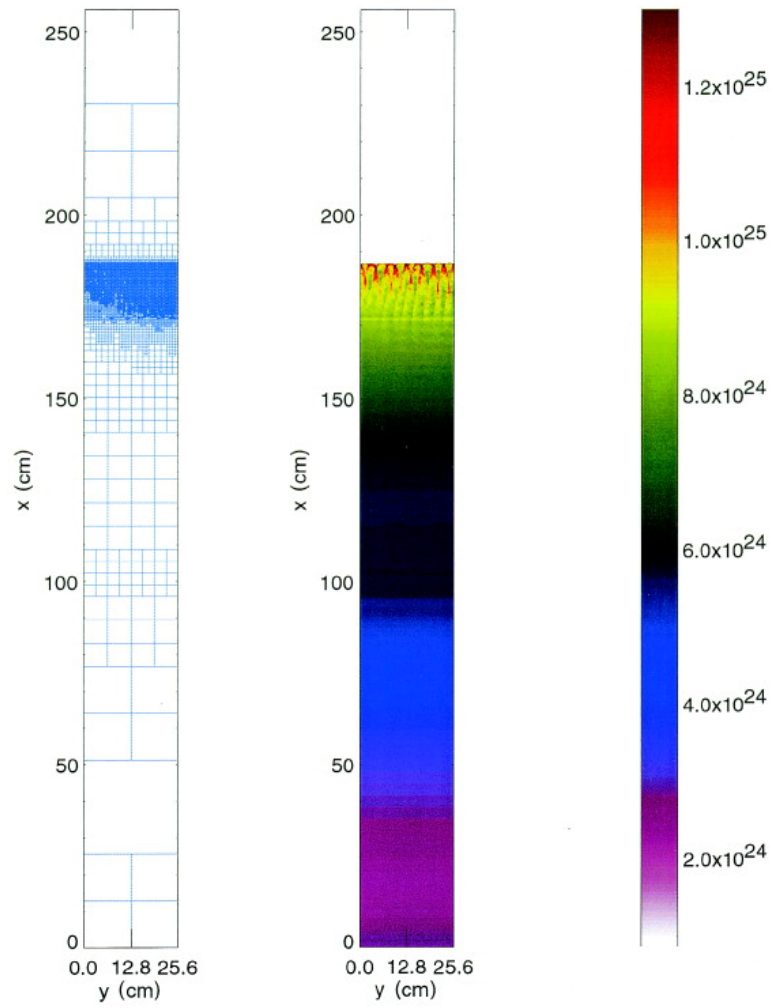


Figure 35.70: Steady-state conditions of the Cellular nuclear burn problem.

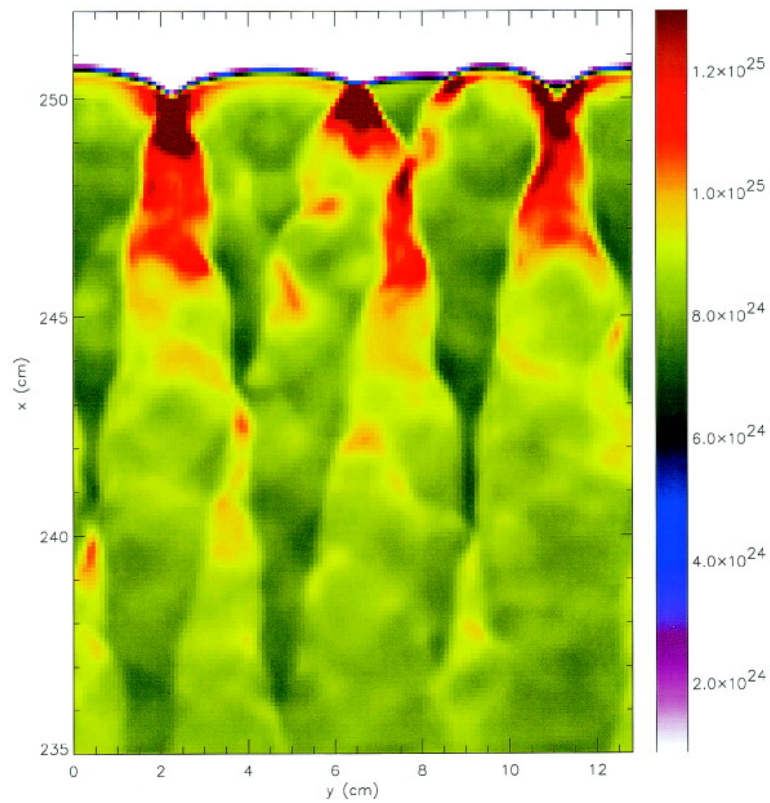


Figure 35.71: Close-up of the detonation front in steady-state for the **Cellular** nuclear burn problem.

## 35.6 RadTrans Test Problems

There are currently two simulations included in FLASH which involve the **RadTrans** unit with multigroup diffusion. These simulations are not comparisons to analytic solutions - instead they are simple tests designed to demonstrate the correct usage of the multigroup diffusion capability in FLASH.

### 35.6.1 Infinite Medium, Energy Equilibration

The simulation directory for this test is **MGDInfinite**. This is a fairly simple simulation which has no spatial gradients. Separate ion/electron/radiation temperatures are set throughout the domain initially. Over time, the temperatures should approach one another. The electrons and the radiation field exchange energy through emission and absorption, and the **HeatExchange** unit controls the rate at which the electrons and ions equilibrate with one another. The radiation field is represented using four radiation energy groups.

A sample setup line is:

```
./setup -auto MGDInfinite -1d +hdf5typeio +mtmmmt +mgd +uhd3t \
  -with-unit=physics/materialProperties/Opacity/OpacityMain/Constant \
  species=be,poli,xe mgd_meshgroups=4
```

Figure 35.72 shows the temperatures as a function of time.

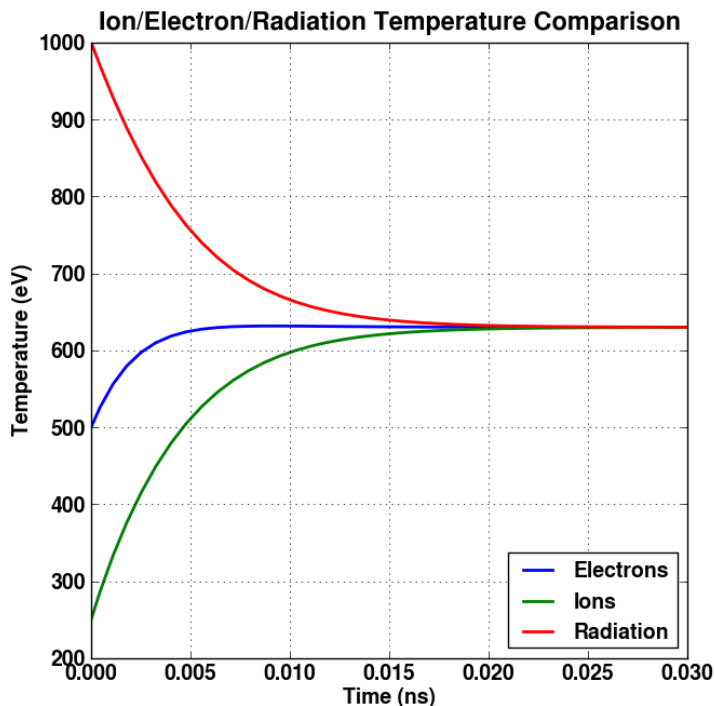


Figure 35.72: Temperatures as a function of time for the Infinite medium, Energy Equilibration simulation

### 35.6.2 Radiation Step Test

The simulation directory for this test is **MGDStep**. This simulation involves four group radiation diffusion and electron conduction. A constant opacity is used in this simulation. The transport opacity is set to a very small value to simulate a vacuum. This is a 1D test where the initial electron and radiation temperatures are discontinuous. Flux limiters are used for both the electrons and radiation. Over time, energy flows from

the hotter region to the colder region. At the same time, the radiation temperature decreases as energy is absorbed by the electrons. **Figure 35.73** shows the final temperature profiles for this simulation. There is a relatively sharp drop-off for each curve caused by the use of flux limiters. The drop-off can be made sharper by increasing the number of cells used in the simulation.

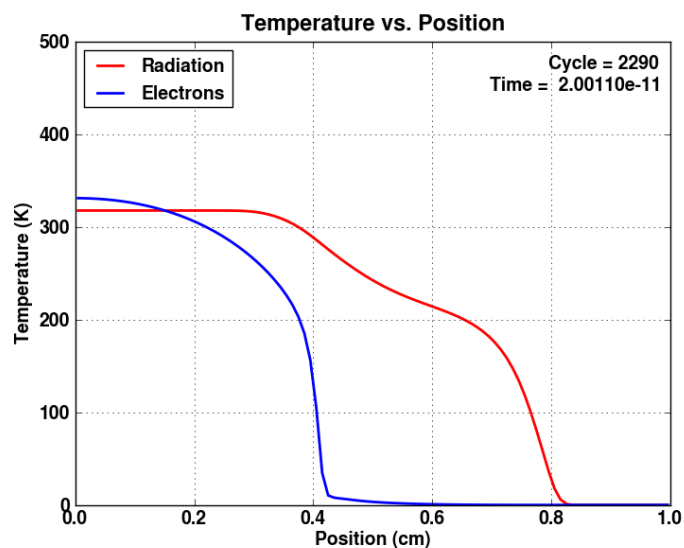


Figure 35.73: Radiation and electron temperatures as a function of position for the MGDStep at 20 ps.

## 35.7 Other Test Problems

### 35.7.1 The non-equilibrium ionization test problem

The `neitest` problem tests the ability of FLASH to calculate non-equilibrium ionization (NEI) ion abundances. It simulates a stationary plasma flow through a temperature step profile. The solutions were checked using an independent stationary code based on a fifth order Runge–Kutta method with adaptive stepsize control by step-doubling (see Orlando *et al.* (1999)).

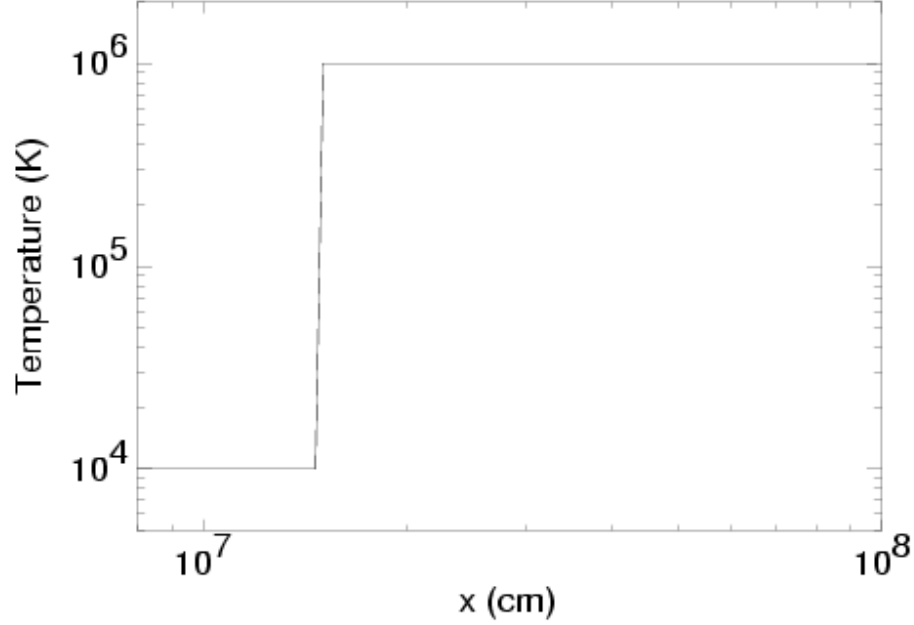


Figure 35.74: Temperature profile assumed for the test.

The test assumes a plasma with a mass density of  $2 \times 10^{-16} \text{ gm cm}^{-3}$  flowing with a constant uniform velocity of  $3 \times 10^5 \text{ cm s}^{-1}$  through a temperature step between  $10^4 \text{ K}$  and  $10^6 \text{ K}$  (*cf.* Figure 35.74). The plasma is in ionization equilibrium before going through the jump in the region at  $T = 10^4 \text{ K}$ . The population fractions in equilibrium are obtained from the equations

$$[n_i^Z]_{eq} S_i^Z = [n_{i+1}^Z]_{eq} \alpha_{i+1}^Z \quad (i = 1, \dots, l_Z - 1) \quad (35.55)$$

$$\sum_{i=1}^{l_Z} [n_i^Z]_{eq} = A_Z n_p \quad (35.56)$$

The presence of a temperature jump causes a strong pressure difference, which in turn should cause significant plasma motions. Since the purpose is to test the NEI module, it is imposed that the pressure difference does not induce any plasma motion and, to this end, the hydro variables (namely,  $T$ ,  $\rho$ ,  $\mathbf{v}$ ) are not updated. In practice, the continuity equations are solved with uniform density and velocity, while the momentum and energy equations are ignored.

Figure 35.75 shows the population fractions for the 12 most abundant elements in astrophysical plasmas derived with the stationary code (Orlando *et al.* (1999)). The out of equilibrium ionization conditions are evident for all the elements just after the flow goes through the temperature jump.



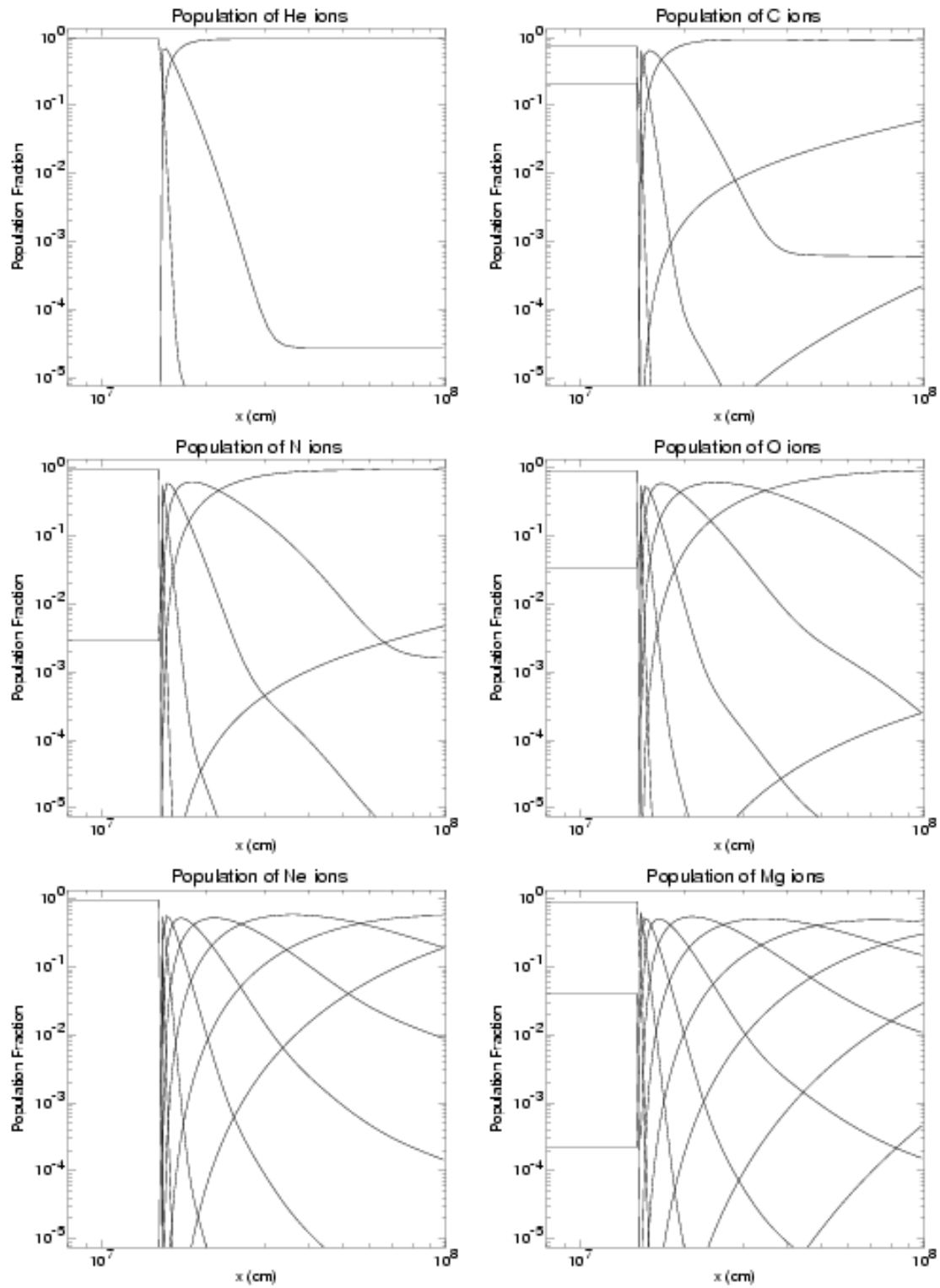


Figure 35.75: Numerical solutions of the stationary code. The figure shows the population fractions vs. space for the 12 elements most abundant in astrophysical plasmas assuming a stationary flow through a temperature jump.

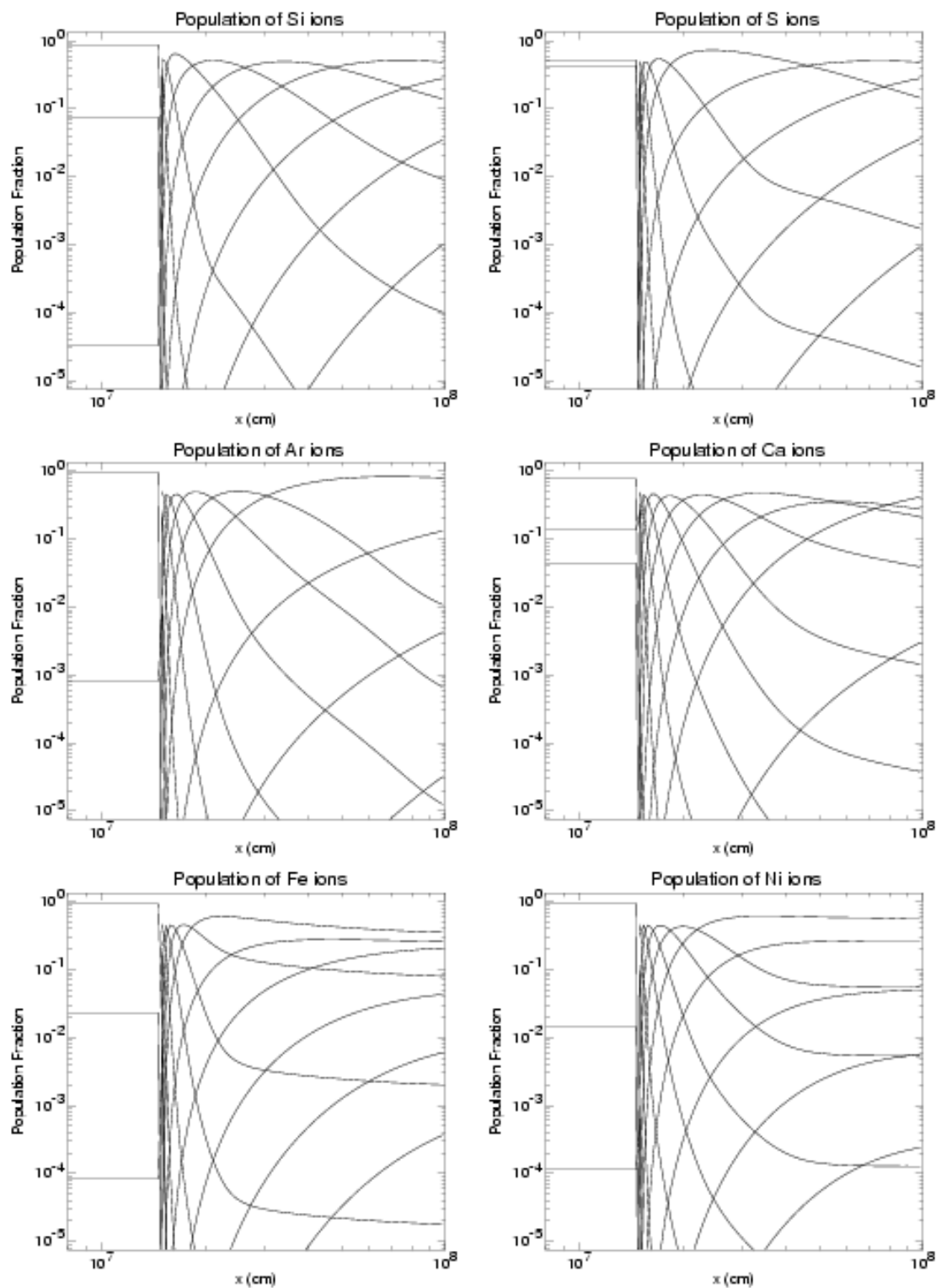


Figure 35.75: ... continued ...

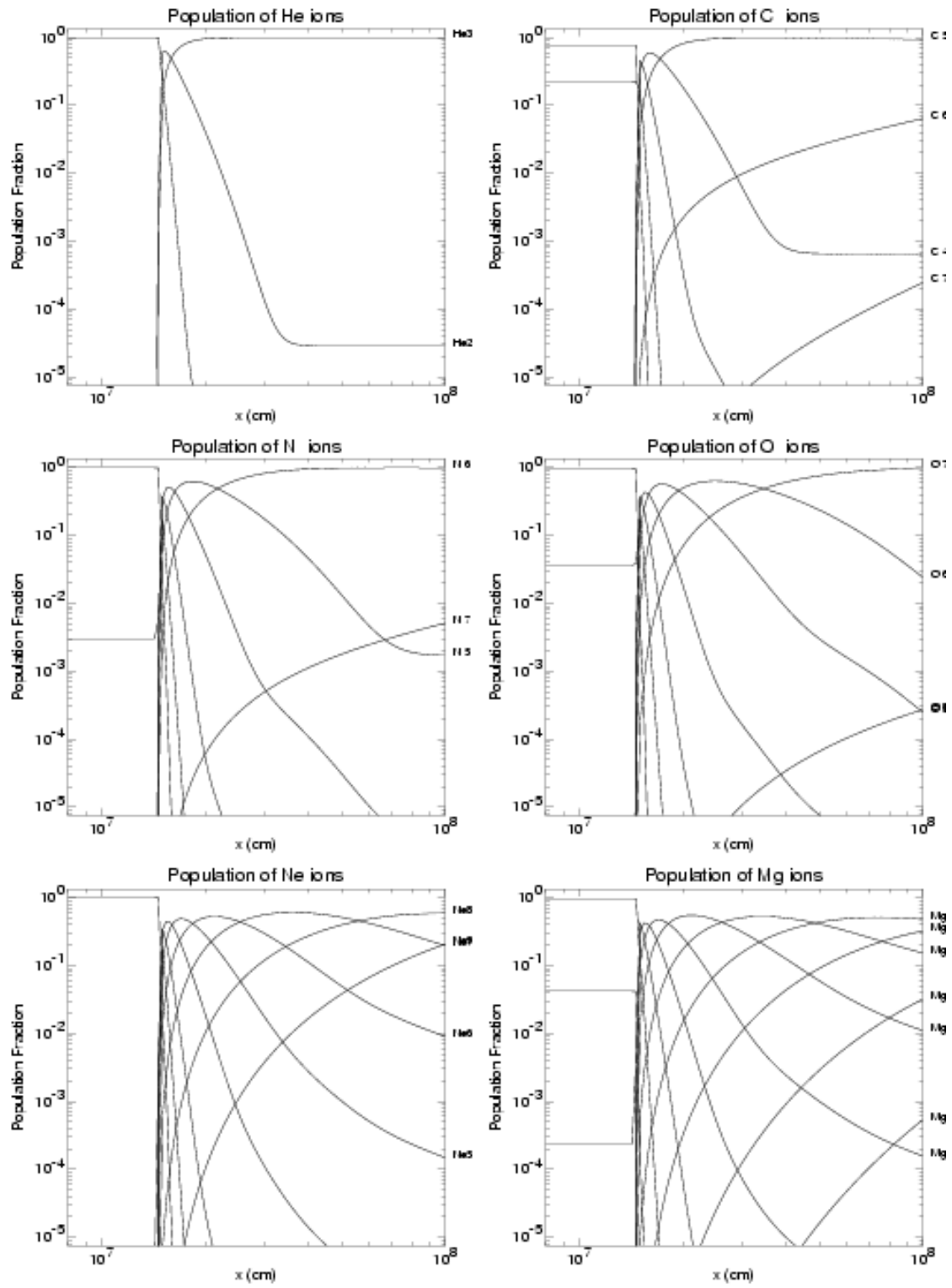


Figure 35.76: As in Figure 35.75 for the solutions of the FLASH code.

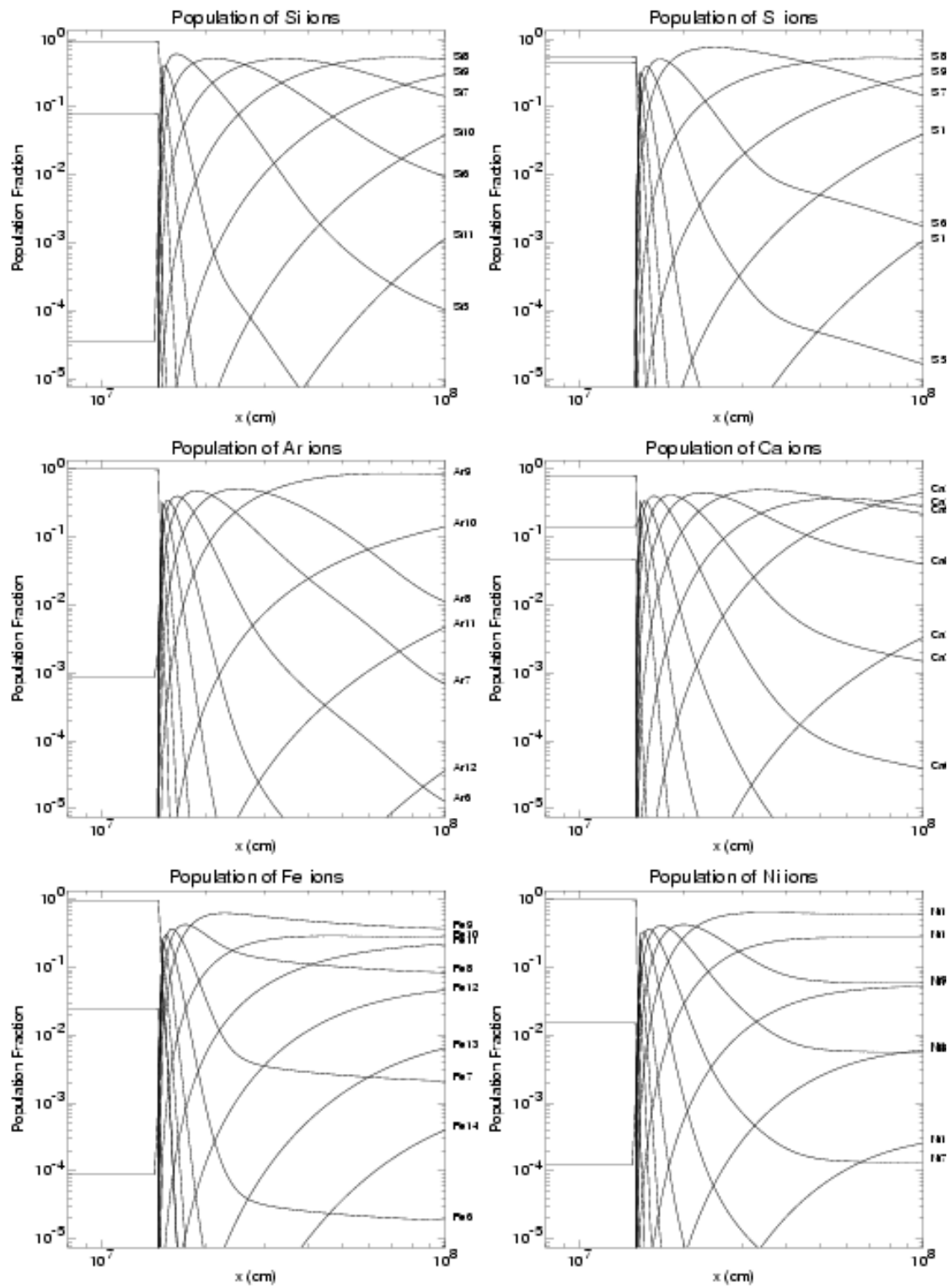


Figure 35.76: ... continued ...

The same problem was solved with the NEI module of the FLASH code, assuming that the plasma is initially in ionization equilibrium at  $t = t_0$  over all the spatial domain. After a transient lasting approximately 700 s, in which the population fractions evolve due to the plasma flow through the temperature jump, the system reaches the stationary configuration. Outflow boundary conditions (zero-gradient) are assumed at both the left and right boundaries. [Figure 35.76](#) shows the population fraction vs. space after 700 s.

### 35.7.2 The Delta-Function Heat Conduction Problem

The `ConductionDelta` problem tests the basic function of the `Diffuse` unit, in particular its use for thermal conduction, in connection with the `Conductivity` material property unit. 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  $t_{\text{ini}}$  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. When using a flux-based `Diffuse` interface, the setup includes the `Hydro` code unit, but changes of the solution due to hydrodynamical effects are completely suppressed by zeroing all hydrodynamic fluxes each time before diffusive fluxes are computed (using `updateHydroFluxes = .FALSE.`); diffusive fluxes are then computed either by calling `Diffuse_therm` (in the `+splitHydro` case) or by calling an internal routine (such as `hy_uhd_addThermalFluxes`) of the `Hydro` unit.

The theoretical solution of this initial value problem is known: For any  $t > t_{\text{ini}}$ , the temperature profile is a Gaussian that spreads self-similarly over time. In particular in 1D, if the initial condition is defined as

$$T(x, t_{\text{ini}}) = Q\delta(x) \quad , \quad (35.57)$$

then

$$T(x, t) = \frac{Q}{(4\pi\chi(t - t_{\text{ini}}))^{1/2}} e^{-x^2/4\chi(t - t_{\text{ini}})} \quad (35.58)$$

(with  $\chi$  the constant coefficient of diffusivity), see for example Zel'dovich and Raizer Ch. X.

See the end of [Section 19.1](#) for alternative ways of configuring the test problem, using either a flux-based or a standalone `Diffuse` interface.

### 35.7.3 The HydroStatic Test Problem

The `Hydrostatic` problem tests the basic function of hydrostatic boundary conditions implemented in the `Grid` unit, in connection with a `Hydro` implementation. It is essentially a 1D problem, but can be configured as 1 1D, 2D, or 3D setup. It can be easily modified to include additional physical effects, by including additional code units in the setup.

In its default configuration, `HydroStatic` is set up with constant `Gravity`. The domain is initialized with density, pressure, *etc.*, fields representing an analytical solution of the hydrostatic problem with the given gravitational acceleration, essentially following the barometric formula.

This initial condition is then evolved in time. Ideally, the solution would remain completely static, and nothing should move. The deviation from this ideal behavior that occurs in practice is a measure of the quality of the discretization of the initial condition, of the hydrodynamics implementation, and of the boundary conditions. The effect of the latter, in particular, can be examined by visualizing artifacts that develop near the boundaries (in particular, velocity artifacts), and studying their dependence on the choice of boundary condition variant.

### 35.7.4 Hybrid-PIC Test Problems

A classic plasma model problem is that of an ion beam into a plasma (Winske 1986). Work by Matthew (1994) describes a two-dimensional simulation of a low density ion beam through a background plasma. The initial condition has a uniform magnetic field, with a beam of ions, number density  $n_b$ , propagating along the field with velocity  $v_b = 10v_A$ , where  $v_A = B/\sqrt{\mu_0 n m_i}$  is the Alfvén velocity, through a background (core) plasma of number density  $n_c$ . Both the background and beam ions have thermal velocities  $v_{th} = v_A$ . Here

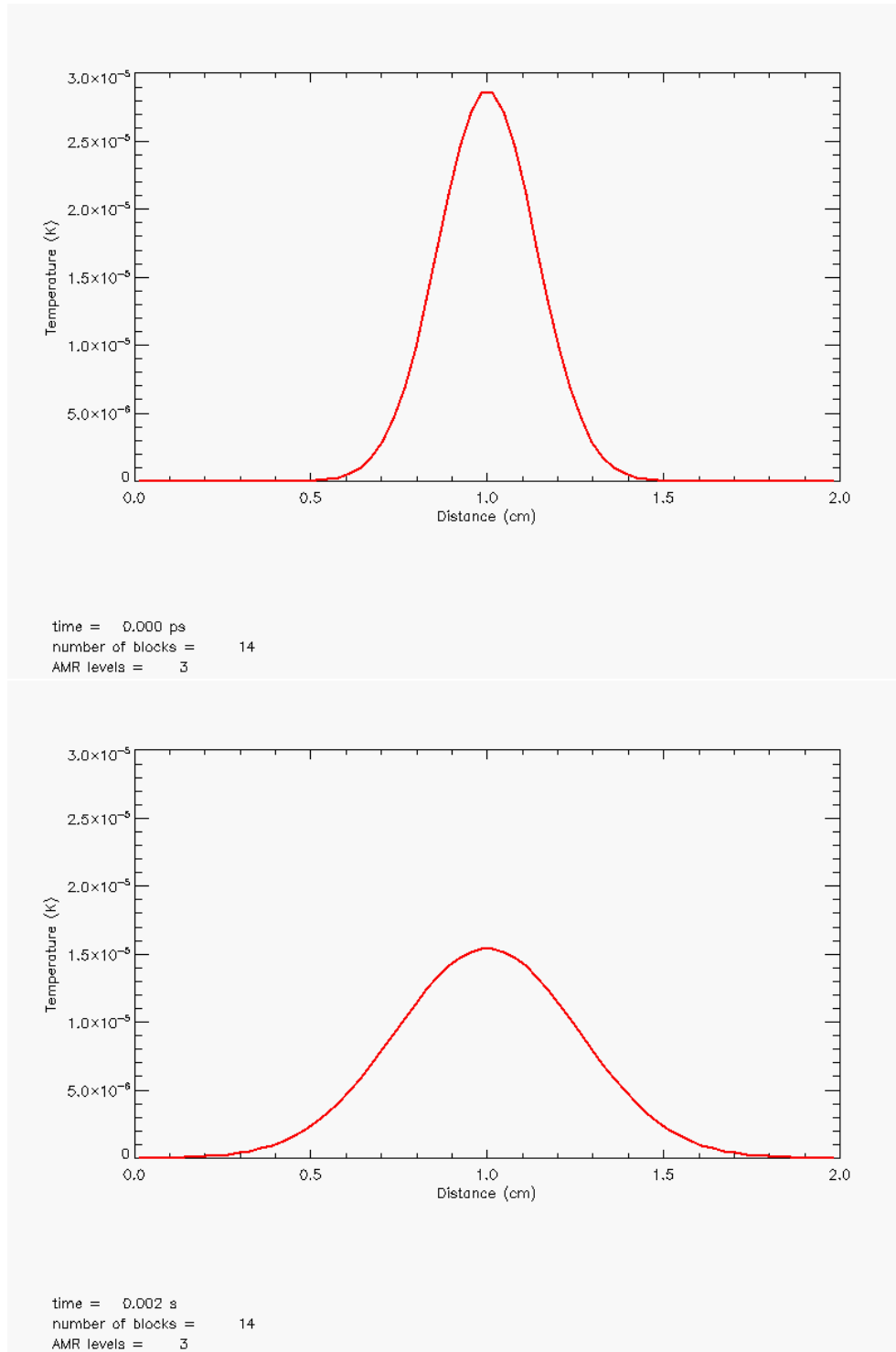


Figure 35.77: Temperature profile of the Delta-Function Heat Conduction Problem at two times.  $t_{\text{ini}} = -1$  ms, top:  $t = 0$  ms, bottom:  $t = 2.46$  ms.

we study what is denoted a *resonant beam* with  $n_b = 0.015 n_c$  and  $v_c = -0.2 v_A$ . Electron temperature is assumed to be zero, so the electron pressure term in the electric field equation of state is zero. The weight of the macroparticles are chosen such that there is an equal number of core and beam macroparticles, each

beam macroparticles thus represent fewer real ions than the core macroparticles. The number of magnetic field update subcycles is nine.

#### 35.7.4.1 One-dimensional ion beam

The spatial extent of the domain is 22016 km along the  $x$ -axis, divided up in 256 cells with periodic boundary conditions. The core number density is  $n_c = 7 \text{ cm}^{-3}$ . The magnetic field magnitude is  $B = 6 \text{ nT}$  directed along the  $x$ -axis, which give an Alfvén velocity of 50 km/s and an ion inertial length of  $\delta_i = 86 \text{ km}$ , where  $\delta_i = v_A/\Omega_i$  and  $\Omega_i = q_i B/m_i$  is the ion gyrofrequency. The time step is  $0.0865 \text{ s} = 0.05 \Omega_i^{-1}$ , and the cell size is  $\delta_i$ . The number of particles per cell is 32. In Fig. 35.78 we show a velocity space plot of the macro-ions at time  $t = 34.6 \text{ s} \approx 20 \Omega_i^{-1}$ . This can be compared to Fig. 5 in Winske (1984).

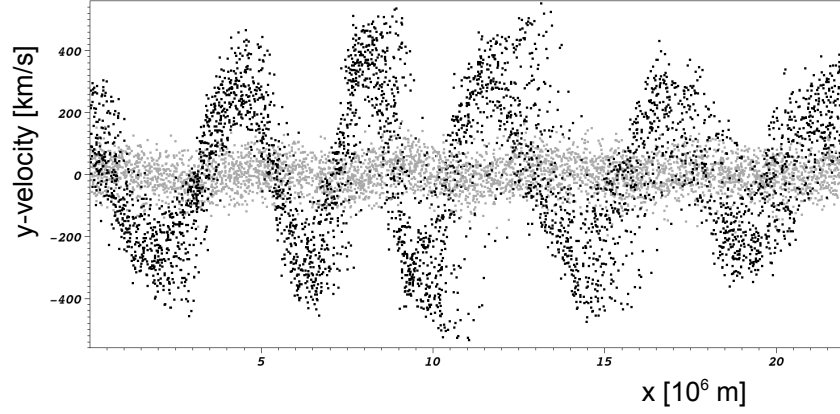


Figure 35.78: Velocity space plot for a one-dimensional ion beam. Velocity along the  $y$ -axis as a function of position at time  $t = 34.6 \text{ s} \approx 20 \Omega_i^{-1}$ . Each gray dot is a core macro-ion, and each black dot is a beam macro-ion.

#### 35.7.4.2 Two-dimensional ion beam

In the two-dimensional case we have a square grid with sides of length 22016 km, and 128 cells in each direction with periodic boundary conditions. The time step is  $0.0216 \text{ s} = 0.05 \Omega_i^{-1}$ , and the cell widths are  $2\delta_i$ . The number of particles per cell is 16. Otherwise the setup is identical to the one-dimensional case. In Fig. 35.79 we show the magnitude of the magnetic field  $y$ -component at time  $t = 77.85 \text{ s} \approx 46 \Omega_i^{-1}$ . This can be compared to Fig. 5 in Winske (1986).

### 35.7.5 Full-physics Laser Driven Simulation

The **LaserSlab** simulation provides an example of how to run a realistic, laser driven, simulation of an HEDP experiment. It exercises a wide range of HEDP physics including:

- 3T unsplit hydrodynamics in  $R - z$  geometry
- Electron thermal conduction with Spitzer conductivities
- Ion/electron equilibration
- Radiation Diffusion with tabulated opacity
- Laser Ray Tracing
- Tabulated Equation of State

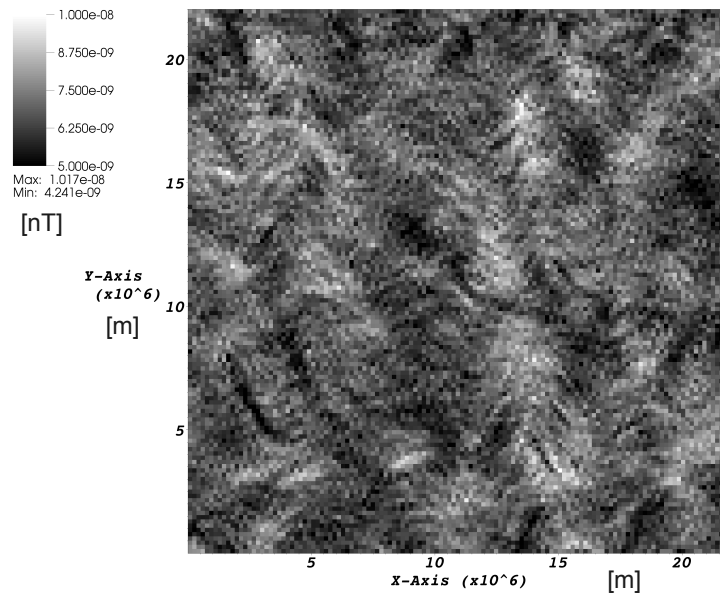


Figure 35.79: The magnetic field  $y$ -component for a two-dimensional ion beam at time  $t = 77.85 \text{ s} \approx 46 \Omega_i^{-1}$ .

This simulation has no analytic solutions associated with it. Rather, it is designed to demonstrate the HEDP capabilities that are available in FLASH.

The simulation models a single laser beam illuminating a solid Aluminum disc in  $R - z$  geometry. The laser is focused on the  $z$ -axis and enters the domain at a 45 degree angle. The simulation contains two materials (or “species” in FLASH terminology). These are called **CHAM** (short for chamber) and **TARG** (short for target). The **TARG** species models the Aluminum disc. The laser travels through a low density Helium region before reaching the Aluminum. The properties of the **CHAM** species are set to model the low density Helium. A schematic showing the initial conditions is shown in [Figure 35.80](#). The laser rays enter just above the lower-right corner of the domain and travel until they are absorbed or reflected. The beam intensity is not uniform across the beam cross section. Rather, it has a super-Gaussian intensity profile with a 10 micron spot radius.

The following setup line is used for this simulation:

```
./setup -auto LaserSlab -2d +cylindrical -nxb=16 -nyb=16 +hdf5typeio \
    species=cham,targ +mtmmmt +laser +uhd3t +mgd mgd_meshgroups=6 \
    -parfile=example.par
```

where:

- **-2d** and **+cylindrical** tells FLASH that this is a 2D cylindrical simulation
- **-nxb=16** and **-nyb=16** sets the AMR block size to 16 by 16
- **+hdf5typeio** enables a newer version of IO that writes to HDF5 files. **HDF5TypeIO** is faster than other IO implementations in FLASH and is also more user friendly when generating figures using external tools, such as VisIt.
- **species=cham,targ** Creates two species named CHAM and TARG using the setup argument *instead of* the **SPECIES** keyword in the **Simulation Config** file. This is required for using the Multispecies opacity unit and is more user friendly since it allows many options to be specified in the runtime parameters file. See [Section 11.4](#) for more information.



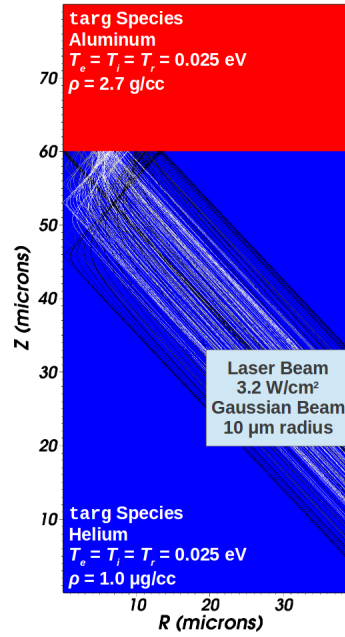


Figure 35.80: Schematic showing the initial conditions in the LasSlab simulation.

- `+mtmmmt` Activates the Multi-Temperature, Multi-Material, Multi-Type EOS in FLASH. This EOS lets users select a different EOS model (such as gamma-law or tabulated) for each species in the simulation through the runtime parameters file. (see [Section 17.4.4](#))
- `+laser` Activates laser ray tracing (see [Section 18.4](#)).
- `+mgd` and `mgd_meshgroups=6` activates multigroup radiation diffusion and tells FLASH that there will be *maximum* of six groups per mesh (see [Chapter 25](#))
- `+uhd3t` Activates the 3T unsplit hydro solver (see [Section 15.1.5](#))
- `-parfile=example.par` Chooses a parameter file that is different than the default

The remainder of this section will describe how each physics unit functions and how the runtime parameters are chosen for this simulation. Images of the results will also be shown at the end of this section.

### 35.7.5.1 Initial Conditions and Material Properties

The section “Initial Conditions” section in the runtime parameter file, “example.par”, sets parameters which define the initial state of the fluid and also set the properties of the materials. The first runtime parameters in this section are:

```
sim_targetRadius = 200.0e-04
sim_targetHeight = 20.0e-04
sim_vacuumHeight = 60.0e-04
```

These options define the space occupied by the target material (the `targ` species, Aluminum for this simulation). The Aluminum disc extends from:

$$0 < R < \text{sim\_targetRadius} \text{ and} \quad (35.59)$$

$$\text{sim\_vacuumHeight} < z < \text{sim\_vacuumHeight} + \text{sim\_targetHeight}, \quad (35.60)$$

where all parameters are in centimeters.

The next section sets the initial fluid state in the target region and also sets the properties of the `targ` species to match Aluminum. The relevant options are:

```
# Target material defaults set for Aluminum at room temperature:
sim_rhoTarg = 2.7                # Set target material density (solid Al)
sim_teleTarg = 290.11375        # Set initial electron temperature
sim_tionTarg = 290.11375        # Set initial ion temperature
sim_tradTarg = 290.11375        # Set initial radiation temperature
ms_targA = 26.9815386           # Set average atomic mass
ms_targZ = 13.0                 # Set average atomic number
ms_targZMin = 0.02              # Set minimum allowed ionization level
eos_targEosType = "eos_tab"      # Set EOS model to tabulated EOS
eos_targSubType = "ionmix4"     # Set EOS table file to IONMIX 4 format
eos_targTableFile = "al-imx-003.cn4" # Set tabulated EOS file name
```

The initial density is chosen to match solid Aluminum. The initial ion, electron, and radiation temperatures are set to room temperature (290 K, 0.025 eV). The runtime parameters `ms_targA` and `ms_targZ` set the average atomic mass and atomic number of the `targ` species. Note that in the past, this information was specified by modifying the `Simulation_initSpecies` subroutine. This information can now be specified using runtime parameters since the `species=cham,targ` setup variable was used (see [Section 11.4](#) for more information). The parameter `ms_targZMin` sets the minimum ionization level for `targ`. Without this parameter, the initial ionization would be nearly zero and the slab would be subcritical with respect to the laser light. The laser rays would penetrate deeply into the slab and could possibly travel through it with no heating. This can be avoided by putting a lower limit on the ionization level. The final three parameters control the behavior of the MTMMMT EOS with respect to `targ`. The option `eos_targEosType` sets the EOS model to use for the Aluminum. In this case, we have chosen to use a tabulated EOS which is in the IONMIX4 format. Finally, we specify the name of the file which contains the EOS data. The file “al-imx-003.cn4” can be found in the LaserSlab simulation directory in the source tree. The IONMIX4 format, described in [Section 23.5.6](#), is not human readable. However, the file “al-imx-003.imx.gz” contains a human readable representation of this EOS and opacity data.

A similar set of runtime parameter options exist for the `cham` species:

```
# Chamber material defaults set for Helium at pressure 1.6 mbar:
sim_rhoCham = 1.0e-05
sim_teleCham = 290.11375
sim_tionCham = 290.11375
sim_tradCham = 290.11375
ms_chamA = 4.002602
ms_chamZ = 2.0
eos_chamEosType = "eos_tab"
eos_chamSubType = "ionmix4"
eos_chamTableFile = "he-imx-005.cn4"
```

The initial Helium density, `sim_rhoCham` is chosen to be fairly low. This ensures that the Helium remains relatively transparent to the laser light (see [Section 18.4.2](#) for a description of how the laser energy absorption coefficient is computed).

For both the `targ` and `cham` species, the initial state of the fluid is defined using a density with three temperatures. The initial EOS mode is set using the `eosModeInit` runtime parameter. It is set to “dens\_temp\_gather”. This tells FLASH that the initial state will be specified using the density and the temperature.

### 35.7.5.2 Multigroup Radiation Diffusion Parameters

This simulation uses multigroup radiation diffusion (MGD) to model the radiation field. This involves setting parameters which control the MGD package itself and setting parameters which control how opacities for the two species are computed. The MGD settings from the parameter file are:

```

rt_useMGD          = .true.
rt_mgdNumGroups    = 6
rt_mgdBounds_1     = 1.0e-01
rt_mgdBounds_2     = 1.0e+00
rt_mgdBounds_3     = 1.0e+01
rt_mgdBounds_4     = 1.0e+02
rt_mgdBounds_5     = 1.0e+03
rt_mgdBounds_6     = 1.0e+04
rt_mgdBounds_7     = 1.0e+05
rt_mgdFlMode       = "fl_harmonic"
rt_mgdFlCoef       = 1.0

rt_mgdXlBoundaryType = "reflecting"
rt_mgdXrBoundaryType = "vacuum"
rt_mgdYlBoundaryType = "vacuum"
rt_mgdYrBoundaryType = "reflecting"
rt_mgdZlBoundaryType = "reflecting"
rt_mgdZrBoundaryType = "reflecting"

```

The first option turns on MGD. The second parameter sets the number of radiation energy groups in the simulation, six in this case. Note that this number must be less than or equal to the value of the `mgd_meshgroups` setup variable. This restriction can be relaxed through the use of mesh replication (see [Section 25.1.2](#)). The runtime parameters beginning with `rt_mgdBounds_` define the group boundaries for each energy group in electron-volts. The parameters `rt_mgdFlMode` and `rt_mgdFlCoef` set the flux limiter coefficient (see [Section 19.1.3](#) and [Section 25.1](#) for more information). The final six parameters set the boundary conditions for all groups.

The next set of parameters tell FLASH how to compute opacities for each material. In general, the total cell opacity is a number density weighted average of the opacity of each species in the cell. The parameters which control the behavior of the opacity unit for this simulation are:

```

useOpacity        = .true.

### SET CHAMBER (HELIUM) OPACITY OPTIONS ###
op_chamAbsorb     = "op_tabpa"
op_chamEmiss      = "op_tabpe"
op_chamTrans      = "op_tabro"
op_chamFileType   = "ionmix4"
op_chamFileName   = "he-imx-005.cn4"

### SET TARGET (ALUMINUM) OPACITY OPTIONS ###
op_targAbsorb     = "op_tabpa"
op_targEmiss      = "op_tabpe"
op_targTrans      = "op_tabro"
op_targFileType   = "ionmix4"
op_targFileName   = "al-imx-003.cn4"

```

The first parameter tells FLASH that opacities will be computed. The next five parameters control how opacities are computed for the `cham` species (Helium) and the final five control the `targ` species (Aluminum). These options tell FLASH to use the IONMIX4 formatted opacity file “he-imx-005.cn4” for Helium and “al-imx-003.cn4” for Aluminum. [Section 23.5.5](#) provides a detailed description of how these parameters are used. Note that the number of groups and group structure in these opacity files must be consistent with the `rt_mgdNumGroups` and `rt_mgdBounds` parameters that were described above.

### 35.7.5.3 Laser Parameters

This section describes the runtime parameters which control the behavior of the laser in FLASH. Modeling laser energy deposition using ray tracing is fairly complicated and requires large numbers of input parameters. It is highly recommended that you read through the section describing the behavior of the laser model in FLASH to learn more about how the ray tracing algorithms actually work (see [Section 18.4](#)).

This simulation uses a single laser beam which travels at an angle of 45 degrees from the  $z$ -axis. The “Laser Parameters” section of the runtime parameters file contains all of the options relevant to defining the laser beam. The first set of parameters are:

```
useEnergyDeposition = .true. # Turn on laser energy deposition
ed_maxRayCount      = 10000 # Set maximum number of rays/cycle/proc to 2000
ed_gradOrder        = 2     # Turn on laser ray refraction
```

The first parameter activates the laser model.

The `ed_maxRayCount` parameter sets the maximum number of rays which can be stored on a given process at once. It tells FLASH how much space to set aside for storing rays. In the case of a single process simulation, `ed_maxRayCount` must be less than the total number of rays launched on each cycle. The total number of rays launched is set by the `ed_numRays_###` parameters, described below. Finally, the parameter `ed_gradOrder` controls how the electron number density and temperature are interpolated within a cell. A value of two specifies linear interpolation.

The next set of parameters are:

```
ed_laser3Din2D      = .true.
ed_laser3Din2DwedgeAngle = 0.1
```

Setting `ed_laser3Din2D` to `.true.` activates the 3D-in-2D ray trace algorithm. This means that in this simulation, the beams are defined in 3D geometry and the laser rays are traced in 3D. The total energy deposited is then projected back on the  $R - z$  plane. This can significantly improve the accuracy of the ray trace when compared to 2D-in-2D ray-tracing. The exception to this is the case of a single beam centered on the  $z$ -axis and traveling along the  $z$  axis. In this case, there is no difference between 3D-in-2D and 2D-in-2D ray tracing algorithms. In this example, the rays travel at a 45 degree angle. Thus, the 3D-in-2D ray-trace is needed. When using the 3D-in-2D ray trace, `ed_laser3Din2DwedgeAngle` sets the wedge angle. It should be set to a number between 0.1 and 1.0 degrees. See [Section 18.4.8](#).

The next set of parameters are:

```
ed_useLaserIO      = .true.
ed_laserIOMaxNumPositions = 10000
ed_laserIONumRays   = 128
```

These parameters make use of the LaserIO package in FLASH for visualizing laser rays (see [Section 18.4.11.4](#)). These options tell FLASH to write out the trajectory of 128 rays to the plot files. The runtime parameter `plotFileIntervalStep` is set to 100 which tells FLASH to write plot files every 10 cycles. Later in this section, instructions and examples for plotting the laser ray data will be presented.

Next, a single laser pulse is defined. Each laser pulse (in this case, there is only one) represents a specific time history of the power and multiple pulses can be defined. The `LaserSlab` simulation has only a single laser beam which uses laser pulse number 1. The relevant runtime parameter options are:

```
# Define Pulse 1:
ed_numberOfSections_1 = 4
ed_time_1_1 = 0.0
ed_time_1_2 = 0.1e-09
ed_time_1_3 = 1.0e-09
ed_time_1_4 = 1.1e-09

ed_power_1_1 = 0.0
ed_power_1_2 = 1.0e+09
ed_power_1_3 = 1.0e+09
ed_power_1_4 = 0.0
```

The laser pulse is defined as a piecewise linear function. The parameter `ed_numSections_1` is set to four, meaning the laser power is defined using four different time/power points. The four time and power pairs are defined using the `ed_time_1_#` and `ed_power_1_#` parameters where the times are in seconds and the powers are specified in Watts. In this case a square pulse is defined with a gradual rise, termination over 0.1 ns. The peak power is  $10^9$  W and the pulse lasts for a total of 1.1 ns.

The next set of parameters define the laser beam itself. The trailing `_1` in the parameter names indicate that these parameters are associated with the first beam (in this case, the only beam):

```
ed_lensX_1           = 1000.0e-04
ed_lensY_1           = 0.0e-04
ed_lensZ_1           = -1000.0e-04
ed_lensSemiAxisMajor_1 = 10.0e-04
ed_targetX_1         = 0.0e-04
ed_targetY_1         = 0.0e-04
ed_targetZ_1         = 60.0e-04
ed_targetSemiAxisMajor_1 = 10.0e-04
ed_targetSemiAxisMinor_1 = 10.0e-04
ed_pulseNumber_1     = 1
ed_wavelength_1      = 1.053
ed_crossSectionFunctionType_1 = "gaussian2D"
ed_gaussianExponent_1 = 4.0
ed_gaussianRadiusMajor_1 = 7.5e-04
ed_gaussianRadiusMinor_1 = 7.5e-04
ed_numberOfRays_1    = 4096
ed_gridType_1        = "radial2D"
ed_gridnRadialTics_1 = 64
ed_semiAxisMajorTorsionAngle_1 = 0.0
ed_semiAxisMajorTorsionAxis_1 = "x"
```

The parameters `ed_targetX_1`, `ed_targetY_1`, and `ed_targetZ_1` define the coordinate of the center of the laser focal spot. Notice that even though this is a 2D simulation, three coordinates are specified. That is because this while most of the FLASH solvers will operate in 2D,  $R-z$  geometry, the laser ray trace operates in 3D Cartesian geometry. For specifying the laser focal spot center, “X” refers to the  $R$  direction in the FLASH simulation (which is also called the “X” direction for many FLASH runtime parameters, regardless of the geometry). The `ed_targetZ_1` parameter is referring to the  $z$  direction in the  $R-z$  simulation. This is actually the “Y” direction for other FLASH runtime parameters. Finally, the “Z” direction for the 3D-in-2D ray trace parameters is the direction pointing out of the computer screen. This same naming convention is used for the parameters `ed_lensX_1`, `ed_lensY_1`, and `ed_lensZ_1`. These parameters specify the center of the lens. On each time step, all rays are traced from a location on the lens to the focal spot. The lens must be defined so that it exists entirely *outside* of the domain. The parameters `ed_targetSemiAxisMajor_1` and `ed_targetSemiAxisMinor_1` set the radius of the laser spot. The laser beam can have an elliptical shape. In this case the two parameters will have different values, but usually, a circular beam is desired. In this case, the beam will have a circular shape and a 10 micron radius. The parameter `ed_lensSemiAxisMajor_1` sets the radius of the focal spot. The laser beam is associated with pulse number 1 (the only pulse in this case) using the parameter `ed_pulseNumber_1`. The parameter `ed_wavelength_1` specifies the wavelength, in microns, of the laser light. The `ed_numberOfRays_1` parameter tells FLASH to launch 4096 rays for the first beam every time the `EnergyDeposition` subroutine is called. When using unsplit hydrodynamics (as in this case), this routine is called a single time per cycle. When using split hydrodynamics, it is called twice.

Taken together, the parameters:

- `ed_crossSectionFunctionType_1`,
- `ed_gaussianExponent_1`,
- `ed_gaussianRadiusMajor_1`, and
- `ed_gaussianRadiusMinor_1`

define the spatial variation of the intensity across the beam. When `ed_crossSectionFunctionType_1` is set to “gaussian2D”, the intensity profile is a supergaussian described by:

$$I(r) = I_0 \exp \left\{ - \left[ \left( \frac{x}{R_x} \right)^2 + \left( \frac{y}{R_y} \right)^2 \right]^\gamma \right\} \quad (35.61)$$

where:

- $R_x = \text{ed\_gaussianRadiusMajor\_1}$ ,
- $R_y = \text{ed\_gaussianRadiusMinor\_1}$ , and
- $\gamma = \text{ed\_gaussianExponent\_1}$ .

In our case, we’ve set up a beam with a super-Gaussian profile ( $\gamma = 4$ ) and an e-folding length of 7.5 microns. Since  $R_x = R_y$  we have a circular beam and not an elliptical beam. The user does not directly specify  $I_0$ . Rather, the user specifies the total beam power (as described above). This then sets  $I_0$  through the integral relation:

$$P = 2\pi \int_0^R dr r I(r), \quad (35.62)$$

The parameters `ed_semiAxisMajorTorsionAngle_1` and `ed_semiAxisMajorTorsionAxis_1` only need to change for non-circular beams.

Finally, the user must specify how rays will be spatially distributed across the beam. The `ed_numberOfRays_1` parameter sets the total number of rays to launch on each time step for each beam. Now we have to decide how to distribute those rays across the cross-section of the laser beam. The parameter `ed_gridType_1` is set to “radial2D”. This means the rays will be laid out on the circular cross-section of the beam with some radial and angular spacing. The area of the beam is divided into regions of equal radius and angle. The parameter `ed_gridnRadialTics_1` is set to 64 meaning that there are 64 radial slices in the beam cross section. Therefore, there are:  $4096/64 = 64$  angular slices.

### 35.7.5.4 Examining LaserSlab Simulation Output

This section describes how the output can be visualized using VisIt and through other means. LaserIO code in FLASH can be used to visualize ray paths (see [Section 18.4.11.4](#) for more information). The LaserIO unit can only be used with parallel HDF5 IO implementations. The `+hdf5typeio` setup option is consistent with this. The runtime parameters:

```
ed_useLaserIO           = .true.
ed_laserIOMaxNumPositions = 10000
ed_laserIONumRays       = 128
```

tell FLASH to use LaserIO and to write out the trajectory of 128 rays to the plot files so they can be visualized in VisIt. Notice that this is a small subset of the total number of rays that are actually launched. Writing out all of the rays is inadvisable since it will have numerous negative effects on performance when large number of rays are used in the simulation. In this case, plot files are written every 0.1 ns and the simulation is run for 2 ns cycles (as indicated by the parameter `tmax`). Thus, the following plot files are generated:

```
lasslab_hdf5_plt_cnt_0000
lasslab_hdf5_plt_cnt_0001
...
lasslab_hdf5_plt_cnt_0199
lasslab_hdf5_plt_cnt_0200
```

These HDF5 files contain the ray trajectories for each of the plot time steps (cycles 1, 11, ... 191, and 201 in this case). Unfortunately, at this time, the FLASH VisIt plugin does not natively support ray visualization. The ray data must be extracted from the plot files in a form that VisIt does understand. The

`extract_rays.py` script, found in the `tools/scripts` directory, performs this operation. Note, however, that `extract_rays.py` requires the NumPy and PyTables python packages to operate.

After this example LaserSlab simulation is run, the `extract_rays.py` script can be used as shown:

```
>>> extract_rays.py lasslab_hdf5_plt_cnt_*
Processing file: lasslab_hdf5_plt_cnt_0000
Processing file: lasslab_hdf5_plt_cnt_0001
...
Processing file: lasslab_hdf5_plt_cnt_0198
Processing file: lasslab_hdf5_plt_cnt_0199
Processing file: lasslab_hdf5_plt_cnt_0200
No ray data in "lasslab_hdf5_plt_cnt_0200" skipping...
```

Note that a warning message was printed for the last plot file. This occurs because there is no ray data for a plot file written on the last cycle of a simulation. This is a limitation of the LaserIO package. The script generates files in the VTK format named:

```
lasslab_las_0000.vtk
lasslab_las_0001.vtk
...
lasslab_las_0198.vtk
lasslab_las_0199.vtk
```

These files contain meshes which define the ray trajectories in a format understood by VisIt.

The ray data can be plotted on top of the computational mesh. In VisIt, load the `lasslab_las_*.vtk` database. Add the “RayPowerWatts” Pseudocolor plot. The “RayPowerWatts” plot colors the trajectories based on the power of each ray in units of Watts. Thus, the power of any one ray at any given time should be less than the beam powers specified in the runtime parameters file.

An example showing the ray powers and trajectories is shown in [Figure 35.80](#). The trajectory of 128 laser rays has been drawn. The rays are colored based on their powers.

### Computing the Number Density and Average Ionization

For a simulation using the MTMMMT EOS, the electron number density itself is not directly represented in the solution vector. Rather, for mostly historical reasons, FLASH only keeps track of the variables `YE` and `SUMY`, which are defined as:

$$YE = \frac{\bar{Z}}{\bar{A}}, \quad SUMY = \frac{1}{\bar{A}}$$

where  $\bar{Z}$  is the average ionization level and  $\bar{A}$  is the average atomic mass of the cell. Thus, one can compute the average ionization and electron number density from these quantities. For example:

$$\begin{aligned} \bar{A} &= \frac{1}{SUMY}, \\ \bar{Z} &= \frac{YE}{SUMY}, \\ n_{\text{ion}} &= N_A \frac{\rho}{\bar{A}} = SUMY N_A \rho, \\ n_{\text{ele}} &= N_A \bar{Z} \frac{\rho}{\bar{A}} = YE N_A \rho \end{aligned}$$

Where  $N_A$  is Avogadro's number,  $n_{\text{ion}}$  is the ion number density,  $n_{\text{ele}}$  is the electron number density, and  $\rho$  is the mass density (`DENS` in FLASH). These quantities can be plotted in VisIt using expressions.



Another useful tool in visualizing the behavior of the laser is the FLASH variable `DEPO`. This variable stores the laser energy deposited in a cell on a particular time step per unit mass. The division by mass removes the geometric factor present in the ray power. The amount of laser energy entering the domain and the amount of laser energy exiting the domain can be used to compute the fraction of laser energy that is deposited. The energy entering and exiting the domain is shown for each cycle and integrated over the entire simulation in the file “`lasslab/LaserEnergyProfile.dat`”. This file is produced whenever the laser ray tracing package is active in FLASH.

### 35.7.6 Laser Driven Simulation with Thomson Scattering Diagnostics

The `LaserSlab` simulation described previously can be configured to include the `ThomsonScattering` diagnostic unit. We describe this configuration here to serve as a demonstration for using that unit. See [Chapter 30](#) for more information on the diagnostic code.

As in the previously described `LaserSlab` setup, The simulation models a single laser beam illuminating a solid Aluminum disc in  $R - z$  geometry. The laser is focused on the  $z$ -axis and enters the domain at a 45 degree angle. The simulation contains two materials (or “species” in FLASH terminology). These are called `CHAM` (short for chamber) and `TARG` (short for target). The `TARG` species models the Aluminum disc. The laser travels through a low density Helium region before reaching the Aluminum. The properties of the `CHAM` species are set to model the low density Helium. A schematic showing the initial conditions is shown in [Figure 35.80](#). The laser rays enter just above the lower-right corner of the domain and travel until they are absorbed or reflected. The beam intensity is not uniform across the beam cross section. Rather, it has a super-Gaussian intensity profile with a 10 micron spot radius.

The following setup line is used for this simulation:

```
./setup -auto LaserSlab -2d -nxb=16 -nyb=16 +hdf5typeio \
  species=cham,targ +mtmmmt +laser +uhd3t +mgd mgd_meshgroups=6 \
  ed_maxBeams=2 ed_maxPulseSections=4 ed_maxPulses=2 \
  thsc_maxBeams=1 thsc_maxPulseSections=4 thsc_maxPulses=1 thsc_maxDetectors=1 \
  -parfile=example_thsc.par ThscDemo=True
```

Here the setup variable `ThscDemo` is used to turn on the optional `ThomsonScattering` configuration of the `LaserSlab` setup. This variable is specific to the `LaserSlab` simulation; see the end of this simulation’s `Config` file to see how this is done for `LaserSlab` and use these or similar directives to enable `ThomsonScattering` for other setups.

Additionally,

- some setup variables beginning with `thsc_` are used to demonstrate how to set some limits for parameters related to `ThomsonScattering` that will be hardwired into the executable;
- some setup variables beginning with `ed_` are used to demonstrate how to set some limits for parameters related to `EnergyDeposition` that will be hardwired into the executable;
- `-parfile=example_ths.par` chooses a parameter file that has additional settings for runtime parameters that control `ThomsonScattering`;
- other arguments are as for the regular `LaserSlab` setup and have been described previously.

### 35.7.7 Z-pinch Simulation

The `ZPinch` simulation provides an example of how to run a realistic, current-driven, simulation of a Z-pinch experiment. It exercises a wide range of HEDP physics including:

- 3T unsplit magnetohydrodynamics in  $R - z$  geometry
- Current drive from the `Circuit` unit
- Implicit magnetic resistive diffusion with `DaviesWen` resistivities



- Electron and ion thermal conduction with JiHeld conductivities
- Ion/electron heat exchange
- Radiation Diffusion with tabulated opacity
- Tabulated Equation of State

This simulation has no analytic solutions associated with it. Rather, it is designed to demonstrate some of the HEDP capabilities that are available in FLASH, and serve as a template for users to simulate their own Z-pinch problems.

The following setup line is used for this simulation:

```
./setup -auto magnetoHD/ZPinch -1d +cylindrical +ug +usm3t +mtmmmt \
    species=fill,line,vacu +mgd mgd_meshgroups=6 +hdf5typeio
```

The included `flash.par` file contains many runtime parameters for customization. Below is a subset of the parameters used in ZPinch:

```
sim_rhoType = 0 # 0: 0 Gaussians
```

```
#### DD Fill
```

```
sim_fill_dens = 30.e-03
sim_fill_minDens = 1.e-05 # used in rhoType 11
sim_fill_ctr = 0. # Gaussian center
sim_fill_sigma = 1.0 # rho ~ exp(-r^2/(2*sigma^2))
sim_fill_tele = 1160452.21 # 100 eV (preheated)
sim_fill_tion = 1160452.21 # 100 eV
sim_fill_trad = 1160452.21 # 100 eV
sim_fill_maxTemp = 1.e12 # default, effectively no limit
sim_fill_minTemp = 1.e-12 # effectively no limit
ms_fillA = 2.014
ms_fillZ = 1.0
eos_fillEosType = "eos_tab"
eos_fillSubType = "ionmix4" # set eos table file to ionmix4 format
eos_fillTableFile = "DD-006-imx.cn4" # set tabulated eos file name
```

```
#### Be Liner
```

```
sim_line_dens = 1.848
sim_line_minDens = 1.e-06 # also acts as density floor for vacuum
sim_line_ctr = 1.0 # Gaussian center
sim_line_sigma = 0.8 # rho ~ exp(-r^2/(2*sigma^2))
sim_innerRadius = 0.29 # inner radius of the liner
sim_thickness = 0.058 # thickness of the liner
sim_pert = 0. # pseudo-random density perturbation amplitude
sim_seed = 1 # seed for random number generator (makes sims repeatable)
sim_line_tele = 5802.26105 # 0.5 eV
sim_line_tion = 5802.26105 # 0.5 eV
sim_line_trad = 5802.26105 # 0.5 eV
sim_line_maxTemp = 1.e12 # default, effectively no limit
sim_line_minTemp = 1.e-12 # effectively no limit
ms_lineA = 9.012
ms_lineZ = 4.
eos_lineEosType = "eos_tab"
eos_lineSubType = "ionmix4" # set eos table file to ionmix4 format
eos_lineTableFile = "Be-006-imx.cn4" # set tabulated eos file name
```

```

#### Vacuum (low density Be)
sim_vacu_dens = 1.e-05          # rho <= this is vacuum
sim_vacu_tele = 5802.26105     # 0.5 eV
sim_vacu_tion = 5802.26105     # 0.5 eV
sim_vacu_trad = 5802.26105     # 0.5 eV
sim_vacu_maxTemp = 5802.26105  # 0.5 eV
sim_vacu_minTemp = 5802.26105  # 0.5 eV
ms_vacuA = 9.012
ms_vacuZ = 4.
eos_vacuEosType = "eos_tab"
eos_vacuSubType = "ionmix4"      # set eos table file to ionmix4 format
eos_vacuTableFile = "Be-006-imx.cn4" # set tabulated eos file name

#### Current drive
circ_currFile = "current.dat"    # filename for current input (ns vs. MA)
#circ_voltFile = "Voc_example.dat" # filename for voltage input (circuit model) (s vs. V)
#circ_cyllLength = 1.            # used in 1D sims with circuit model

# circuit model restart parameters
#circ_initIload      = 1.404306536824466661E+07      # these init parameters get used when a restart
#circ_initIstack= 1.828569946357272193E+07          # is detected. User should refer to 'circuit.dat'
#circ_initVc         = 3.834045253898037598E+06      # for the appropriate values to use.
#circ_initFlux       = 1.724316746860757865E-01

#### trajectory output settings
sim_trajOutputInterval = 1.e-10      # output interval for trajectory.dat
sim_trajOutputIntervalNearStag = 1.e-11 # output interval near stagnation
sim_rNearStag = 0.145                # fuel radius at which <= defines "near stagnation"

```

Many of the parameters are self-explanatory, or the comments included in the `flash.par` file suffice. The parameter `sim_rhoType` controls which density profile is used for initialization; 0 is the default value and is simply constant densities in each region. Other options are 1 (single Gaussian for fill species), 2 (a double Gaussian for fill and line species, summed), and 11 (a double Gaussian for fill and line species, piecewise). The `_ctr` and `_sigma` parameters are used for Gaussian profile options. For more details about how `sim_rhoType` is used, see the `Simulation_initBlock.F90` subroutine.

The `_minTemp` and `_maxTemp` parameters provide temperature control for each region. These are enforced every timestep in the `Simulation_adjustEvolution.F90` subroutine and are typically most useful for the vacuum region. The `Simulation_adjustEvolution.F90` subroutine also outputs a file named `trajectory.dat`, which contains radial positions and moving-average velocities of the inner and outer liner interfaces. The `sim_trajOutputInterval`, `sim_trajOutputIntervalNearStag`, and `sim_rNearStag` parameters provide control of the output frequency of this trajectory data.

The default `Config` file is set to use the `FileInput` implementation of the `Circuit` unit (see [Section 18.8](#)). Furthermore, the parameter `circ_currFile` defines the name of the provided current input file as `current.dat`. Parameters related to the McBride circuit model implementation are commented out; to use this, the user first needs to change the requested implementation in the `Config` file, resetup, and recompile.

This simulation uses a high constant resistivity for the vacuum and the DaviesWen resistivity coefficients elsewhere. See ?? for details on how to control resistivity and Ohmic heating in a vacuum region.

The default parameters included in FLASH give a 1D Z-pinch that stagnates around 160 ns. The uniform grid settings use 128 cells across the 0.5 cm domain, and `iProcs` defines the number of processors to run on (currently set to 32). In ??, radial profiles of the azimuthal magnetic field are shown for three different simulation times. We see that the target pinches due to  $\mathbf{J} \times \mathbf{B}$  forces in a reasonable fashion and the vacuum region maintains a  $B/r$  profile (zero current).

There is a 2D version of the `flash.par` file included, but it likely requires much higher resolution to obtain reasonable results.

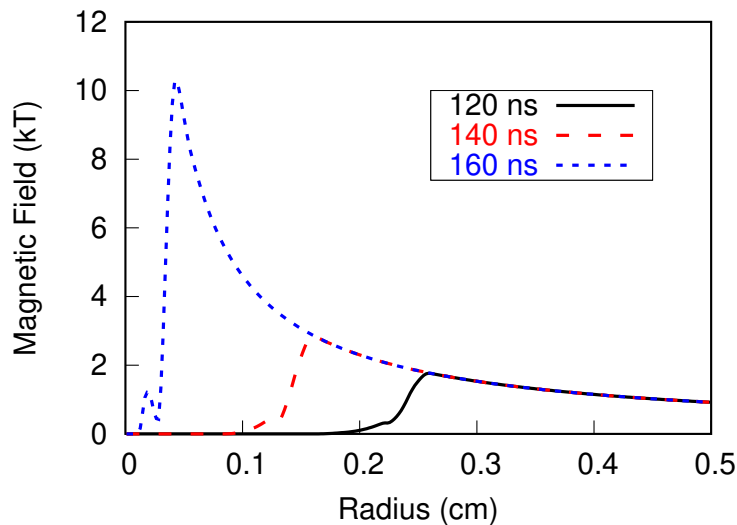


Figure 35.81: Radial profiles of azimuthal magnetic field for the ZPinch example problem.

## 35.8 3T Shock Simulations

FLASH has the ability to simulate plasmas which have separate ion, electron, and radiation temperatures (see [Chapter 14](#)). Usually, simulations which multiple temperatures have several physics models active including:

- Electron thermal conduction
- Ion/Electron equilibration
- Radiation emission, absorption, and diffusion

This section contains a series of simulations which verify FLASH through comparisons with analytic solutions of steady shocks where various assumptions are active. Unfortunately, no single analytic solution contains three distinct temperatures with realistic physical coefficients. Thus, each simulation is performed with a different set of assumptions active. Taken together, they adequately exercise the 3T capabilities in FLASH.

### 35.8.1 Shafranov Shock

The Shafranov problem (Shafranov, 1957) is a one-dimensional problem that provides a good verification for structure of 1D shock waves in a two-temperature plasma with separate ion and electron temperatures. The Shafranov shock solutions takes as input a given upstream condition and shock speed. It then computes the downstream conditions and a shock profile. The solution is fairly sophisticated in that it takes into account electron thermal conduction and ion/electron equilibration. An assumption is made that the electron entropy is continuous across the shock. Thus, immediately downstream of the shock, the ion-temperature is substantially higher than the electron temperature. Far downstream, the temperature equilibrate. Electron conduction creates a preheat region upstream of the shock. A gamma-law EOS is used (typically with  $\gamma = 5/3$ ).

Unfortunately, the Shafranov shock solution can only be simplified to an ODE which must be numerically integrated. The `ShafranovShock` simulation directory includes analytic solutions for several materials including Hydrogen, Helium, and Xenon in the files `plasma_shock.out`, `plasma_shock_Z2.out`, and `plasma_shock_Z54.out` respectively.

Several solutions are compared here for the fully-ionized Helium case with the following initial/boundary conditions:

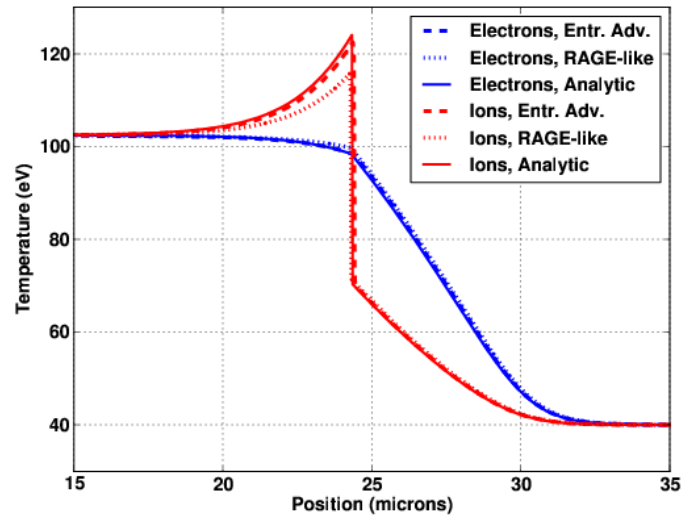


Figure 35.82: Electron and ion temperatures from Shafranov shock simulation

- Upstream Ion/Electron Temperature: 5 eV
- Upstream Density: 0.0018 g/cc
- Upstream Velocity: 0.0 cm/s
- Shock Speed: 1.623415272E+07 cm/s

Figure 35.82, Figure 35.83, and Figure 35.84, shows the electron/ion temperature, density, and velocity at 0.15 ns for three cases:

- Analytic Solution: This is the analytic solution for the steady shock. This solution is used to initialize the FLASH simulations. The simulations are correct to the extent that they are able to maintain this shock profile.
- Entropy Advection Solution: This is a 1D FLASH simulation using the split hydrodynamics solver in “entropy advection” mode as described in Section 15.1.5.1.
- RAGE-like Solution: This is a 1D FLASH simulation using the split hydrodynamics solver in “RAGE-like” mode as described in Section 15.1.5.2.

The figures show that the both the entropy advection and RAGE-like FLASH simulation are able to maintain the correct shock speeds. However, the entropy advection approach closely matches the correct analytic ion temperature profile while the RAGE-like simulation a peak ion temperature that is too low. This is the expected behavior since the RAGE-like mode does not attempt to ensure that electrons are adiabatically compressed by the shock.

This simulation can be setup with the following setup command:

```
# For the Entropy Advection Case:
./setup ShafranovShock -auto -1d +pm4dev +3t -parfile=flash_Z2.par

# For the RAGE-like Case:
./setup ShafranovShock -auto -1d +pm4dev +3t -parfile=ragelike_Z2.par
```

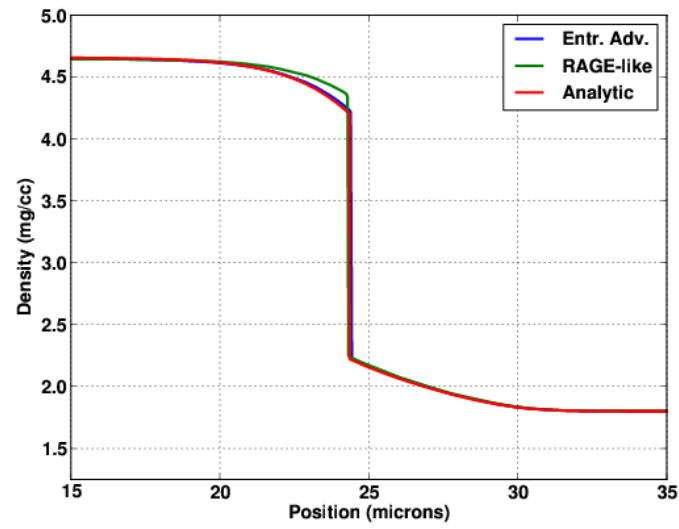


Figure 35.83: Mass density from Shafranov shock simulation

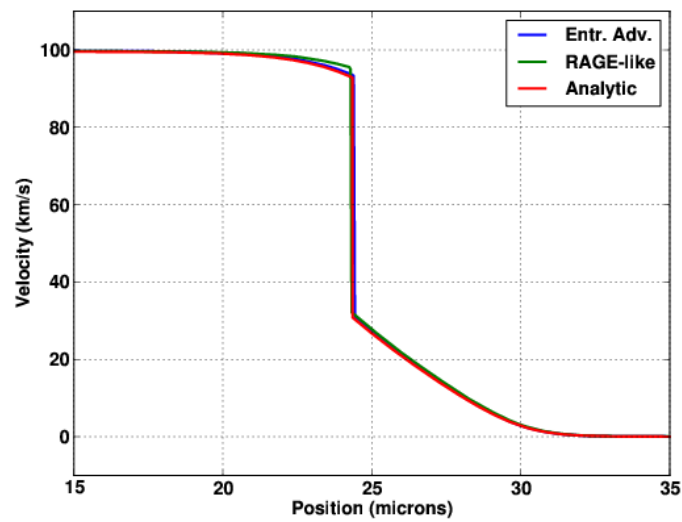


Figure 35.84: Velocity from Shafranov shock simulation

### 35.8.2 Non-Equilibrium Radiative Shock

The non-equilibrium radiative shock solution is an analytical solution to a steady, 1D, radiative shock where  $T_e = T_i$  but  $T_e \neq T_r$ . It is presented in (Lowrie, 2008). A constant opacity is assumed, making this a gray simulation. The “analytic” solution is fairly complex and reduces to an ODE which must be evaluated numerically. This ODE is evaluated for a given set of upstream conditions and a given Mach number (evaluated relative to the upstream sound speed). A gamma-law equation of state is assumed.

The FLASH implementation of this simulation resides in the `GrayDiffRadShock` simulation directory. The simulation can be set up using the following command:

```
./setup -auto GrayDiffRadShock -1d +pm4dev +splitHydro +3t mgd_meshgroups=1
```

The simulation is performed using 3T hydrodynamics with the multigroup radiation diffusion (MGD) unit (see [Chapter 25](#)). A single radiation energy group is used with opacities set to  $\sigma_a = \sigma_e = 423 \text{ cm}^{-1}$  and  $\sigma_t = 788 \text{ cm}^{-1}$  (see [Section 23.5.1](#)). A  $\gamma = 5/3$  gamma-law EOS is used with  $Z = 1$  and  $A = 2$ . The electron and ion temperatures are forced to equilibrium by using the Spitzer `Heatexchange` implementation (see: [Section 18.5.1](#)) where the ion/electron equilibration time has been reduced by a factor of  $10^6$  by setting the `hx_ieTimeCoef` runtime parameter.

The initial conditions are defined by a step function where the jump occurs at  $x = 0$ . The upstream ( $x < 0$ ) and downstream ( $x > 0$ ) conditions are chosen so that the shock remains stationary. The correct jump conditions are to maintain a stationary mach two shock are:

- $\rho_0 = 1.0 \text{ g/cc}$
- $T_0 = 100 \text{ eV}$
- $u_0 = 2.536e + 07 \text{ cm/s}$
- $\rho_1 = 2.286 \text{ g/cc}$
- $T_1 = 2.078 \text{ eV}$
- $u_1 = 1.11e + 07 \text{ cm/s}$

where the subscript 1 represents downstream conditions and the subscript 0 represents upstream conditions. The runtime parameter `sim_P0` representing the ratio of radiation to matter pressure is set to  $10^{-4}$ .

The verification test is successful to the extent that FLASH is able to maintain a stationary shock with the correct steady state spatial profile as shown in Figure 8 of (Lowrie, 2008). This is an excellent verification test in that no special modifications to the FLASH code are needed to perform this test. The simulation is run for 4.25 ns which is enough time for the initial step function profile to reach a steady state solution. [Figure 35.85](#) compares the temperatures in the FLASH simulation to the analytic solution. [Figure 35.86](#) compares density to the analytic solution. Excellent agreement is obtained.

### 35.8.3 Blast Wave with Thermal Conduction

The ReinickeMeyer blast wave solution (Reinicke, 1991) models a blast wave in a single temperature fluid with thermal conduction. The semi-analytic solution reduces to an ODE which must be integrated numerically. [Figure 35.87](#) compares a FLASH simulation to the analytic solution (obtained from [cococubed.asu.edu/code\\_pages/vv.shtml](http://cococubed.asu.edu/code_pages/vv.shtml)) after 0.3242 ns for a particular set of initial conditions. Excellent agreement is obtained with the analytic solution. The image shows that at this time the blast wave is at approximately 0.45 cm and the conduction front is at 0.9 cm. The simulation can be set up using the following command:

```
./setup -auto ReinickeMeyer -1d +pm4dev +spherical -parfile=flash_SPH1D.par
```

The shortcut `+splitHydro` may have to be added to reproduce the presented results.

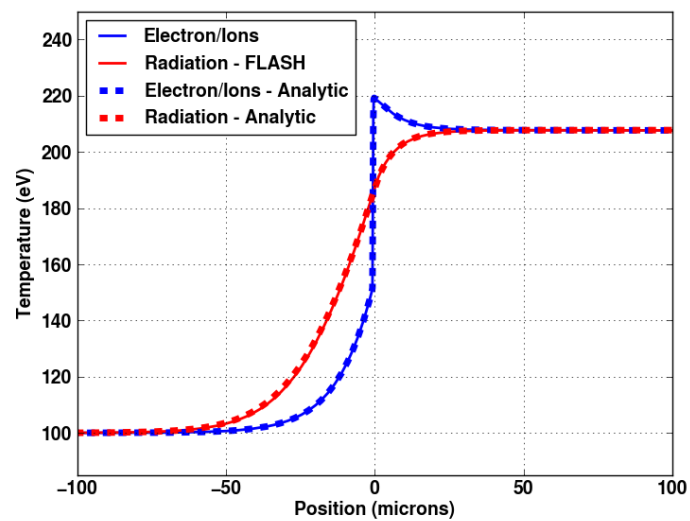


Figure 35.85: Temperatures from non-equilibrium radiative shock simulation. The FLASH results are compared to analytic solution.

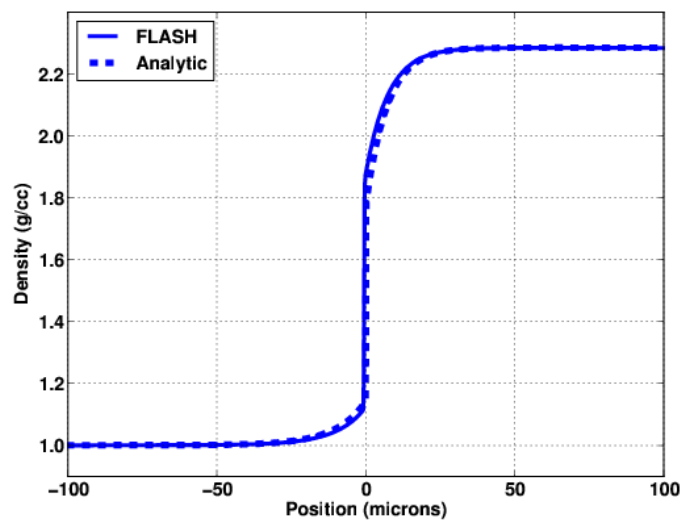


Figure 35.86: Mass density from non-equilibrium radiative shock simulation. The FLASH result is compared to the analytic solution.

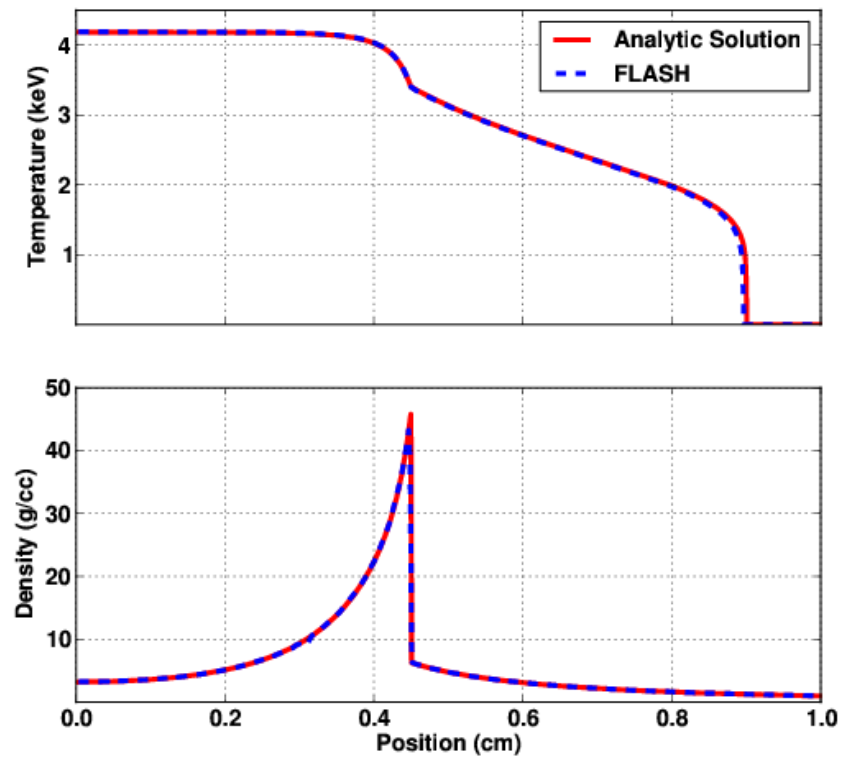


Figure 35.87: Mass density and temperature from ReinickeMeyer blast wave FLASH simulation compared to analytic solution.



## 35.9 Matter+Radiation Simulations

Here are some simulations that use the **RadFLAH** (radiation-fluxlimiter-aware hydro) variant of **unsplit Hydro**. They differ from generic 3T simulations in that there is no physical distinction between the temperatures of electrons and ions; that is, electrons and ions are understood as forming a “matter” component. The “matter” component can have a different temperature from radiation.

### 35.9.1 Radiation-Inhibited Bondi Accretion

This setup can be used as a 1D spherical version of the radiation-inhibited Bondi accretion problem described by Krumholz et al (2007). The simulation can be set up using the following command:

```
./setup radflaHD/BondiAccretion -1d -auto +spherical -nxb=16 +mgd mgd_meshgroups=1 \
    species=h1 ManualSpeciesDirectives=True +parallelio +uhd3tr \
    -without-unit=physics/Hydro/HydroMain/unsplit
```

Note that this setup differs significantly from the one described in Krumholz et al (2007), by using 1D instead of 3D geometry, and by excluding the central region from the domain instead of representing the accreting mass at the origin as a sink (or “star”) particle.

### 35.9.2 Radiation Blast Wave

This setup can be used to reproduce the Radiation Blast Wave problem described in the Castro II paper by Zhang et al (2011). The simulation can be set up using the following command for using the default MGD solver:

```
./setup radflaHD/RadBlastWave -1d -auto +spherical -nxb=16 +mgd mgd_meshgroups=1 \
    species=h1 ManualSpeciesDirectives=True +parallelio +uhd3tr
```

For Case II of Zhang et al (2011), the following variant, using the experimental ExpRelax MGD solver implementation, may give better results:

```
./setup radflaHD/RadBlastWave -1d -auto +spherical -nxb=16 +mgd mgd_meshgroups=1 \
    species=h1 ManualSpeciesDirectives=True +parallelio +uhd3tr RadTransImpl=ExpRelax
```

For both cases, the user has to create an appropriate parameter file; several sample \*.par files can be found in the `Simulation/SimulationMain/radflaHD/RadBlastWave` directory.

### 35.9.3 1D Supernova

A 1D supernova simulation using RadFLAH from which light curves can be generated is included in FLASH4.5. More information will be included here after draft publication of Chatzopoulos and Weide (2017).

Example configuration:

```
./setup radflaHD/SupernovaRad1D -1d -auto +spherical +mgd mgd_meshgroups=1 -nxb=16 \
    -objdir mesa_sn_2T --maxblocks=1000 +pm4dev +uhd3tr +newMpole \
    species=neut,prot,h1,he3,he4,c12,n14,o16,ne20,mg24,si28,s32,ar36,ca40,ti44,cr48,fe52,fe54,ni56 \
    ManualSpeciesDirectives=True \
    +parallelio -without-unit=physics/Eos/EosMain/multiTemp/Multigamma RadTransImpl=MGD \
    -parfile=flash_working.par
```



# Part X

## Tools



Two tools are included in the release of **FLASH** to assist users with analysis of data. The first, **sfocu**, provides a way to compare output files. The second, **fidlr3.0**, provides visualization and analysis tools by using the proprietary IDL package.



## Chapter 36

# VisIt

The developers of FLASH also highly recommend VisIt, a free parallel interactive visualization package provided by Lawrence Livermore National Laboratory (see <https://wci.llnl.gov/codes/visit/>). VisIt runs on Unix and PC platforms, and can handle small desktop-size datasets as well as very large parallel datasets in the terascale range. VisIt provides a native reader to import FLASH2.5 and FLASH3. Version 1.10 and higher natively support FLASH3. For VisIt versions 1.8 or less, FLASH3 support can be obtained by installing a tarball patch available at [http://flash.rochester.edu/site/flashcode/user\\_support/visit/](http://flash.rochester.edu/site/flashcode/user_support/visit/). Full instructions are also available at that site.





## Chapter 37

# 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.

### 37.1 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.

### 37.2 Using **sfocu**

The basic and most common usage is to run the command **sfocu <file1> <file2>**. The option **-t <dist>** allows a distance tolerance in comparing bounding boxes of blocks in two different files to determine which are the same (which have data to compare to one another). You might need to widen your terminal to view the output, since it can be over 80 columns. Sample output follows:

```

A: 2006-04-25/sod_2d_45deg_4lev_ncmpi_chk_0001
B: 2005-12-14/sod_2d_45deg_4lev_ncmpi_chk_0001
Min Error: inf(2|a-b| / max(|a+b|, 1e-99) )
Max Error: sup(2|a-b| / max(|a+b|, 1e-99) )
Abs Error: sup|a-b|
Mag Error: sup|a-b| / max(sup|a|, sup|b|, 1e-99)
Block shapes for both files are: [8,8,1]
Mag-error tolerance: 1e-12
Total leaf blocks compared: 541 (all other blocks are ignored)

```

Var	Bad Blocks	Min Error	Max Error			Abs Error		
			Error	A	B	Error	A	B
dens	502	0	1.098e-11	0.424	0.424	4.661e-12	0.424	0.424
eint	502	0	1.1e-11	1.78	1.78	1.956e-11	1.78	1.78
ener	502	0	8.847e-12	2.21	2.21	1.956e-11	2.21	2.21
gamc	0	0	0	0	0	0	0	0
game	0	0	0	0	0	0	0	0
pres	502	0	1.838e-14	0.302	0.302	1.221e-14	0.982	0.982
temp	502	0	1.1e-11	8.56e-09	8.56e-09	9.41e-20	8.56e-09	8.56e-09
velx	516	0	5.985	5.62e-17	-1.13e-16	2.887e-14	0.657	0.657
vely	516	0	2	1e-89	-4.27e-73	1.814e-14	0.102	0.102
velz	0	0	0	0	0	0	0	0
mfrfc	0	0	0	0	0	0	0	0

Var	Bad Blocks	Mag Error	A			B		
			Sum	Max	Min	Sum	Max	Min
dens	502	4.661e-12	1.36e+04	1	0.125	1.36e+04	1	0.125
eint	502	6.678e-12	7.3e+04	2.93	1.61	7.3e+04	2.93	1.61
ener	502	5.858e-12	8.43e+04	3.34	2	8.43e+04	3.34	2
gamc	0	0	4.85e+04	1.4	1.4	4.85e+04	1.4	1.4
game	0	0	4.85e+04	1.4	1.4	4.85e+04	1.4	1.4
pres	502	1.221e-14	1.13e+04	1	0.1	1.13e+04	1	0.1
temp	502	6.678e-12	0.000351	1.41e-08	7.75e-09	0.000351	1.41e-08	7.75e-09
velx	516	3.45e-14	1.79e+04	0.837	-6.09e-06	1.79e+04	0.837	-6.09e-06
vely	516	2.166e-14	1.79e+04	0.838	-1.96e-06	1.79e+04	0.838	-1.96e-06
velz	0	0	0	0	0	0	0	0
mfrfc	0	0	3.46e+04	1	1	3.46e+04	1	1

FAILURE

“Bad Blocks” is the number of leaf blocks where the data was found to differ between datasets. Four different error measures (min/max/abs/mag) are defined in the output above. In addition, the last six columns report the sum, maximum and minimum of the variables in the two files. Note that the sum is physically meaningless, since it is not volume-weighted. Finally, the last line permits other programs to parse the `sfocu` output easily: when the files are identical, the line will instead read `SUCCESS`.

It is possible for `sfocu` to miss machine-precision variations in the data on certain machines because of compiler or library issues, although this has only been observed on one platform, where the compiler produced code that ignored IEEE rules until the right flag was found.

# Chapter 38

## Drift

### 38.1 Introduction

Drift is a debugging tool added to FLASH to help catch programming mistakes that occur while refactoring code in a way that *should not* change numerical behavior. Historically, simulation checkpoints have been used to verify that results obtained after a code modification have not affected the numerics. But if changes are observed, then the best a developer can do to narrow the bug hunting search space is to look at pairs of checkpoint files from the two different code bases sequentially. The first pair to compare unequal will tell you that somewhere between that checkpoint and its immediate predecessor something in the code changed the numerics. Therefor, the search space can only be narrowed to the limit allow by the checkpointing interval, which in FLASH, without clever calls to IO sprinkled about, is at best once per time cycle.

Drift aims to refine that granularity considerably by allowing comparisons to be made upon every modification to a block's contents. To achieve this, drift intercepts calls to `Grid.releaseBlkPtr`, and inserts into them a step to checksum each of the variables stored on the block. Any checksums that do not match with respect to the last checksums recorded for that block are logged to a text file along with the source file and line number. The developer can then compare two drift logs generated by the different runs using `diff` to find the first log entry that generates unequal checksums, thus telling the developer which call to `Grid.releaseBlkPtr` first witnessed divergent values.

The following are example excerpts from two drift logs. Notice the checksum value has changed for variable `dens` on block 18. This should clue the developer in that the cause of divergent behavior lies somewhere between `Eos_wrapped.F90:249` and `hy_ppm_sweep.F90:533`.

```
inst=2036
step=1
src=Eos_wrapped.F90:249
blk=57
dens E8366F6E49DD1B44
eint 89D635E5F46E4CE4
ener C6ED4F02E60C9E8F
pres 6434628E2D2E24E1
temp DB675D5AFF7D48B8
velx 42546C82E30F08B3
```

```
inst=2100
step=1
src=hy_ppm_sweep.F90:533
blk=18
dens A462F49FFC3112DE
eint 9CD79B2E504C7C7E
ener 4A3E03520C3536B9
velx 8193E8C2691A0725
vely 86C5305CB7DE275E
```

```
inst=2036
step=1
src=Eos_wrapped.F90:249
blk=57
dens E8366F6E49DD1B44
eint 89D635E5F46E4CE4
ener C6ED4F02E60C9E8F
pres 6434628E2D2E24E1
temp DB675D5AFF7D48B8
velx 42546C82E30F08B3
```

```
inst=2100
step=1
src=hy_ppm_sweep.F90:533
blk=18
dens 5E52D67C5E93FFF1
eint 9CD79B2E504C7C7E
ener 4A3E03520C3536B9
velx 8193E8C2691A0725
vely 86C5305CB7DE275E
```

## 38.2 Enabling drift

In FLASH, drift is disabled by default. Enabling drift is done by hand editing the `Flash.h` file generated by the setup process. The directive line `#define DRIFT_ENABLE 0` should be changed to `#define DRIFT_ENABLE 1`. Once this has been changed, a recompilation will be necessary by executing `make`.

With drift enabled, the FLASH executable will generate log files in the same directory it is executed in. These files will be named `drift.<rank>.log`, one for each MPI process.

The following runtime parameters are read by drift to control its behavior:

Parameter	Default	Description
<code>drift_trunc_mantissa</code>	2	The number of least significant mantissa bits to zero out before hashing a floating point value. This can be used to stop numerical noise from altering checksum values.
<code>drift_verbose_inst</code>	0	The instance index at which drift should start logging checksums per call to <code>Grid_releaseBlkPtr</code> . Before this instance is hit, only user calls to <code>Driver_driftUnk</code> will generate log data. A value of zero means never log checksums per block. Instance counting is described below.
<code>drift_tuples</code>	.false.	A boolean switch indicating if drift should write logs in the more human readable "non-tuples" format or the machine friendly "tuples" format that can be read in by the <code>driftDee</code> script found in the <code>tools/</code> directory. Generally the "non-tuples" format is a better choice to use with tools such as <code>diff</code> .

## 38.3 Typical workflow

Drift has two levels of output verbosity, let us refer to them as verbose and not verbose. When in non-verbose mode, drift will only generate output when directly told to do so through the `Driver_driftUnk` API call. This call tells drift to generate a checksum for each `unk` variable over all blocks in the domain and then log those checksums that have changed since the last call to `Driver_driftUnk`. Verbose mode also generates this information and additionally includes the per-block checksums for every call to `Grid_releaseBlkPtr`. Verbose mode can generate *a lot* of log data and so should only be activated when the simulation nears the point at which divergence originates. This is the reason for the `drift_verbose_inst` runtime parameter.

Drift internally maintains an "instance" counter that is incremented with every intercepted call to `Grid_releaseBlkPtr`. This is drift's way of enumerating the program states. When comparing two drift logs, if the first checksum discrepancy occurs at instance number 1349 (arbitrary), then it is clear that somewhere between the 1348'th and 1349'th call to `Grid_releaseBlkPtr` a divergent event occurred.

The suggested workflow once drift is enabled is to first run both simulations with verbose mode off (`drift_verbose_inst=0`). The main `Driver_evolveFlash` implementations have calls to `Driver_driftUnk` between all calls to FLASH unit advancement routines. So the default behavior of drift will generate multiple unk-wide checksums for each variable per timestep. These two drift logs should be compared to find the first entry with a mismatched checksum. Each entry generated by `Driver_driftUnk` will contain an instance range like in the following:

```
step=1
from=Driver_evolveFlash.F90:276
unks inst=1234 to 2345
  dens 9CF3C169A5BB129C
  eint 9573173C3B51CD12
  ener 028A5D0DED1BC399
...
```

The line "unks inst=1349 to 2345" informs us these checksums were generated sometime after the 2345'th call to `Grid_releaseBlkPtr`. Assume this entry is the first such entry to not match checksums with its counterpart. Then we know that somewhere between instance 1234 and 2345 divergence began. So we set `drift_verbose_inst = 1234` in the runtime parameters file of each simulation and then run them both again. Now drift will run up to instance 1234 as before, only printing at calls to `Driver_driftUnk`, but starting with instance 1234 each call to `Grid_releaseBlkPtr` will induce a per block checksum to be logged as well. Now these two drift files can be compared to find the first difference, and hopefully get you on your way to hunting down the cause of the bug.

## 38.4 Caveats and Annoyances

The machinery drift uses to intercept calls to `Grid_releaseBlkPtr` is lacking in sophistication, and as such can put some unwanted constraints on the code base. The technique used is to declare a preprocessor `#define` in `Flash.h` to expand occurrences of `Grid_releaseBlkPtr` to something larger that includes `__FILE__` and `__LINE__`. This is how drift is able to correlate calls to `Grid_releaseBlkPtr` with the originating line of source code. Unfortunately this technique places a very specific restriction on the code once drift is enabled. The trouble comes from the types of source lines that may refer to a subroutine without calling it. The primary offender being `use` statements with `only` clauses listing the module members to import into scope. Because macro expansion is dumb with respect to context, it will expand occurrences of `Grid_releaseBlkPtr` in these `use` statements, turning them into syntactic rubbish. The remedy for this issue is to make sure the line `#include "Flash.h"` comes after all statements involving `Grid_releaseBlkPtr` but not calling it, and before all statements that are calls to `Grid_releaseBlkPtr`. In practice this is quite easy. With only one subroutine per file, there will only be one line like:

```
use Grid\_interface, only: ..., Grid\_releaseBlkPtr, ...
```

and it will come before all calls to `Grid_releaseBlkPtr`, so just move the `#include "Flash.h"` after the `use` statements. The following is an example:

Incorrect	Correct
<pre>#include "Flash.h" subroutine Flash_subroutine()   use Grid_interface, only: Grid_releaseBlkPtr   implicit none   [...]   call Grid_releaseBlkPtr(...)   [...] end subroutine Flash_subroutine</pre>	<pre>subroutine Flash_subroutine()   use Grid_interface, only: Grid_releaseBlkPtr   implicit none   #include "Flash.h"   [...]   call Grid_releaseBlkPtr(...)   [...] end subroutine Flash_subroutine</pre>

If such a solution is not possible because no separation between all `use` and `call` statements exists, then there are two remaining courses of action to get the source file to compile. One, hide these calls to `Grid_releaseBlkPtr` from drift by forcefully disabling the macro expansion. To do so, just add the line `#undef Grid_releaseBlkPtr` after `#include "Flash.h"`. The second option is to carry out the macro expansion by hand. This also requires disabling the macro with the `undef` just mentioned, but then also rewriting each call to `Grid_releaseBlkPtr` just as the preprocessor would. Please consult `Flash.h` to see the text that gets substituted in for `Grid_releaseBlkPtr`.



## Chapter 39

# FLASH IDL Routines (`fidlr3.0`)

`fidlr3.0` is a set of routines to read and plot data files produced by FLASH. The routines are written in the graphical display language IDL (Interactive Data Language) and require the IDL program from Research Systems Inc. (<http://www.rsi.com>). These routines include programs which can be run from the IDL command line to read 1D, 2D, or 3D FLASH datasets, interactively analyze datasets, and interpolate them onto uniform grids.

However, some of these routines are not described in this release of the documentation because they have not been thoroughly tested with FLASH4. A graphical user interface (GUI) to these routines (`xflash3`) is provided, which enables users to read FLASH AMR datasets and make plots of the data. Both plotfiles and checkpoint files, which differ only in the number and numerical precision of the variables stored, are supported.

`fidlr3.0` supports Cartesian, cylindrical, polar and spherical geometries. The proper geometry should be detected by `xflash3` using the geometry attribute in the file header. System requirements for running `fidlr3.0` are: IDL version 5.6 and above, and the HDF5 library. The routines also have limited support for data files written with `netCDF`, although this output format has not been thoroughly tested. The routines are intended to be backwards-compatible with FLASH2, although again extensive testing has not been performed.

### 39.1 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.

#### 39.1.1 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.

### 39.1.2 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
```

## 39.2 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 39.1](#).

```
IDL> xflash3
```

`xflash3` produces colormap plots of FLASH data with optional overlays of velocity vectors, contours, and the block structure. The basic operation of `xflash3` is to specify a single output file (either checkpoint or plotfile) as a prototype for the FLASH simulation. The prototype is probed for the list of variables it contains, and then the remaining plot options become active.

`xflash3` can output to the screen, Postscript, or a PNG image file. If the data is significantly higher resolution than the output device, `xflash3` will sub-sample the image by one or more levels of refinement before plotting.

Once the image is plotted, the *query* (2-d data only) and *1-d slice* buttons will become active. Pressing *query* and then clicking anywhere in the domain will pop up a window containing the values of all the FLASH variables in the cell nearest the cursor. The query function uses the actual FLASH data—not the interpolated/uniformly gridded data generated for the plots. Pressing *1-d slice* and then left-clicking on the plot will produce a 1-d slice vertically through the point. Right-clicking on the domain produces a horizontal slice through the data.

The widget is broken into several sections, with some features initially disabled. Not all options are available in all dimensions, or in this release of FLASH4. These sections are explained below.

### 39.2.1 File Menu

The file to be visualized is composed of the path, the basename (the same base name used in the `flash.par` file) with any file type information appended to it (*e.g.* `'hdf5_chk_'`) and the range of suffixes through which to loop. By default, `xflash3` sets the path to the working directory from which IDL was started. `xflash3` requires a prototype file to work on a dataset. The prototype can be any of the files in the dataset that has the same name structure (*i.e.* everything is the same but the suffix) and contains the same variables.



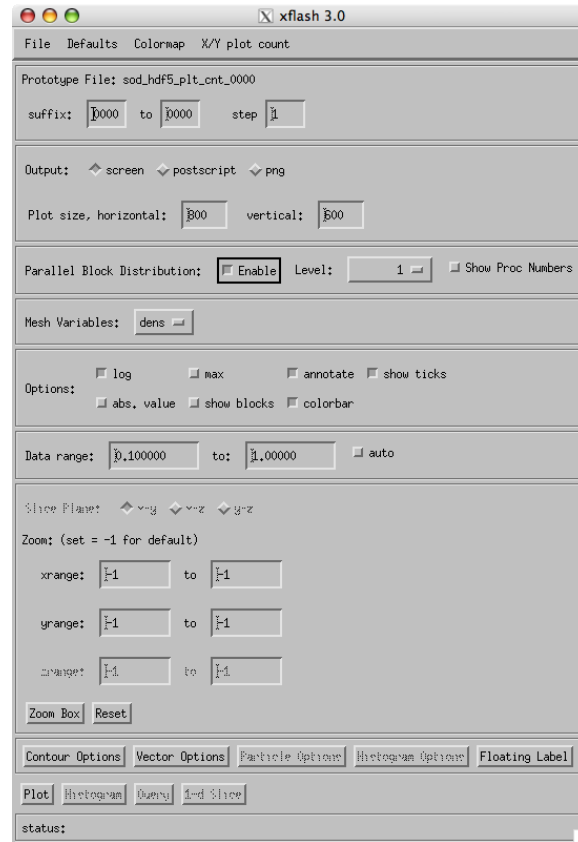


Figure 39.1: The main xflash3 widget.

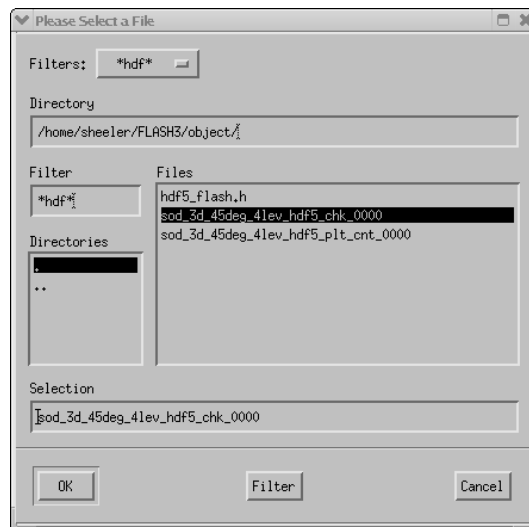
### 39.2.1.1 File/Open prototype...

The *Open prototype...* menu option will bring up the file selection dialog box (see [Figure 39.2](#)). 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 39.3](#)).

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.

## 39.2.2 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.

Figure 39.2: The `xflash3` file selection dialog.

### 39.2.3 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.

### 39.2.4 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.

### 39.2.5 Plotting options available from the GUI

Various options are available on the `xflash3` user interface to change the appearance of the plots.

#### 39.2.5.1 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.

#### 39.2.5.2 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.

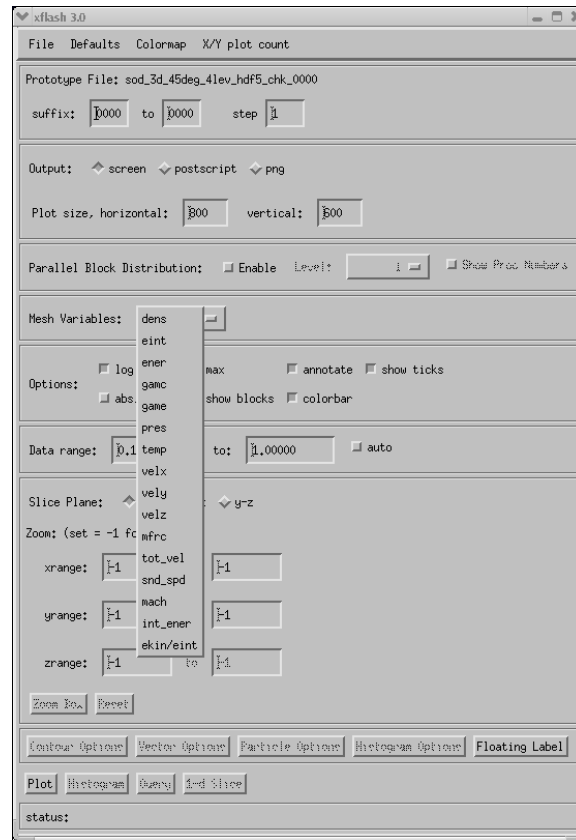


Figure 39.3: The `xflash3` main window after a prototype has been selected, showing the variable list.

### 39.2.5.3 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.

### 39.2.5.4 Mesh Variables

The variables dropbox lists the mesh, or grid variables stored in the ‘unknown names’ record in the data file and any derived variables that `xflash3` knows how to construct from these variables (*e.g.* sound speed). This selection allows you to choose the variable to be plotted. By default, `xflash3` reads all the variables in a file in 1- and 2-d datasets, so switching the variable to plot can be done without re-reading. At present, there is no easy way to add a derived variable. Both the widget routine (`xflash3.pro`) and the plotting backend (`xplot#d_amr.pro`) will need to be told about any new derived variables. Users wishing to add derived variables should look at how the total velocity (`tot_vel`) is computed.

### 39.2.5.5 Options

The options block allows you to toggle various options on/off. Table 39.1 lists the various options available.

Table 39.1: xflash3 options

log	Plot the base-10 log of the variable.
max	When a sequence of files is defined in the file options, plot the maximum of the variable in each cell over all the files.
annotate	Toggle the title and time information on the plot.
show ticks	Show the axis tick marks on the plot.
abs. value	Plot the absolute value of the dataset. This operation is performed before taking the log.
show blocks	Draw the zone boundaries on the plot.
colorbar	Plot the colorbar legend for the data range.

### 39.2.5.6 Data Range

These fields allow you to specify the range of the variable to plot. Data outside of the range will be set to the minimum or maximum values of the colormap. If the *auto* box is checked, the limits will be ignored, and the data will be scaled to the minimum and maximum values of the variable in the dataset.

### 39.2.5.7 Slice Plane

The slice plane group is only active for 3-d datasets. This option allows you to select a plane for plotting in  $(x-y, x-z, y-z)$ .

### 39.2.5.8 Zoom

The zoom options allow you to set the domain limits for the plot. A value of -1 uses the actual limit of the domain. For 3-d plots, only one field will be available in the direction perpendicular to the slice plane. The *zoom box* button puts a box cursor on the plot and allows you to select a region to zoom in on by positioning and resizing the box with the mouse. Use the left mouse button to move the center of the zoom box. Use the middle button to resize the box. Finally, right-click with the mouse when you are satisfied with the zoom box selection. (Note that you must choose the "Plot" button again to see the results of the selected zoom. The *reset* button will reset the zoom limits.

## 39.2.6 Plotting buttons

Several buttons are located at the bottom of the **xflash3** user interface which pop up additional windows. Most of these set additional groups of options, but the actual commands to create plots are located on the bottom row.

### 39.2.6.1 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 39.4](#)). 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.

### 39.2.6.2 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 39.5](#)). 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

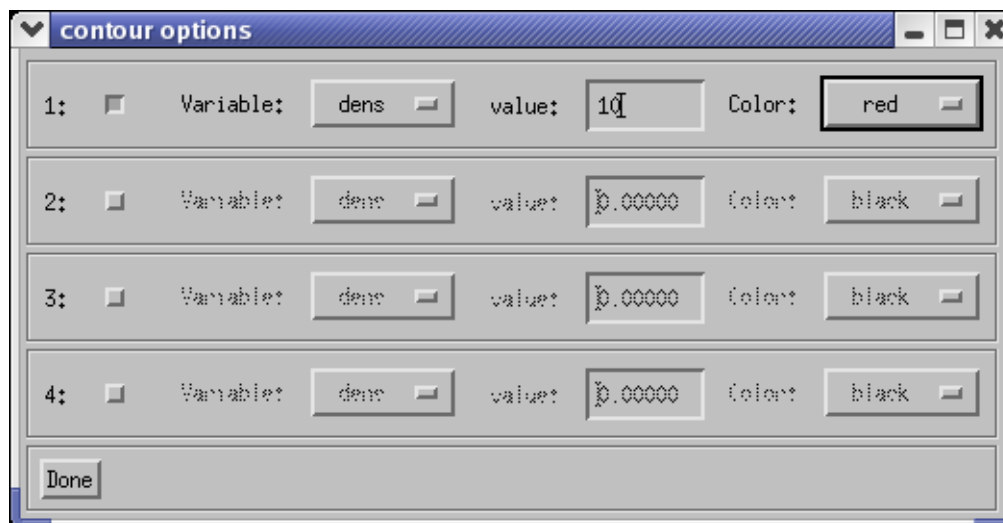


Figure 39.4: The `xflash3` contour option subwidget.

which to scale the vectors, and *minimum vector* and *maximum vector* specify the range of vectors for which to plot. *xskip* and *yskip* allow you to thin out the arrows. This option is available in 2-d only.

### 39.2.6.3 Particle Options Button

This button enables the user to set options for plotting particles. Since particle-handling routines are not present in this release of FLASH4, this option is disabled in `xflash3`.

### 39.2.6.4 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.

### 39.2.6.5 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.

### 39.2.6.6 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.

### 39.2.6.7 Histogram Button

Create a histogram of the data. This option is disabled in this release of FLASH3.

### 39.2.6.8 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 39.6). Use the *Close* button to dismiss the results.

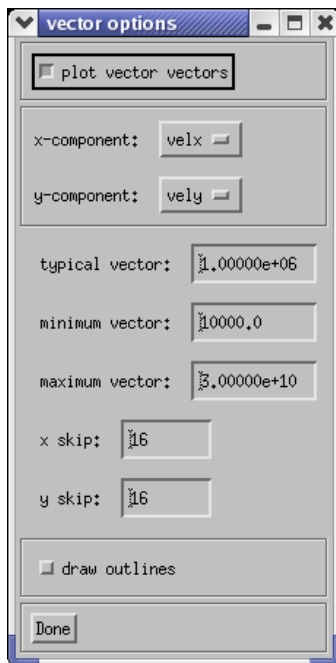


Figure 39.5: The xflash3 velocity option subwidget.

#### 39.2.6.9 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.

### 39.3 Comparing two datasets

From the IDL prompt, a plot showing visual difference can be created with the command `diff3`. Comparisons can be made between any two variables in different files or within the same file. However, this command currently is supported ONLY for two-dimensional datasets. For example, the command:

```
IDL> diff3, '<path_to_file>/flash_hdf5_0001', 'dens', '<path_to_file>/flash_hdf5_0002', 'dens'
```

plots the difference between the 'dens' variable in two different Flash files.

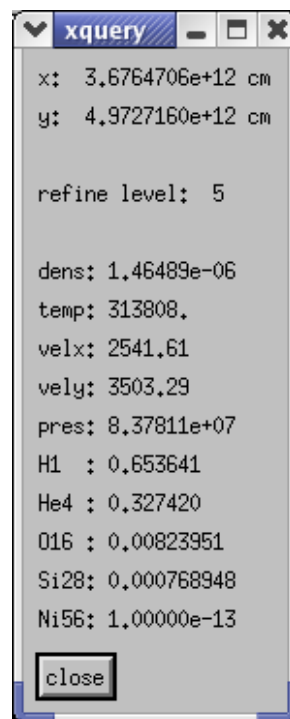


Figure 39.6: The xflash3 query widget, displaying information for a cell.





# Chapter 40

## convertspec3d

The Spect3D software, developed by Prism Scientific Computing:

<http://www.prism-cs.com/Software/Spect3D/Spect3D.htm>

is a widely used tool that can be used to model simulated diagnostic responses generated by radiation hydrodynamics codes. The simulated images mimic common diagnostics fielded in many HEDP experiments. The `convertspec3d` python script can be used to convert FLASH output into input for the Spect3D code.

### 40.1 Installation

The `convertspec3d` script is written in python and depends on several popular python packages. The dependencies include:

- Python 2.7
- PyTables
- The python NETCDF package: <http://code.google.com/p/netcdf4-python/>
- The setuptools package

Once these dependencies are installed, you can install the `convertspec3d` script. To do so, enter the `tools` directory of the FLASH source tree and enter:

```
python setup.py install
```

By default, this will try to install the FLASH python tools to a system wide location. You will need root/administrator privileges for this. If you are on a system where you do not have root privileges, just enter:

```
python setup.py install --user
```

for a user space installation. Type `convertspec3d` in the command line and you should see some help information - if the installation went well. For example, you should see something like:

```
> convertspect3d
usage: convertspec3d [-h] --species SPECIES [--extra EXTRA]
                    file_names [file_names ...]
convertspec3d: error: too few arguments
```

## 40.2 Usage

To describe the usage of the script, it will be assumed that the user is modeling a laser driven HEDP experiment. The example will be based on the **LaserSlab** simulation described in 35.7.5. The script essentially assumes you are running a simulation using the MTMMMT EOS, where the plasma electron number density and ion number density can be determined using the **SUMY** and **YE** variables. Spect3D requires this information for most applications. The simulation must have material information where species are used to model each material. In the case of the **LaserSlab** simulation, there are two species: **cham** and **targ**.

The script takes, as arguments, a list of one or more FLASH checkpoint/plot files and converts each of these files. The new files have the same names as the old files, but have a **.exo** extension. For example, the checkpoint file **lasslab\_hdf5\_chk\_0000** would be converted into the file **lasslab\_hdf5\_chk\_0000.exo**. The minimal information required by the script is the list of the FLASH output files and a list of the species. For example, for the **LaserSlab** simulation described in section 35.7.5, the command for convert the checkpoint files would be:

```
> convertspec3d lasslab_hdf5_chk_* --species=cham,targ
```

You should then see a series of **.exo** files generated. The new files contain the following information by default:

- The electron temperature in eV
- The ion temperature in eV
- The electron number density in 1/cc
- The ion number density in 1/cc
- The mass density in g/cc
- The “partial” mass density for each species (i.e. the fraction of the total density in each cell accounted for by each species)

With this information, users can run many Spect3D simulations.

If FLASH plot files are being converted, they must contain certain variables. The required variables are:

- **dens**
- **tele**
- **tion**
- **sumy**
- **ye**
- The variables for each species. In this example, **cham** and **targ**

In addition to simulating diagnostic responses, Spect3D can also plot simulation code output directly. Thus, even though Spect3D doesn't require it, users might want to plot the radiation temperature, or some other variable, in Spect3D. The **--extra** command line argument for **convertspec3d** allows users to add additional variables. For example, let's extend the **convertspec3d** command to include the total specific internal energy and radiation temperature:

```
> convertspec3d lasslab_hdf5_chk_* --species=cham,targ --extra=trad,eint
```

## Part XI

# Going Further with FLASH



rochester



## Chapter 41

# Adding new solvers

Adding new solvers (either for new or existing physics) to FLASH starts with a new subdirectory at an appropriate location in the source tree. If the solver provides an alternative implementation for the already supported physics, then the subdirectory should be added below the main sub-unit directory for that sub-unit. For example, adding a *Constant* implementation to the *Heatexchange* unit requires adding a subdirectory *Constant* under “source/physics/sourceTerms/Heatexchange/HeatexchangeMain”. If the solver is adding new physics to the code, a new capitalized subdirectory should be placed under the *physics* subdirectory, effectively creating a new unit, and its API should be defined by placing the null implementations of the relevant functions in the subdirectory. The implementations themselves should go into the *UnitMain* subdirectory placed in the unit. For example adding a *Heat* unit the to code requires putting a directory *Heat* under “source/physics”, and adding files *Heat\_init.F90*, *Heat\_finalize.F90* and at least one “doer” function such as *Heat.F90* with null implementations of the corresponding functions. Other than the source files, a *Makefile* and a *Config* file may need to be added to the subdirectory. If the solver is a new unit, both those files will be needed at the *Unit* and *UnitMain* levels. If the solver is an alternative implementation, the files may or may not be needed. See [http://flash.rochester.edu/site/flashcode/user\\_support/flash\\_howtos/ExampleFlashUnit/](http://flash.rochester.edu/site/flashcode/user_support/flash_howtos/ExampleFlashUnit/) for the architecture of a unit.

**Makefile:** The Makefile snippet added to any directory should reflect all the files in the subdirectory that are not inherited from a higher level directory. At the API level, the Makefile should include all the functions that constitute the API for the unit. At any lower level, those functions should not be included in the Makefile, instead all the local files at that level should be included. The implemented source files appear at *UnitMain* or some other peer level (if there is more than one sub-unit in the unit). An example of Makefile for a unit “Heat” with only three functions is as follows:

```
# Makefile for Heat unit API
Heat = Heat_init.o Heat_finalize.o Heat.o
```

All the subdirectories lying below this level have their Makefiles use macro concatenation to add to the list. For example, if the implementation of the Heat unit adds one file *he\_getTemp.F90* in the *HeatMain* implementation, the corresponding Makefile will be

```
# Makefile for HeatMain sub-unit
Heat += Heat\_data.o he_getTemp.o
```

Note that the sub-unit’s Makefile only makes reference to files provided at its own level of the directory hierarchy, and the units will have at least one data module (in this instance *Heat\_data*) which contains all the unit scope data. If the sub-unit provides special versions of routines to override those supplied by the any implementation at a higher level directory in the unit, they do not need to be mentioned again in the object list,

**Config:** Create a configuration file for the unit, sub-unit or alternative implementation you are creating. All configuration files in a sub-unit path are used by **setup**, so a sub-unit inherits its parent unit’s configuration. **Config** should declare any runtime parameters you wish to make available to the code when this

unit is included. It should indicate which (if any) other units your unit requires in order to function, and it should indicate which (if any) of its sub-units should be used as a default if none is specified when `setup` is run. The configuration file format is described in [Section 3.1](#).

This is all that is necessary to add a unit or sub-unit to the code. If you are creating a new solver for an existing physics unit, the unit itself should provide the interface layer to the rest of the code. As long as your sub-unit provides the routines expected by the interface layer, the sub-unit should be ready to work. However, if you are adding a new unit, you will need to add calls to your API routines elsewhere in the code as needed. The most likely place where these calls will need to be placed will be in the `Driver` unit.

It is difficult to give complete general guidance; here we simply note a few things to keep in mind. If you wish to be able to turn your unit on or off without recompiling the code, create a new runtime parameter (e.g., `useUnit`) at the API level of the your unit. You can then test the value of this parameter before calling your API routines from the main code. For example, the `Burn` unit routines are only called if (`useBurn==.true.`). (Of course, if the `Burn` unit is not included in the code, setting `useBurn` to true will result in empty subroutine calls.)

You will need to add `Unit_data.F90` file which will contain all the data you want to have visible everywhere within the unit. Note that this data is not supposed to be visible outside the unit. You may wish have `Driver_initFlash` call your unit's initialization routine and `Driver_finalizeFlash` to call your unit's finalize routine.

If your solver introduces a constraint on the timestep, you should create a routine named `Unit_computeDt()` that computes this constraint. Add a call to this routine in `Drive_computeDt.F90` (part of the `Driver` unit). Your routine should operate on a single block and take three parameters: the timestep variable (a real variable which you should set to the smaller of itself and your constraint before returning), the minimum timestep location (an integer array with five elements), and the block identifier (an integer). Returning anything for the minimum location is optional, but the other timestep routines interpret it in the following way. The first three elements are set to the coordinates within the block of the cell contributing the minimum timestep. The fourth element is set to the block identifier, and the fifth is set to the current processor identifier. This information tags, along with the timestep constraint, when blocks and solvers are compared across processors, and it is printed on stdout by the master processor along with the timestep information as FLASH advances.

If your solver is time-dependent, you will need to add a call to your solver in the `Drive_evolveFlash` routine. Limit the entry points into your unit to the API functions. It is a FLASH architecture requirement, violating this will make the code unmaintainable and will make it hard to get support from the code developers.



## Chapter 42

# Porting FLASH to other machines

Porting FLASH to new architectures should be fairly straightforward for most Unix or Unix-like systems. For systems which look nothing like Unix or which have no ability to interpret the setup script or makefiles, extensive reworking of the meta-code which configures and builds FLASH would be necessary. We do not treat such systems here; rather than do so, it would be simpler for us to do the port ourselves.

For Unix-like systems, you should make sure that your system has `csh`, a `gmake` that permits included makefiles, `awk`, `sed`, and `Python`. Next, create a directory in `sites/` with the name of your site. It is best to name your site according to the output from the `hostname -f` command (it will return something like `code.rochester.edu`) so that the setup script can automatically determine your site without you needing to manually specify the `-site` setup script option. Copy a `Makefile.h` from a pre-existing site directory into your new site directory.

### 42.1 Writing a Makefile.h

To reflect your system, you must modify the different macros defined in `Makefile`:

`FCOMP` : the name of the Fortran 90 compiler

`CCOMP` : the name of the C compiler

`CPPCOMP` : the name of the C++ compiler

`LINK` : the name of the linker (usually the Fortran compiler should serve as the linker)

`PP` : a flag (if any) that should precede a preprocessor directive (typically `-D`)

`FFLAGS_OPT` : the Fortran compilation flags to produce an optimized executable. These are the flags used when `-auto` is given to setup.

`FFLAGS_DEBUG` : the Fortran compilation flags to produce an executable that can be used with a debugger (*e.g.* totalview). These flags are used when `-debug` is passed to setup.

`FFLAGS_TEST` : Fortran compilation flags to produce an executable suitable for testing. These usually involve less optimization. These flags are used when `-test` is passed to setup.

`CFLAGS_OPT` : the optimized set of compilation flags for C/C++ code.

`CFLAGS_DEBUG` : the debug compilation flags for C/C++ code.

`CFLAGS_TEST` : the test compilation flags for C/C++ code.

`LFLAGS_OPT` : linker flags used with the `_OPT` compilation flags. This usually ends in `'-o'` to rename the executable.

LFLAGS\_DEBUG : linker flags used with the \_DEBUG compilation.

LFLAGS\_TEST : linker flags used with the \_TEST compilation.

LIB\_OPT : libraries to link in with the \_OPT compilation. This should include the MPI library if an MPI wrapper for the linker was not used (*e.g.* mpif90).

LIB\_DEBUG : libraries to link in the with the \_DEBUG compilation.

LIB\_TEST : libraries to link in with the \_TEST compilation.

LIB\_HDF5 the necessary link line required to link in the HDF5 library. This will look something like

```
-L /path/to/library -lhdf5
```

For example, here's how you might modify the macros defined in the `Makefile.h` for code.rochester.edu. The first part of `Makefile.h` defines the paths for the MPI wrapper script and various I/O and numerical libraries.

```
MPI_PATH   = /opt/mpich2/intel/1.4.1p1
HDF5_PATH  = /opt/hdf5/intel/1.8.7
HYPRE_PATH = /opt/hypre/intel/2.7.0b
NCMPI_PATH = /opt/netcdf/intel/1.2.0
```

These should be modified to reflect the locations on your system. Next we setup the compilers and linker. We almost always use the Fortran 90 compiler as the linker, so the Fortran libraries are automatically linked in.

```
FCOMP      = ${MPI_PATH}/bin/mpif90
CCOMP      = ${MPI_PATH}/bin/mpicc
CPPCOMP    = ${MPI_PATH}/bin/mpicxx
LINK       = ${MPI_PATH}/bin/mpif90
```

```
# pre-processor flag
PP         = -D
```

These commands will need to be changed if your compiler names are different. You are encouraged to use the compiler wrapper scripts instead of the compilers directly. Note that some older versions of MPICH do not recognize `.F90` as a valid extension. For these, you can either update MPICH to a later version or edit `mpif90` and add `.F90` to the line that checks for a valid extension. The `PP` macro refers to the pre-processor and should be set to the flag that the compiler uses to pass information to the C preprocessor (usually `-D`).

We define three different setups of compiler flags as described earlier: the “\_OPT” set for normal, fully optimized code, the “\_DEBUG” set for debugging FLASH, and the “\_TEST” set for regression testing. This latter set usually has less optimization. These three sets are picked with the `-auto`, `-debug`, and `-test` flags to `setup` respectively.

```
FFLAGS_OPT   = -c -g -r8 -i4 -O3 -real_size 64 -diag-disable 10120
FFLAGS_DEBUG = -c -g -r8 -i4 -O0 -check bounds -check format \
-check output_conversion -warn all -warn error -real_size 64 -check uninit \
-traceback -fp-stack-check -diag-disable 10120 -fpe0 -check pointers
FFLAGS_TEST  = ${FFLAGS_OPT} -fp-model precise
F90FLAGS     =

CFLAGS_OPT   = -c -O3 -g -D_LARGEFILE64_SOURCE -D_FORTIFY_SOURCE=2 \
-diag-disable 10120
CFLAGS_DEBUG = -c -O0 -g -traceback -debug all -debug extended \
-D_LARGEFILE64_SOURCE -diag-disable 10120 -ftrapuv -fp-stack-check
CFLAGS_TEST  = ${CFLAGS_OPT} -fp-model precise
```

Next come the linker flags. Typically, these have only `-o` to rename the executable, and some debug flags (*e.g.*, `-g`) for the “`_DEBUG`” set.

```
LFLAGS_OPT   = -diag-disable 10120 -O3 -o
LFLAGS_DEBUG = -diag-disable 10120 -o
LFLAGS_TEST  = ${LFLAGS_OPT}
```

Then there are macros for additional flags that may be required by various libraries used by FLASH, *e.g.*, HDF5. Macros beginning with `CFLAGS_` are used for compiling C source files to object files; macros beginning with `FFLAGS_` are used for compiling Fortran source files to object files; and macros beginning with `LIB_` are used for the command (see `LINK` macro above) that links object files into the FLASH executable. Note that `LIB_` macros need to be specified explicitly even if they have empty contents (as shown for `LIB_MPI`), if the corresponding library has been configured into FLASH.

```
CFLAGS_MPI   = -I$(MPI_PATH)/include
CFLAGS_HDF5  = -I$(HDF5_PATH)/include -DH5_USE_16_API
CFLAGS_NCMPi = -I$(NCMPI_PATH)/include
FFLAGS_HYPRE = -I${HYPRE_PATH}/include
```

```
LIB_MPI      =
LIB_HDF5     = -L$(HDF5_PATH)/lib -lhdf5_fortran -lhdf5 -lz
LIB_NCMPi    = -L$(NCMPI_PATH)/lib -lpnetcdf
LIB_HYPRE    = -L${HYPRE_PATH}/lib -lhypre
```

Finally, we have a macro to specify any platform dependent code and some macros for basic file manipulation and library tools.

```
MACHOBJ =

MV = mv -f
AR = ar -r
RM = rm -f
CD = cd
RL = ranlib
ECHO = echo
```

On most platforms, these will not need to be modified.



# Chapter 43

## Multithreaded FLASH

### 43.1 Overview

We have added the capability to run multiple threads of execution within each MPI task. The multithreading is made possible by usage of OpenMP which is described at [www.openmp.org](http://www.openmp.org). For information about building a multithreaded FLASH application please refer to Section 5.8.

The OpenMP parallel regions do not cover all FLASH units in FLASH4. At this time we have only focussed on the units which are exercised most heavily in typical FLASH applications: these are the split and unsplit hydrodynamics solvers, the unsplit magnetohydrodynamic solvers, Gamma law and multigamma EOS, Helmholtz EOS, Multipole solver (improved version (support for 2D cylindrical and 3D cartesian)) and energy deposition. The level of OpenMP coverage will increase in future releases.

The setup script will prevent you from setting up invalid threaded applications by exiting with a conflict error. We have added the conflicts to prevent using, for example, a non thread-safe EOS in a OpenMP parallel region. Example setup lines can be found in [sites/code.rochester.edu/openmp\\_test\\_suite/test.info](http://sites/code.rochester.edu/openmp_test_suite/test.info).

### 43.2 Threading strategies

We have experimented with two different threading strategies for adding an extra layer of parallelism to the standard MPI decomposition described in Chapter 8.

The first strategy makes use of the fact that the solution updates on different Paramesh blocks are independent, and so, multiple threads can safely update the solution on different blocks at the same time. This is a coarse-grained threading approach which is only applicable to Paramesh. It can be included in a FLASH application by adding `threadBlockList=True` to the FLASH setup line.

```
call Grid_getListOfBlocks(LEAF,blockList,blockCount)
!$omp do schedule(static)
do b=1,blockCount
  blockID = blockList(b)
  call update_soln_on_a_block(blockID)
```

The second strategy parallelises the nested do loops in the kernel subroutines such that different threads are updating the solution on independent cells from the same block at the same time. This is a fine-grained threading approach which is applicable to both Paramesh and UG. It can be included in a FLASH application by adding `threadWithinBlock=True` to the FLASH setup line. Notice that we parallelise the outermost do loop for a given dimensionality to improve cache usage for each thread.

```
#if NDIM == 3
  !$omp do schedule(static)
#endif
do k=k0-2,kmax+2
```

```

#if NDIM == 2
    !$omp do schedule(static)
#endif
    do j=j0-2,jmax+2
#if NDIM == 1
    !$omp do schedule(static)
#endif
    do i=i0-2,imax+2
        soln(i,j,k) = ....
    
```

The final threading option is only applicable to the energy deposition unit. In this unit we assign different rays to each thread, where the number of rays assigned is dynamic because the work per ray is not fixed. It can be included in a FLASH application by adding `threadRayTrace=True` to the FLASH setup line

It is perfectly fine to mix these setup options except for `threadBlockList=True` and `threadWithinBlock=True`.

## 43.3 Running multithreaded FLASH

### 43.3.1 OpenMP variables

There are OpenMP environmental variables which control the threaded runtime. The most frequently used is `OMP_NUM_THREADS` which controls the number of OpenMP threads in each parallel region. It should be exported to your environment from your current shell. In the following example we specify that each OpenMP parallel region will be executed by 4 threads (assuming bash shell)

```
export OMP_NUM_THREADS=4
```

You may also need to set `OMP_STACKSIZE` to increase the stack size of the threads created by the OpenMP runtime. This is because some test problems exceed the default limit which leads to a segmentation fault from an innocent piece of source code. In practice we have only encountered this situation in a White Dwarf Supernova simulation with block list threading on a machine which had a default stack size of 4MB. For safety we now set a default value of 16MB for all of our runs to provide plenty of stack space for each thread.

```
export OMP_STACKSIZE=16M
```

If you need to use a job submission script to run FLASH on compute nodes then you should export the OpenMP variables in your job submission script.

### 43.3.2 FLASH variables

There are FLASH runtime parameters which can be set to `.true.` or `.false.` to turn on/off the parallel regions in different units. They are

Unit	Block list threading	Within block threading	Ray trace threading
Hydro	<code>threadHydroBlockList</code>	<code>threadHydroWithinBlock</code>	N/A
EOS	N/A	<code>threadEosWithinBlock</code>	N/A
Multipole	<code>threadMpoleBlockList</code>	<code>threadMpoleWithinBlock</code>	N/A
Energy Deposition	N/A	N/A	<code>threadRayTrace</code>

There is no parameter for block list threading of EOS because, quite simply, there is no block list in EOS. In many FLASH simulations the EOS subroutines are only called from the hydrodynamic subroutines which means the EOS will be called in parallel if you thread the Hydro unit block list loop.

In general you do not need to worry about these runtime parameters because the default values will be set according to the setup variable passed to the setup script. This means that if you setup a Sedov simulation with block list threading then `threadHydroBlockList` will default to `.true.` and `threadHydroWithinBlock` will default to `.false.`. The runtime control of parallel regions may be useful if we ever need to investigate a possible multithreaded bug in a particular unit.

### 43.3.3 FLASH constants

There are FLASH constant runtime parameters which describe the threading strategies included in your FLASH application. The constants are named `threadBlockListBuild`, `threadWithinBlockBuild` and `threadRayTraceBuild`. We query these constants in the unit initialization file (e.g. `Hydro_init.F90`) to verify that your FLASH application supports the parallel regions requested by the runtime parameters in 43.3.2. If the parallel regions are not supported you will get a warning message in your FLASH log file and the parallel regions will be disabled. If this happens you can still use the requested parallel regions but you must re-setup FLASH with the appropriate threading strategy. In general you do not need to worry about the existence of these constants because if you setup the application with, say, block list threading it only makes sense for you to adjust the block list threading runtime parameters in your `flash.par`.

A nice property of using constant runtime parameters instead of pre-processor defines is that `-noclobber` rebuilds are extremely fast. This means you can resetup a `threadBlockList` application with `threadWithinBlock` and only rebuild the handful of source files that actually changed. If we used pre-processor defines to identify the thread strategy then we would have to rebuild every single file because any file could make use of the define. We actually take advantage of the forced rebuild from changing/adding/removing a pre-processor define when we switch from a threaded to a non-threaded application. We do this because OpenMP directives are conditionally compiled when a compiler option such as `-fopenmp` (GNU) is added to the compile line. Any source file could contain OpenMP and so we must do a complete rebuild to ensure correctness. We force a complete rebuild by defining `FLASH_OPENMP` in `Flash.h` for any type of multithreaded build.

## 43.4 Verifying correctness

The hybrid (MPI+OpenMP) tests using only the multithreaded Hydro and Eos units have the nice property that they **can** give identical results to the corresponding MPI only test. I emphasise **can** because it is possible for an optimized build of the MPI+OpenMP application to give a different answer to the MPI only application. This is because the OpenMP directives can impede the compiler from optimizing the floating point calculations. Therefore, to obtain an exact match between an unthreaded and threaded test you should add a compiler option which prevents the compiler from making any aggressive floating point transformations. For the Intel compiler we needed to add `-fp-model precise` to both the MPI-only and MPI+OpenMP compiler flags to get the same answer. It should be possible to get identical answers for tests like Sod 35.1.1 and Sedov 35.1.4.

This approach does not hold if you use the multithreaded Multipole or energy deposition unit because the floating point accumulation order is different to the order in the unthreaded versions. In the multithreading of the Multipole solver we give each thread its own private moments array and then accumulate the private moment arrays into a single moments array. Similarly, in the energy deposition unit we give each thread a private energy deposition array which is then accumulated into `DEPO_VAR` mesh variable. It is reasonable to expect a small discrepancy, e.g. a magnitude error of  $1\text{E-}14$  in `DEPO_VAR`, in a single time step, but be aware that over a large number of time steps the accumulation of small differences can become much larger.

## 43.5 Performance results

We present some performance results for each of the multithreaded FLASH units. It is important to note that the multithreaded speedup of each unit is not necessarily representative of a full application because a full application spends time in units which are not (currently) threaded, such as Paramesh, Particles and I/O.

All of the experiments are run on Eureka, a machine at Argonne National Laboratory, which contains 100 compute nodes each with 2 x quad-core Xeon w/32GB RAM. We make use of 1 node and run FLASH with 1 MPI task and 1-8 OpenMP threads.

### 43.5.1 Multipole solver

We tested the MacLaurin Spheroid problem described in Section 35.3.4 with the new Multipole solver described in Section 8.10.2.2 using three FLASH applications: one with Paramesh and block list threading,

one with Paramesh and within block threading and one with UG and within block threading. The FLASH setup lines are

```
./setup unitTest/Gravity/Poisson3 -auto -3d -maxblocks=600 -opt \
+pm4dev +newMpole +noio threadBlockList=True timeMultipole=True

./setup unitTest/Gravity/Poisson3 -auto -3d -maxblocks=600 -opt \
+pm4dev +newMpole +noio threadWithinBlock=True timeMultipole=True

./setup unitTest/Gravity/Poisson3 -auto -3d +ug -opt +newMpole +noio \
threadWithinBlock=True timeMultipole=True -nxb=64 -nyb=64 -nzb=64
```

The effective resolution of the experiments are the same: the Paramesh experiments have `lrefine_max=4` and blocks containing 8 internal cells along each dimension and the UG experiment has a single block containing 64 internal cells along each dimension. We add the setup variable `timeMultipole` to include a custom implementation of `Gravity_potentialListOfBlocks.F90` which repeats the Poisson solve 100 times.

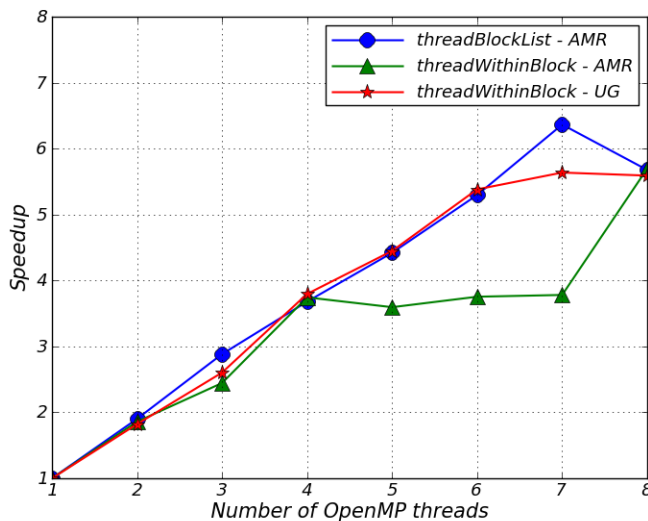


Figure 43.1: Speedup of the threaded Multipole solver in the Maclaurin Spheroid test problem.

The results in Figure 43.1 demonstrate multithreaded speedup for three different configurations of FLASH. The speedup for the AMR `threadWithinBlock` application is about the same for 4 threads as it is for 5, 6 and 7 threads. This is because we parallelize only the slowest varying dimension with a static loop schedule and so some threads are assigned 64 cells per block whilst others are assigned 128 cells per block. We do not see the same issue with the UG `threadWithinBlock` application because the greater number of cells per block avoids the significant load imbalance per thread.

### 43.5.2 Helmholtz EOS

We tested the Helmholtz EOS unit test described in Section 17.7 with a UG and a NoFBS FLASH application. The FLASH setup lines are

```
./setup unitTest/Eos/Helmholtz -auto -3d +ug threadWithinBlock=True \
timeEosUnitTest=True +noio
```



```
./setup unitTest/Eos/Helmholtz -auto -3d +nofbs threadWithinBlock=True \
timeEosUnitTest=True +noio
```

The `timeEosUnitTest` setup variable includes a version of `Eos_unit.F90` which calls `Eos_wrapped` 30,000 times in `MODE_DENS_TEMP`.

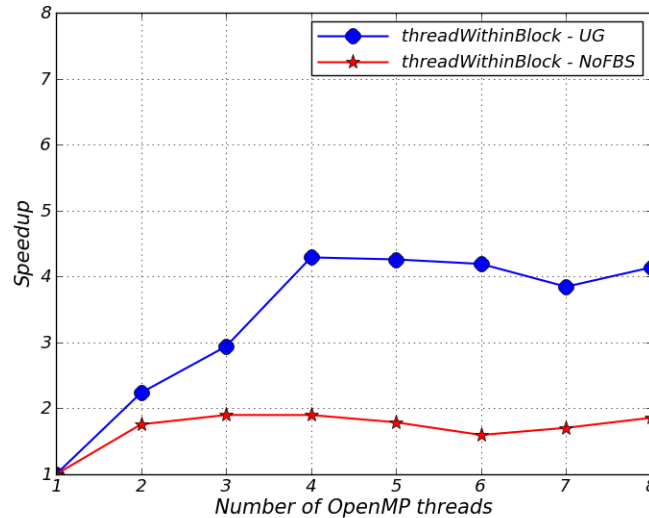


Figure 43.2: Speedup of the threaded Helmholtz EOS in the EOS unit test.

The results in Figure 43.2 show that the UG version gives better speedup than the NoFBS version. This is most likely because the NoFBS version contains dynamic memory allocation / deallocation within `Eos_wrapped`.

The UG application does not show speedup beyond 4 threads. This could be because the work per thread for each invocation of `Eos_wrapped` is too small, and so it would be interesting to re-run this exact test with blocks larger than  $8 \times 8 \times 8$  cells. Unfortunately the Helmholtz EOS unit test does not currently support larger blocks – this is a limitation of the Helmholtz EOS unit test and not Helmholtz EOS.

### 43.5.3 Sedov

We ran 2d and 3d multithreaded versions of the standard Sedov simulation described in Section 35.1.4. We tested both split and unsplit hydrodynamic solvers and both `threadBlockList` and `threadWithinBlock` threading strategies. The 2d setup lines are

```
./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \
-nxb=16 -nyb=16 threadBlockList=True +uhd
```

```
./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \
-nxb=16 -nyb=16 threadWithinBlock=True +uhd
```

```
./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \
-nxb=16 -nyb=16 threadBlockList=True
```

```
./setup Sedov -2d -auto +pm4dev -parfile=coldstart_pm.par \
-nxb=16 -nyb=16 threadWithinBlock=True
```

and the 3d setup lines are

```

./setup Sedov -3d -auto +pm4dev -parfile=coldstart_pm_3d.par \
+cube16 threadBlockList=True +uhd

./setup Sedov -3d -auto +pm4dev -parfile=coldstart_pm_3d.par \
+cube16 threadWithinBlock=True +uhd

./setup Sedov -3d -auto +pm4dev -parfile=coldstart_pm_3d.par \
+cube16 threadBlockList=True

./setup Sedov -3d -auto +pm4dev -parfile=coldstart_pm_3d.par \
+cube16 threadWithinBlock=True

```

The results are shown in Figures 43.3 and 43.4. The speedup graph is based on times obtained from the “hydro” timer label in the FLASH log file.

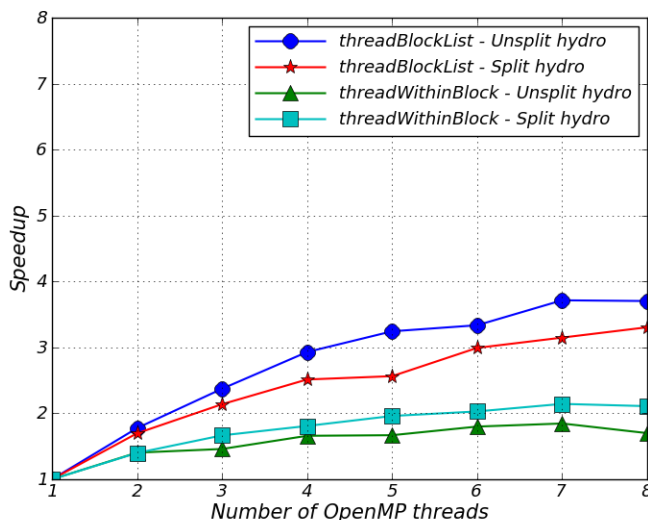


Figure 43.3: Speedup of the threaded hydrodynamic solvers in a 2d Sedov test.

Part of the reason for loss of speedup in Figures 43.3 and 43.4 is that the hydrodynamic solvers in FLASH call Paramesh subroutines to exchange guard cells and conserve fluxes. Paramesh is not (yet) multithreaded and so we call these subroutines with a single thread. All other threads wait at a barrier until the single thread has completed execution.

The unsplit hydro solver has one guardcell fill per hydro timestep and the split hydro solver has NDIM guardcell fills per hydro timestep (one per directional sweep). The extra guardcell fills are part of the reason that the split applications generally give worse speedup than the unsplit applications.

#### 43.5.4 LaserSlab

We finish with a scaling test for the ray trace in the energy deposition unit. We setup the LaserSlab simulation described in Section 35.7.5 with the additional setup option `threadRayTrace=True`. The setup line and our parameter modifications are

```

./setup LaserSlab -2d -auto +pm4dev -geometry=cylindrical \
-nxb=16 -nyb=16 +mtmmmt +laser +mgd +uhd3t species=cham,targ \
mgd_meshgroups=6 -without-unit=flashUtilities/contiguousConversion \
-without-unit=flashUtilities/interpolation/oneDim \

```

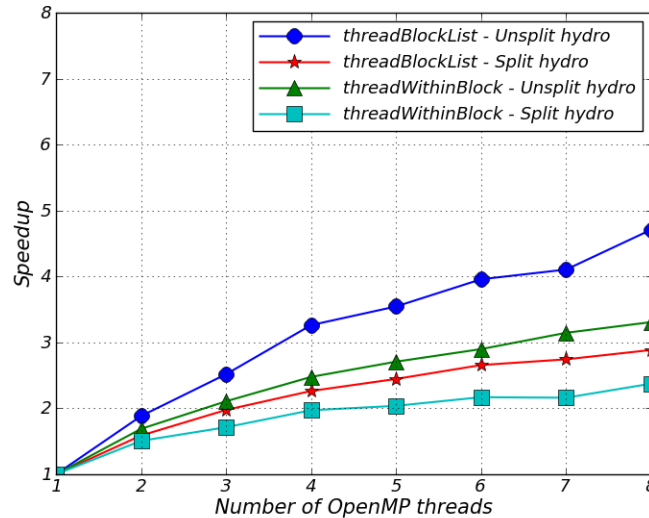


Figure 43.4: Speedup of the threaded hydrodynamic solvers in a 3d Sedov test.

```
-parfile=coldstart_pm_rz.par threadRayTrace=True +noio
```

```
ed_maxRayCount = 80000
ed_numRays_1 = 50000
nend = 20
```

The speedup is calculated from times obtained from the “Transport Rays” label in the FLASH log file and the speedup graph is shown in Figure 43.5. The timer records how long it takes to follow all rays through the complete domain during the timestep. As an aside we remove the flashUtilities units in the setup line to keep the number of line continuation characters in a setup script generated Fortran file less than 39. This is just a workaround for picky compilers; another strategy could be a compiler option that allows compilation of Fortran files containing more than 39 continuation characters.

## 43.6 Conclusion

At the current time we have only really exposed an extra layer of parallelism in FLASH and have not yet focused on tuning the multithreading. We do not have enough experience to suggest the most efficient ways to run the multithreaded code, however, we can suggest some things that may help the efficiency. In an application setup with `threadBlockList` it makes sense to maintain at least 5 blocks per thread. This is because the computational load imbalance between thread A being assigned 1 block and thread B being assigned 2 blocks is larger than thread A being assigned 5 blocks and thread B being assigned 6 blocks. In an application setup with `threadWithinBlock` you should probably use larger blocks, perhaps  $16 \times 16 \times 16$  or even  $32 \times 32 \times 32$ , so that each thread has more cells to work on.

As a closing note, you should be aware of the amount of time spent in the threaded FLASH units compared to the non-threaded FLASH units in your particular FLASH application - perfect speedup in a threaded unit may be insignificant if most of the time is spent in a non-threaded FLASH unit.

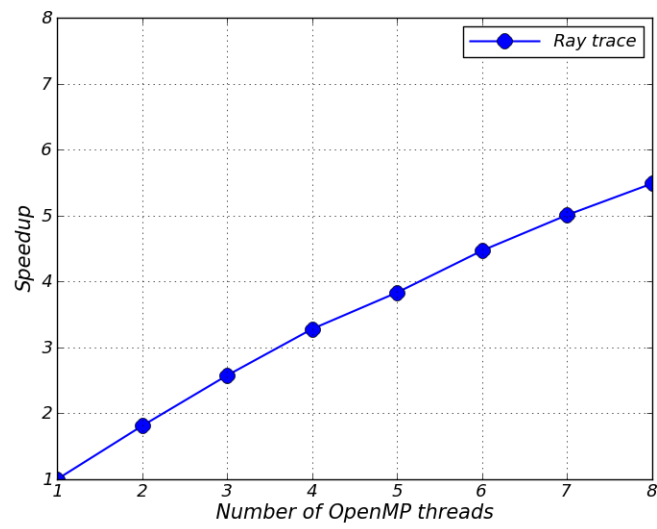


Figure 43.5: Speedup of the energy deposition ray trace in the LaserSlab test.

# Back Matter



# References

- Anninos, W. Y., & Norman, M. L. 1994, ApJ, 429, 434
- Aparicio, J. M. 1998, ApJS, 117, 627
- Atzeni, S. and Meyer-Ter-Vehn, J. 2004, The Physics of Inertial Fusion: Beam Plasma Interaction, Hydrodynamics, Hot Dense Matter (Oxford Science Publications)
- Armfield, S. & Street, R. 2002, Int. J. Numer. Meth. Fluids, 38, 255-282
- Bader, G. & Deuffhard, P. 1983, NuMat, 41, 373
- Balsara, D. S., 2001, JCP, 174, 614
- Balsara, D. S. 2004, ApJS, 151, 149
- Balsara, D. S., & Spicer, D. S. 1999, JCP, 149, 270
- Bardeen, J. M., Bond, J. R., Kaiser, N., & Szalay, A. S. 1986, ApJ, 304, 15
- Barnes, J. & Hut, P. 1986, Nature, 324, 446
- Berger, M. J. & Collela, P. 1989, JCP, 82, 64
- Berger, M. J. & Oliger, J. 1984, JCP, 53, 484
- Blinnikov, S. I., Dunina-Barkovskaya, N. V., & Nadyozhin, D. K. 1996, ApJS, 106, 171
- Boris, J. P. & Book, D. L. 1973, JCP, 11, 38
- Bonnor, W. B. 1956, MNRAS, 116, 351
- Brackbill, J. & Barnes, D. C. 1980 JCP, 35, 426
- Braginskii, S. I. 1965 Rev. Plasma Phys., 1, 205
- Brandt, A. 1977, Math. Comp., 31, 333
- Brio, M. & Wu, C. C. 1988 JCP, 75, 400
- Brown, P. N., Byrne, G. D., & Hindmarsh, A. C. 1989, SIAM J. Sci. Stat. Comput., 10, 1038
- Brysk, H., Campbell, P.M., Hammerling, P. 1974, Plasma Physics, 17, 473
- Bunn, E. F. & White, M. 1997, ApJ, 480, 6
- Burgers, J. M. 1969, Flow Equations for Composite Gases (New York: Academic)
- Nakamura, K. *et al.* 2010, J. Phys., G 37, 075021 (also [http://pdg.lbl.gov/2010/reviews/contents\\_sports.html](http://pdg.lbl.gov/2010/reviews/contents_sports.html))
- Caughlan, G. R. & Fowler, W. A. 1988, Atomic Data and Nuclear Data Tables, 40, 283

- Chandrasekhar, S. 1961, *Hydrodynamic and Hydromagnetic Stability* (Oxford: Clarendon)
- Chandrasekhar, S. 1939, *An Introduction to the Study of Stellar Structure* (Toronto: Dover)
- Chandrasekhar, S. 1987, *Ellipsoidal Figures of Equilibrium* (New York: Dover)
- Chapman, D. L. 1899, *Philos. Mag.*, 47, 90
- Chapman, S. & Cowling, T. G. 1970, *The Mathematical Theory of Non-uniform Gases* (Cambridge: CUP)
- Chatzopoulos, E. & Weide, K. 2017, arXiv:1712.10091
- Christy, R. F. 1966, *ApJ*, 144, 108
- Cohen, E. R. & Taylor, B. N. 1987, *Rev.Mod.Phys.*, 59, 1121
- Colella, P. & Glaz, H. M. 1985, *JCP*, 59, 264
- Colella, P. & Woodward, P. 1984, *JCP*, 54, 174
- Colgate, S. A., & White, R. H. 1966, *ApJ*, 143, 626
- Colin, O. & Ducros, F. & Veynante, D. & Poinso, T. 2000, *Physics of Fluids*, 12, 7, 1843
- Couch, S. M., Graziani, C. & Flocke, N. 2013, *ApJ*, 778, 181
- Decyk, V. K., Norton, C. D., & Szymanski, B. K. 1997, *ACM Fortran Forum*, 16 (also <http://www.cs.rpi.edu/~szymansk/oof90.html>)
- A. Dedner, F. Kemm, D. Krner, C.-D. Munz, T. Schnitzer, and M. Wesenberg, Hyperbolic Divergence Cleaning for the MHD Equations, *Journal of Computational Physics*, vol. 175, no. 2, pp. 645673, Jan. 2002, doi: 10.1006/jcph.2001.6961.
- DeZeeuw, D. & Powell, K. G. 1993, *JCP*, 104, 56
- Doring, W. 1943, *Ann. Phys.*, 43, 421
- Dubey, A., Daley, C., ZuHone, J., Ricker, P. M., Weide, K. & Graziani, C. 2012, *ApJS*, 201, 27
- Duff, I. S., Erisman, A. M., & Reid, J. K. 1986, *Direct Methods for Sparse Matrices* (Oxford: Clarendon Press)
- Einfeldt, B., Munz, C. D., Roe, P. L., & Sjögren, B. 1991, *JCP*, 92, 273–295
- Emery, A. F. 1968, *JCP*, 2, 306
- Epperlein, E. M. & Haines, M. G. 1986, *Phys. Fluids*, 29, 1029
- Eswaran, V. and Pope, S. B. 1988, *Computers & Fluids*, 16, 257–278
- Evans, C. R. & Hawley, J. F. 1988, *ApJ*, 332, 659
- Ewald, P. P. 1921, *Ann. Phys.*, 64, 253
- Federrath, C., Banerjee, R., Clark, P. C. & Klessen, R. S. 2010, *ApJ* 713, 269
- Federrath, C., Roman-Duval, J., Klessen, R. S., Schmidt, W., Mac Low, M.-M. 2010, *A&A*, 512, A81
- Ferguson, J. W., Alexander, D. R., Allard, F., Barman, T., Bodnarik, J. G., Hauschildt, P. H., Heffner-Wong, A., & Tamanai, A. 2005, *ApJ*, 623, 585
- Fletcher, C. A. J. 1991, *Computational Techniques for Fluid Dynamics*, 2d ed. (Berlin: Springer)
- Flocke, N. 2015, *ACM Transactions on Mathematical Software*, 41, Article 30



- Forester, C. K. 1977, JCP, 23, 1
- Foster, P. N., & Chevalier, R. A. 1993, ApJ, 416, 303
- Froula, D. H., Glenzer, S. H., Luhmann Jr., N. C., & Sheffield, J. 2011, Plasma Scattering of Electromagnetic Radiation, 2d ed. (Elsevier: AP)
- Fryxell, B. A., Müller, E., & Arnett, D. 1989, in Numerical Methods in Astrophysics, ed. P. R. Woodward (New York: Academic)
- Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., & Tufo, H. 2000, ApJS, 131, 273
- Gardiner, T., & Stone, J. 2005, JCP, 205, 509
- Gittings, M., et al. 2008, Computational Science and Discovery, 1, 015005
- Godunov, S. K. 1959, Mat. Sbornik, 47, 271
- Grevesse, N., & Sauval, A. J. 1998, Space Science Reviews, 85, 161
- Harten, A. 1983, JCP, 49, 357–393
- Hawley, J. F. 2000, ApJ, 528, 462
- Holmström, M., Fatemi, S., Futaana, Y., & Nilsson, H. 2012, The interaction between the Moon and the solar wind. Earth, Planets and Space, 74, 237–245
- Holmström, M. 2013, Handling vacuum regions in a hybrid plasma solver. Numerical modeling of space plasma flows (ASTRONUM-2012), ASP Conference Series, vol. 474, 202–207. arXiv:1301.0272
- Hu, W. & Sugiyama, N. 1996, ApJ, 471, 542
- Huang, J. & Greengard, L. 2000, SIAM. J. Sci. Comput., 21, 1551
- Huba, J.D. 2009, NRL Plasma Formulary (NRL/PU/6790–11-551).
- Huba, J.D. 2003, Space Plasma Simulation, 615
- Iben, I. Jr. 1975, ApJ, 196, 525
- Itoh, N., Hayashi, H., Nishikawa, A., & Kohyama, Y. 1996, ApJS, 102, 411
- Jackson, A. P. & Townsley, D. M. & Calder, A. C. 2014, ApJ, 784, 174
- James, R. A. 1977, JCP, 25, 71
- Jeans, J. H. 1902, Phil. Trans. Roy. Soc. (London), 199, 1
- Jouguet, E. 1905, J. Math. Pure Appl., 1, 347
- Kaiser, T. B. 2000, Phys. Rev. E, 61, 895
- Khokhlov, A. M. 1997, Naval Research Lab memo 6406-97-7950
- Krumholz, M., Klein, R., McKee, C., & Bolstad, J. 2007, ApJ, 667, 626
- Kurganov, A., Noelle, S., & Petrova, G. 2001, SIAM. J. 23, 707
- Kurganov, A., & Tadmor, E. 2000, JCP, 160, 241
- Lacey, C. G. & Cole, S. 1993, MNRAS, 262, 627
- Ledvina, S., Ma, Y.-J., and Kallio, E., Modeling and Simulating Flowing Plasmas and Related Phenomena, Space Sci. Rev., 143–189, 2008.

- Lee, D. 2006, Ph.D. Dissertation, Univ. of Maryland
- Lee, D. & Deane, A. 2009, JCP, 228, 952
- Lee, D. 2013, JCP, 243, 269–292, doi:10.1016/j.jcp.2013.02.049
- Lee, Y. T. & More, R. M. 1984, Phys. Fluids, 27, 1273
- Li, S., & Li, H. 2004, JCP, 199, 1
- Li, S. 2005, JCP, 203, 344–357
- Löhner, R. 1987, Comp. Meth. App. Mech. Eng., 61, 323
- LeVeque, R. J. 1992, Numerical Methods for Conservation Laws, Birkhäuser.
- LeVeque, R. J. 1997, JCP, 131, 327
- LeVeque, R. J. 1998, “Nonlinear Conservation Laws and Finite Volume Methods”, LeVeque, R. J., Mihalas, D., Dorfi, E. A., Mueller, E. (ed.), Computational Methods for Astrophysical Fluid Flow, Springer.
- Lowrie, R. 2008, Shock Waves, 18, 129
- MacLaurin, C. 1801, A Treatise on Fluxions, (W. Baynes), Available on <http://books.google.com>
- MacNeice, P., Olson, K. M., Mobarry, C., de Fainchtein, R., & Packer, C. 1999, CPC, 126, 3
- Marder, B. 1987, JCP, 68, 48
- Martí, J. M., Müller, E., 2003, Living Rev. Relativ., 6, 7, .
- Martin, D. & Cartwright, K., “Solving Poisson’s Equation using Adaptive Mesh Refinement.” (<http://seesar.lbl.gov/anag/staff/martin/AMRPoisson.html>)
- Martin, D., & Cartwright, K. 1996, “Solving Poisson’s Equation using Adaptive Mesh Refinement”, Technical Report No. UCB/ERL M96/66
- Matthews, A. P., Current Advance Method and Cyclic Leapfrog for 2D Multispecies Hybrid Plasma Simulations, Journal of Computational Physics, 112, 102–116, 1994.
- Mignone, A., Plewa, T., Bodo, G. 2005, ApJ, 160, 199
- Mignone, A., Bodo, G., Massaglia, S., Matsakos, T., Tesileanu, O., Zanni, C., & Ferrari, A. 2007, ApJS, 170, 228.
- Miller, G. H. & Puckett, E. G. 1996 JCP, 128, 134–164
- Miyoshi, T. & Kusano K. 2005, JCP, 208, 315–344
- Mönchmeyer, R., & Müller, E. 1989, A&A, 217, 351
- Morozova, V., Piro, A. L., Renzo, M., Ott, C. D., Clausen, D., Couch, S. M., Ellis, J., & Roberts, L. F. 2015, ApJ, 814, 63
- Müller, E., & Steinmetz, M. 1995, Comp. Phys. Comm., 89, 45
- Munz, C. D., Omnes, P., Schneider, R., Sonnendrücker, & Voß, U. 2000, JCP, 161, 484
- Nadyozhin, D. K. 1974, Nauchnye informatsii Astron, Sov. USSR, 32, 33
- Noh, W. F. 1987, JCP, 72, 78
- Orszag, S. A., Tang, C.-M., 1979, JFM, 90, 129

- Ostriker, J.P., ed. 1992, *Selected Works of Yakov Borisovich Zeldovich* (Princeton Univ. Press)
- Paczyński B., & Wiita, P. J. 1980, *A&A*, 88, 23.
- Parashar M., 1999, private communication (<http://www.caip.rutgers.edu/~parashar/DAGH>)
- Paxton, B., Bildsten, L., Dotter, A., Herwig, F., Lesaffre, P., & Timmes, F. 2011, *ApJS*, 192, 3
- Paxton, B., Cantiello, M., Arras, P., Bildsten, L., Brown, E. F., Dotter, A., Mankovich, C., Montgomery, M. H., Stello, D., Timmes, F. X., & Townsend, R. 2013, *ApJS*, 208, 4
- Paxton, B., Marchant, P., Schwab, J., Bauer, E. B., Bildsten, L., Cantiello, M., Dessart, L., Farmer, R., Hu, H., Langer, N., Townsend, R. H. D., Townsley, D. M. & Timmes, F. X. 2015, *ApJS*, 220, 15
- Pert, G.J., 1977, *J. Plasma Physics*, vol.18, part 2, pp. 227-241
- Plewa, T. & Müller, E. 1999, *A&A*, 342, 179.
- Potekhin, A. Y., Chabrier, G., & Yakovlev, D. G. 1997, *A&A*323, 415
- Powell, K. G., Roe, P. L., Linde, T. J. , Gombosi, T. I., & DeZeeuw, D. L. 1999, *JCP*, 154, 284
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, *Numerical Recipes in Fortran*, 2d ed. (Cambridge: CUP)
- Price, D. J. & Federrath, C. 2010, *MNRAS*, 406, 1659
- Pritchett, Philip L., *Particle-in-cell Simulations of Magnetosphere Electrodynamics*, *IEEE Transactions on Plasma Science*, 28, 6, 1976–1990, 2000.
- Quirk, J. J., in *Upwind and High-Resolution Schemes*, ed. M. Yousuff Hussaini, Bram van Leer, and John Van Rosendale, 1997, (Berlin: Springer).
- Reinicke, P. and Meyer-ter-Vehn, J. 1991, *Phys. Fluids A*, 3, 1807
- Ricker, P. M. 2008, arXiv: 0710.4397, *ApJS*, 176, 293
- Rider, W. J. & Kothe, D. B. 1998, *JCP*, 141, 112-152
- Ross, R., Nurmi, D., Cheng, A. & Zingale, M. 2001, *Proceedings of SC2001*.
- Salmon, J. K. & Warren, M. S. 1994, *JCP*, 111, 136
- Schulz-Rinne, C. W, Collins, J. P., and Glaz, H. M. 1993, *SIAM J. Sci. Comput.* 14, 1394.
- Sedov, L. I. 1959, *Similarity and Dimensional Methods in Mechanics* (New York: Academic)
- Shu, C.-W. 1998, in *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations*, ed. A. Quarteroni, *Lecture Notes in Mathematics*, 1697, (Berlin: Springer)
- Shu, C.-W. & Osher, S. 1989, *JCP*, 83, 32
- Skinner, M. A., & Ostriker, E. C. *ApJS*, 188, 290
- Spitzer, L. 1962, *Physics of Fully Ionized Gases*. (New York: Wiley)
- Sportisse, B. 2000, *JCP*, 161, 140
- Springel, V. 2005, *MNRAS*, 364, 1105
- Sod, G. 1978, *JCP*, 27, 1
- Strang, G. 1968, *SIAM J. Numer. Anal.*, 5, 506

- Timmes, F. X. 1992, *ApJ*, 390, L107
- Timmes, F. X. 1999, *ApJS*, 124, 241
- Timmes, F. X. 2000, *ApJ*, 528, 913
- Timmes, F. X. & Arnett, D. 2000, *ApJS*, 125, 294
- Timmes, F. X. & Brown, E. 2002 (in preparation)
- Timmes, F. X. & Swesty, F. D. 2000, *ApJS*, 126, 501
- Timmes, F. X. & Zingale, M. & Olson, K. & Fryxell, B. & Ricker, P. & Calder, A. C. & Dursi, L. J. & Tufo, H. & MacNeice, P. & Truran, J. W. & Rosner, R. 2000, *ApJ*543, 938
- Toro, E. F. 1999, *Riemann Solvers and Numerical Methods for Fluid Dynamics*, 2nd Ed. (New York: Springer)
- Tóth, G. 2000, *JCP*, 161, 605
- Townsend, D. M. & Calder, A. C. & Asida, S. M. & Seitzzahl, I. R. & Peng, F. & Vladimirova, N. & Lamb, D. Q. & Truran, J. W. 2007, *ApJ*, 668, 1118
- Trottenberg, U., Oosterlee, C., & Schüller, A. 2001, *Multigrid* (San Diego: Academic Press)
- Tzeferacos, P., Fatenejad, M., Flocke, N., Gregori, G., Lamb, D. Q., Lee, D., Meinecke, J., Scopatz, A., & Weide, K. 2012 *HEDP*, 8, 322, doi:10.1016/j.hedp.2012.08.001
- Uhlenbeck, G. E. & Ornstein, L. S. 1930, *Phys. Rev.* 36, 823
- van der Vorst, H. 1992, “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems”, *SIAM J. Sci. and Stat. Comput.*, 13:2, 631-644
- van Leer, B. 1979, *JCP*, 32, 101
- Vanella, M., Rabenold, P. & Balaras, E. 2010, *J. Comput. Phys.*, 229:18, 6427-6449
- von Neumann, J. 1942, *ORSD Rep. No. 549*
- Velikovich, A. L., Giuliani, J. L., Zalesak, S. T., Thornhill, J. W., & Gardiner, T. A. 2012, *Phys. Plasmas*, 19, 012707
- Vladimirova, N. & Weirs, G. & Ryzhik, L. 2006, *Combustion Theory and Modelling*, 10, 727
- Wallace, R. K., Woosley, S. E., & Weaver, T. A. 1982, *ApJ*, 258, 696
- Warren, M. S. & Salmon, J. K. 1993, in *Proc. Supercomputing 1993* (Washington, DC: IEEE Computer Soc.)
- Weaver, T. A., Zimmerman, G. B., & Woosley, S. E. 1978, *ApJ*, 225, 1021
- Williams, F. A. 1988, *Combustion Theory* (Menlo Park: Benjamin-Cummings)
- Williamson, J. H. 1980, *JCP*, 35, 48
- Winske, D. and Leroy, M. M., Diffuse ions produced by electromagnetic ion beam instabilities, *Journal of Geophysical Research*, 89, 2673–2688, 1984.
- Winske, D. and Quest, K. B., Electromagnetic Ion Beam Instabilities: Comparison of one- and two-dimensional simulations, *Journal of Geophysical Research*, 91, 8789–8797, 1986.
- Woodward, P. & Colella, P. 1984, *JCP*, 54, 115
- Xin, J. 2000, *SIAM Rev.*, 42, 161

- Yakovlev, D. G., & Urpin, V. A. (YU) 1980 24, 303
- Yang, J. & Balaras, E. 2006, J. Comput. Phys., 215:1, 12-40
- Yee, H. C., Vinokur, M., and Djomehri, M. J. 2000, JCP, 162, 33
- Yee, K. S., IEEE Trans. Antenna Propagation, 1966, AP-14, 302
- Youngs, D. L., 1984, AWRE/44/92/35
- Zachary, A., Malagoli, A., Colella, P. 1994, SIAM, 15, 263
- Zalesak, S. T. 1987, in Advances in Computer Methods for Partial Differential Equations VI, eds. Vichnevetsky, R. and Stepleman, R. S. (IMACS), 15
- Zel'dovich, Ya. B. 1970, A&A, 5, 84
- Zel'dovich, Ya. B. and Raizer, Yu. P. 2002, Physics of Shock Waves and High-Temperature Hydrodynamic Phenomena, Dover
- Zhang, W., Howell, L., Almgren, A., Burrows, A., & Bell, J. 2011, ApJS, 196, 20
- Zingale, M., Dursi, L. J., ZuHone, J., Calder, A. C., Fryxell, B., Plewa, T., Truran, J. W., Caceres, A., Olson, K., Ricker, P. M., Riley, K., Rosner, R., Siegel, A., Timmes, F. X., and Vladimirova, N. 2002, ApJS, 143, 539



# Runtime Parameters

## Burn

- algebra, 264, 267
- enucDtFactor, 12, 270
- odeStepper, 264, 269
- useBurnTable, 269
- useShockBurn, 269

## Conductivity

- useConductivity, 376

## Cosmology

- CosmologicalConstant, 373
- MaxScaleChange, 373
- OmegaMatter, 373
- OmegaRadiation, 373

## Diffuse

- useDiffuse, 330
- useDiffuseTherm, 330
- useDiffuseVisc, 330

## Driver

- dr\_numPosdefVars, 104
- dr\_usePosdefComputeDt, 104
- dtinit, 103
- dtmax, 103, 366
- dtmin, 103
- meshCopyCount, 110
- nend, 176
- restart, 177
- tmax, 175
- tstep\_change\_factor, 103

## EnergyDeposition

- ed\_adjustBeamsTargetIntensity, 308, 310
- ed\_adjustBySymmetryX, 308
- ed\_adjustBySymmetryY, 308
- ed\_adjustBySymmetryZ, 308
- ed\_adjustInitialRaySpeed, 306
- ed\_beamsCheckExpandX, 308
- ed\_beamsCheckExpandY, 308
- ed\_beamsCheckExpandZ, 308
- ed\_cellStepTolerance, 308
- ed\_cellTimeEnergyDeposition, 308
- ed\_cellWallThicknessFactor, 308
- ed\_computeGradNeleP, 308
- ed\_computeGradNeleR, 308

- ed\_computeGradNeleT, 308
- ed\_computeGradNeleX, 308
- ed\_computeGradNeleY, 308
- ed\_computeGradNeleZ, 309
- ed\_createRaysExpandX, 308, 309
- ed\_createRaysExpandY, 308, 309
- ed\_createRaysExpandZ, 308, 309
- ed\_crossSectionFunctionType\_n, 306
- ed\_cubicInterpolationZeroDerv, 309
- ed\_enforcePositiveNele, 309
- ed\_enforcePositiveTele, 309
- ed\_gaussianCenterMajor\_n, 306
- ed\_gaussianCenterMinor\_n, 307
- ed\_gaussianExponent\_n, 307
- ed\_gaussianRadiusMajor\_n, 307
- ed\_gaussianRadiusMinor\_n, 307
- ed\_gradOrder, 309
- ed\_gridDeltaSemiAxisMajor\_n, 307
- ed\_gridDeltaSemiAxisMinor\_n, 307
- ed\_gridnAngularTics\_n, 307
- ed\_gridnRadialTics\_n, 307
- ed\_gridnSemiAxisMajorTics\_n, 307
- ed\_gridnSemiAxisMinorTics\_n, 307
- ed\_gridType\_n, 307
- ed\_ignoreBoundaryCondition\_n, 307
- ed\_laser3Din2D, 279, 309
- ed\_laser3Din2DwedgeAngle, 309
- ed\_laserIOMaxNumberOfPositions, 311
- ed\_laserIOMaxNumberOfRays, 311
- ed\_lensSemiAxisMajor\_n, 307
- ed\_lensX\_n, 307
- ed\_lensY\_n, 307
- ed\_lensZ\_n, 307
- ed\_maxRayCount, 309
- ed\_numberOfBeams, 309
- ed\_numberOfPulses, 306, 309
- ed\_numberOfRays\_n, 307
- ed\_numberOfSections\_n, 306
- ed\_power\_n\_i, 306
- ed\_powerStepTolerance, 309
- ed\_printBeams, 309
- ed\_printMain, 309
- ed\_printPulses, 309
- ed\_printRays, 309

- ed\_pulseNumber\_n, 307
- ed\_rayZeroPower, 309
- ed\_RungeKuttaMethod, 310
- ed\_saveOutOfDomainRays, 310
- ed\_semiAxisMajorTorsionAngle\_n, 307
- ed\_semiAxisMajorTorsionAxis\_n, 307
- ed\_targetSemiAxisMajor, 308, 310
- ed\_targetSemiAxisMajor\_n, 307
- ed\_targetSemiAxisMinor\_n, 307
- ed\_targetX, 310
- ed\_targetX\_n, 308
- ed\_targetY\_n, 308
- ed\_targetZ\_n, 308
- ed\_time\_n\_i, 306
- ed\_useLaserIQ, 311
- ed\_useRayCoords2GetBlockID, 310
- ed\_velocityStepTolerance, 310
- ed\_wavelength\_n, 308
- threadRayTrace, 310
- useEnergyDeposition, 310
- Eos
  - eintSwitch, 218, 228
  - eos\_coulumbMult, 260
  - eos\_maxNewton, 260
  - eos\_singleSpeciesA, 253, 259
  - eos\_singleSpeciesZ, 253, 259
  - eos\_tolerance, 260
  - eosModeInit, 392, 393
  - gamma, 253, 259, 480, 490, 498
  - gammaEle, 253
  - gammaIon, 253
  - gammaRad, 253
- eos\_specSesDblFmt, 258
- Gravity
  - grav\_boundary\_type, 114, 115, 346, 347
  - grav\_boundary\_type\_x, 346, 347
  - grav\_boundary\_type\_y, 346, 347
  - grav\_boundary\_type\_z, 346, 347
  - grv\_bhAccErr, 344–346
  - grv\_bhEwaldAlwaysGenerate, 347
  - grv\_bhEwaldFieldNxV42, 346, 347
  - grv\_bhEwaldFieldNyV42, 346, 347
  - grv\_bhEwaldFieldNzV42, 346, 347
  - grv\_bhEwaldFName, 347
  - grv\_bhEwaldFNameAccV42, 347
  - grv\_bhEwaldFNamePotV42, 347
  - grv\_bhEwaldNPer, 346, 347
  - grv\_bhEwaldNRefV42, 346, 347
  - grv\_bhEwaldSeriesN, 346, 347
  - grv\_bhExtrnPotCenterX, 346, 347
  - grv\_bhExtrnPotCenterY, 346, 347
  - grv\_bhExtrnPotCenterZ, 346, 347
  - grv\_bhExtrnPotFile, 346, 347
  - grv\_bhExtrnPotType, 346, 347
  - grv\_bhLinearInterpolOnlyV42, 346, 347
  - grv\_bhMAC, 345, 346
  - grv\_bhMPDegree, 345, 346
  - grv\_bhNewton, 345, 346
  - grv\_bhUseExternalPotential, 347
  - grv\_bhUsePoissonPotential, 347
  - grv\_bhUseRelAccErr, 344–346
  - grv\_useExternalPotential, 346
  - useGravity, 345
- Grid
  - convertToConsvdForMeshCalls, 71, 72, 122, 126
  - convertToConsvdInMeshInterp, 71, 72, 122, 126
  - derefine\_cutoff\_#, 123
  - eosMode, 227
  - eosModeInit, 227
  - flux\_correct, 117
  - geometry, 164
  - gr\_bhPhysMACComm, 155, 345
  - gr\_bhPhysMACTW, 155, 345
  - gr\_bhTreeLimAngle, 155
  - gr\_bhTreeSafeBox, 148, 155
  - gr\_bhTWMaxQueueSize, 155
  - gr\_bhUnifiedTreeWalk, 148
  - gr\_bhUseUnifiedTW, 155
  - gr\_lrefineMaxRedDoByLogR, 124
  - gr\_lrefineMaxRedDoByTime, 125
  - gr\_lrefineMaxRedLogBase, 125
  - gr\_lrefineMaxRedRadiusFact, 125
  - gr\_lrefineMaxRedTimeScale, 125
  - gr\_lrefineMaxRedTRef, 125
  - gr\_pfftDiffOpDiscretize, 156
  - gr\_pmrpForceConsistency, 12
  - iGridSize, 118
  - interpol\_order, 121, 152
  - jGridSize, 118
  - kGridSize, 118
  - lrefine\_max, 124
  - max\_particles\_per\_blk, 123
  - mg\_maxCorrections, 156
  - mg\_MaxResidualNorm, 156
  - mg\_printNorm, 156
  - mpole\_2DSymmetryPlane, 153, 154
  - mpole\_3DAxisymmetry, 153, 154
  - mpole\_DumpMoments, 153, 154
  - mpole\_IgnoreInnerZone, 153, 154
  - mpole\_InnerZoneResolution, 153, 154
  - mpole\_InnerZoneSize, 153, 154
  - mpole\_Lmax, 153, 154
  - mpole\_lmax, 156, 539, 540
  - mpole\_MaxRadialZones, 154
  - mpole\_PrintRadialInfo, 153, 154



- mpole\_subSample, 140
- mpole\_ZoneExponent\_n, 154
- mpole\_ZoneRadiusFraction\_n, 154
- mpole\_ZoneScalar\_n, 154
- mpole\_ZoneType\_n, 154
- Nblockx, 113, 498
- nblockx, 112
- Nblocky, 113, 498
- nblocky, 112
- Nblockz, 113
- nblockz, 112
- pfft\_setupOnce, 136
- refine\_cutoff\_#, 123
- refine\_filter\_#, 123
- refine\_on\_particle\_count, 123
- refine\_var\_#, 122, 123
- smlrho, 539
- tagradius, 126
- xl\_boundary\_type, 113, 114, 498
- xmax, 112, 493, 498
- xmin, 112, 493
- xr\_boundary\_type, 498
- ymax, 112, 493, 498
- ymin, 112, 493
- zmax, 112
- zmin, 112

#### Hydro

- CFL, 104
- cfl, 103
- chomboLikeUpdateSoln, 228
- hy\_cflFallbackFactor, 224
- hy\_eosModeAfter, 227
- hybrid\_riemann, 220
- HydroMain, 219
- ppm\_modifystates, 220
- updateHydroFluxes, 559
- use\_avisc, 7
- use\_cma\_flattening, 220
- use\_steepening, 220

#### IO

- alwaysComputeUserVars, 189
- alwaysRestrictCheckpoint, 189
- checkpointFileIntervalStep, 194
- checkpointFileNumber, 177
- chkGuardCellsInput, 183
- chkGuardCellsOutput, 183
- outputSplitNum, 189
- particleFileNumber, 177
- plot\_grid\_var\_1, 182
- plot\_grid\_var\_2, 182
- plot\_var\_1, 177
- plot\_var\_2, 177

- plotFileNumber, 195
- plotfileNumber, 177

#### Ionize

- dneimax, 271
- dneimin, 271
- tneimax, 271
- tneimin, 271

#### Logfile

- log\_file, 399

#### Opacity

- op\_absorbScale, 381
- op\_emitScale, 381
- op\_transScale, 381

#### Particles

- initializeParticleAtRestart, 364
- particle\_attribute\_1, 365
- particle\_attribute\_2, 365
- pt\_dtChangeToleranceDown, 359
- pt\_dtChangeToleranceUp, 359
- pt\_dtFactor, 364
- pt\_initialXMin, 364
- pt\_numX, 364
- pt\_numx, 194
- pt\_nump, 194
- pt\_picPpc\_1, 354
- useParticles, 364

#### PhysicalConstants

- pc\_unitsBase, 206

#### ProtonEmission

- pem\_appendOldDetectorFiles, 433
- pem\_cellStepTolerance, 433
- pem\_cellWallThicknessFactor, 433
- pem\_detailedTiming, 433
- pem\_detectorCenter[X,Y,Z]\_n, 432
- pem\_detectorFileNameTimeStamp, 433
- pem\_detectorNormal[X,Y,Z]\_n, 432
- pem\_detectorPinholeDist2Det\_n, 432
- pem\_detectorPinholeRadius\_n, 432
- pem\_detectorSideLength\_n, 432
- pem\_detectorSideTiltingAngle\_n, 432
- pem\_detectorSideTiltingAxis\_n, 432
- pem\_detectorXYwriteFormat, 433
- pem\_emissionAmplificationFactor, 433
- pem\_emissionBoxCornerL[X,Y,Z]\_n, 432
- pem\_emissionBoxCornerU[X,Y,Z]\_n, 432
- pem\_emissionConeCenter[X,Y,Z], 433
- pem\_emissionConeHalfApexAngle, 433
- pem\_ignoreElectricalField, 433
- pem\_ignoreMagneticField, 433
- pem\_maxProtonCount, 433
- pem\_numberOfDetectors, 432
- pem\_numberOfEmissionBoxes, 432

- pem\_opaqueBoundaries, 433
- pem\_printDetectors, 433
- pem\_printMain, 433
- pem\_printProtons, 433
- pem\_printProtonSources, 433
- pem\_protonDeterminism, 434
- pem\_RungeKuttaMethod, 434
- pem\_screenProtonBucketSize, 434
- pem\_useParabolicApproximation, 434
- threadProtonTrace, 434
- useProtonEmission, 434
- ProtonImaging
  - pi\_beamApertureAngle\_n, 419
  - pi\_beamCapsule[X,Y,Z]\_n, 419
  - pi\_beamCapsuleRadius\_n, 419
  - pi\_beamDetector\_n, 419
  - pi\_beamNoBoundaryCondition\_n, 419
  - pi\_beamNumberOfProtons\_n, 419
  - pi\_beamProtonEnergy\_n, 419
  - pi\_beamTarget[X,Y,Z]\_n, 419
  - pi\_beamTime2Launch\_n, 419
  - pi\_cellStepTolerance, 420
  - pi\_cellWallThicknessFactor, 420
  - pi\_detectorAlignWRTbeamNr\_n, 419
  - pi\_detectorCenter[X,Y,Z]\_n, 419
  - pi\_detectorDGwriteFormat, 420
  - pi\_detectorDist2BeamCapsule\_n, 419
  - pi\_detectorFileNameTimeStamp, 420
  - pi\_detectorNormal[X,Y,Z]\_n, 419
  - pi\_detectorPinholeDist2Det\_n, 419
  - pi\_detectorPinholeRadius\_n, 419
  - pi\_detectorSideLength\_n, 419
  - pi\_detectorSideTiltingAngle\_n, 419
  - pi\_detectorSideTiltingAxis\_n, 419
  - pi\_detectorXYwriteFormat, 420
  - pi\_flagDomainMissingProtons, 420
  - pi\_ignoreElectricalField, 420
  - pi\_IOaddDetectorScreens, 420
  - pi\_IOaddProtonsCapsule2Domain, 420
  - pi\_IOaddProtonsDomain2Screen, 420
  - pi\_IOmaxBlockCrossingNumber, 420
  - pi\_IOnumberOfProtons2Plot, 420
  - pi\_maxProtonCount, 420
  - pi\_numberOfBeams, 419
  - pi\_numberOfDetectors, 419
  - pi\_opaqueBoundaries, 420
  - pi\_printBeams, 420
  - pi\_printDetectors, 420
  - pi\_printMain, 420
  - pi\_printProtons, 420
  - pi\_protonDeterminism, 421
  - pi\_recordOffScreenProtons, 421
  - pi\_relativisticTracing, 421
  - pi\_RungeKuttaMethod, 421
  - pi\_screenProtonBucketSize, 421
  - pi\_screenProtonDiagnostics, 421
  - pi\_timeResolvedProtonImaging, 421
  - pi\_useIOprotonPlot, 421
  - pi\_useParabolicApproximation, 421
  - threadProtonTrace, 421
  - useProtonImaging, 421
- RadTrans
  - rt\_mgdFlCoef, 390
  - rt\_mgdFlMode, 390
  - rt\_mgdNumGroups, 391
  - rt\_planckIntMethod, 390
  - rt\_useMGD, 391
- Simulation
  - sim\_maxTolCoeff0, 365
  - sim\_maxTolCoeff1, 365
  - sim\_maxTolCoeff2, 365
  - sim\_schemeOrder, 365
  - sim\_smlRho, 538
- Stir
  - st\_computedDt, 274
  - st\_decay, 272
  - st\_energy, 272
  - st\_infilename, 274
  - useStir, 274
- ThomsonScattering
  - Beam Runtime Parameters, 446
  - Detector Runtime Parameters, 446
  - threadThscRayTrace, 442
  - thsc\_beamDetector\_n, 446
  - thsc\_beamLens[X,Y,Z]\_n, 446
  - thsc\_beamTarget[X,Y,Z]\_n, 446
  - thsc\_cellTimeEnergyDeposition, 441
  - thsc\_cellWallThicknessFactor, 441
  - thsc\_detectorApertureAngle\_n, 446
  - thsc\_detectorCenter[X,Y,Z]\_n, 446
  - thsc\_detectorDGwriteFormat, 447
  - thsc\_detectorFileNameTimeStamp, 447
  - thsc\_detectorLaserBeamNumber\_n, 440
  - thsc\_detectorLensSAMajor\_n, 440
  - thsc\_detectorLensX\_n, 440
  - thsc\_detectorLensY\_n, 440
  - thsc\_detectorLensZ\_n, 440
  - thsc\_detectorNoRayDeflection\_n, 440
  - thsc\_detectorNoRayPowerLoss\_n, 440
  - thsc\_detectorNumberOfRays\_n, 440
  - thsc\_detectorNumberOfSpecPoints\_n, 440
  - thsc\_detectorSAMajorTorsionAngle\_n, 440
  - thsc\_detectorSAMajorTorsionAxis\_n, 440
  - thsc\_detectorScreenSpectralFlux\_n, 440
  - thsc\_detectorScreenWavelengths\_n, 440
  - thsc\_detectorSideLength\_n, 446

thsc.detectorSkipTimeResolve\_n, 441  
 thsc.detectorTarget[X,Y,Z]\_n, 446  
 thsc.detectorTargetRadius\_n, 446  
 thsc.detectorTargetSAMajor\_n, 441  
 thsc.detectorTargetSAMinor\_n, 441  
 thsc.detectorTargetX\_n, 441  
 thsc.detectorTargetY\_n, 441  
 thsc.detectorTargetZ\_n, 441  
 thsc.detectorUseRelativityTerm\_n, 441  
 thsc.detectorWavelengthMax\_n, 441  
 thsc.detectorWavelengthMin\_n, 441  
 thsc.detectorWriteFormatX, 441  
 thsc.detectorWriteFormatY, 441  
 thsc.detectorXYwriteFormat, 447  
 thsc.enforcePositiveNele, 441  
 thsc.enforcePositiveTele, 441  
 thsc.laserCrossSecFuncType\_n, 439  
 thsc.laserGaussCenterMajor\_n, 439  
 thsc.laserGaussCenterMinor\_n, 439  
 thsc.laserGaussExponent\_n, 439  
 thsc.laserGaussRadiusMajor\_n, 439  
 thsc.laserGaussRadiusMinor\_n, 439  
 thsc.laserInitialRaySpeed\_n, 439  
 thsc.laserLensSAMajor\_n, 439  
 thsc.laserLensX\_n, 439  
 thsc.laserLensY\_n, 439  
 thsc.laserLensZ\_n, 439  
 thsc.laserNoEnergyDeposition\_n, 439  
 thsc.laserNumberOfRays\_n, 439  
 thsc.laserPolarizationAngle\_n, 439  
 thsc.laserPulseNumber\_n, 439  
 thsc.laserSAMajorTorsionAngle\_n, 439  
 thsc.laserSAMajorTorsionAxis\_n, 439  
 thsc.laserTargetSAMajor\_n, 439  
 thsc.laserTargetSAMinor\_n, 439  
 thsc.laserTargetX\_n, 440  
 thsc.laserTargetY\_n, 440  
 thsc.laserTargetZ\_n, 440  
 thsc.laserWavelength\_n, 440  
 thsc.maxRayCount, 442, 446, 447  
 thsc.numberOfBeams, 446  
 thsc.numberOfDetectors, 440, 446  
 thsc.numberOfLaserBeams, 439  
 thsc.numberOfPulses, 438  
 thsc.numberOfSections\_n, 438  
 thsc.power\_n\_i, 438  
 thsc.printBeams, 447  
 thsc.printDetectors, 442, 447  
 thsc.printEnergyInfo, 442  
 thsc.printLaserBeams, 442  
 thsc.printMain, 442, 447  
 thsc.printPulses, 442, 447  
 thsc.printRays, 442  
 thsc.printSpecies, 442

thsc.rayDeterminism, 442  
 thsc.rayZeroPower, 442  
 thsc.recordOffScreenRays, 447  
 thsc.screenRayBucketSize, 447  
 thsc.screenRayDiagnostics, 447  
 thsc.spectrumFileIntervalStep, 444, 445  
 thsc.spectrumFileIntervalTime, 444, 445  
 thsc.spectrumNpts\_N, 445  
 thsc.time\_n\_i, 438  
 useThomsonScattering, 442, 447

#### Timers

eachProcWritesSummary, 406  
 writeStatSummary, 406

#### Turb

turb.c2, 325  
 turb.stepSize, 325

#### Viscosity

useViscosity, 379  
 visc.whichCoefficientIsConst, 379

#### XrayImaging

threadXrayTrace, 452  
 useXrayImaging, 452  
 xray\_3Din2D, 452  
 xray.cellWallThicknessFactor, 452  
 xray.detectorAlignWRTorigin\_n, 451  
 xray.detectorCenter[X,Y,Z]\_n, 451  
 xray.detectorDGwriteFormat, 452  
 xray.detectorEnergy\_nFraction\_i, 451  
 xray.detectorEnergy\_nLevel\_i, 451  
 xray.detectorEnergyLevelCount\_n, 451  
 xray.detectorFileNameTimeStamp, 452  
 xray.detectorNormal[X,Y,Z]\_n, 451  
 xray.detectorOrigin[X,Y,Z]\_n, 451  
 xray.detectorPerpXrays\_n, 451  
 xray.detectorResolution\_n, 451  
 xray.detectorSideLength\_n, 451  
 xray.detectorSideTiltingAngle\_n, 451  
 xray.detectorSideTiltingAxis\_n, 451  
 xray.detectorXYwriteFormat, 452  
 xray.maxXrayCount, 452  
 xray.numberOfDetectors, 451  
 xray.printDetectors, 452  
 xray.printMain, 452  
 xray.printSpecies, 452  
 xray.printXrays, 452  
 xray.screenXrayBucketSize, 452  
 xray.XrayDeterminism, 452



# FLASH4 API

## diagnostics

- ProtonEmission
  - ProtonEmission, 429
- ProtonImaging
  - ProtonImaging, 411, 415, 418
- XrayImaging
  - XrayImaging, 449

## Driver

- Driver\_abortFlash, 103, 181, 195, 402
- Driver\_abortFlashC, 103
- Driver\_computeDt, 103
- Driver\_evolveFlash, 45, 48, 100, 103, 247, 330, 356
- Driver\_finalizeFlash, 102
- Driver\_getDt, 102
- Driver\_getElapsedWCTime, 102
- Driver\_getNStep, 102
- Driver\_getSimTime, 102
- Driver\_initFlash, 29, 46, 100, 103, 175, 194, 206
- Driver\_initParallel, 100
- Driver\_sendOutputData, 181
- Driver\_verifyInitDt, 103

## Grid

- Grid\_advanceDiffusion, 156
- Grid\_applyBCEdge, 115, 116
- Grid\_applyBCEdgeAllUnkVars, 115, 116
- Grid\_bcApplyToRegion, 115, 116
- Grid\_bcApplyToRegionSpecialized, 115, 116
- Grid\_conserveFluxes, 332
- Grid\_fillGuardCells, 44, 116, 120, 128, 131
- Grid\_getBlkBoundingBox, 120
- Grid\_getBlkCenterCoords, 120, 121, 165
- Grid\_getBlkCornerID, 121
- Grid\_getBlkData, 44, 127, 165
- Grid\_getBlkIndexLimits, 33, 127
- Grid\_getBlkPhysicalSize, 120, 121, 165
- Grid\_getBlkPtr, 127
- Grid\_getBlkRefineLevel, 120
- Grid\_getBlkType, 120
- Grid\_getBlPtr, 380
- Grid\_getCellCoords, 33, 44, 121, 164
- Grid\_getDeltas, 165

- Grid\_getGeometry, 164
- Grid\_getListOfBlocks, 45, 127
- Grid\_getPlaneData, 127, 165
- Grid\_getPointData, 127, 165
- Grid\_getRowData, 127, 165
- Grid\_getSingleCellVol, 165
- Grid\_init, 112, 127
- Grid\_initDomain, 112, 127
- Grid\_mapParticlesToMesh, 347, 363
- Grid\_markRefineDerefine, 27, 124
- Grid\_markRefineSpecialized, 124
- Grid\_pfft, 136
- Grid\_pfftFinalize, 136
- Grid\_pfftInit, 136
- Grid\_putBlkData, 44, 127
- Grid\_restrictAllLevels, 12
- Grid\_sendOutputData, 128, 181
- Grid\_solvePoisson, 151
- Grid\_updateRefinement, 44, 45, 120, 128

## IO

- IO\_getPrevScalar, 181
- IO\_getScalar, 181
- IO\_init, 175, 195
- IO\_output, 175, 177, 189
- IO\_readUserArray, 181
- IO\_setScalar, 181, 184
- IO\_writeCheckpoint, 175, 189
- IO\_writeIntegralQuantities, 180, 365
- IO\_writeParticles, 179
- IO\_writePlotfile, 177, 189
- IO\_writeUserArray, 181

## monitors

- Logfile
  - Logfile.init, 399
  - Logfile.stamp, 402
  - Logfile.stampMessage, 402
  - Logfile.writeSummary, 403
- Timers
  - Timers.create, 406
  - Timers.getSummary, 406
  - Timers.init, 406
  - Timers.reset, 406

- Timers\_start, 406
- Timers\_stop, 406
- Multispecies
  - Multispecies\_getAvg, 202
  - Multispecies\_getProperty, 201
  - Multispecies\_getSum, 202
  - Multispecies\_getSumFrac, 202
  - Multispecies\_getSumInv, 202
  - Multispecies\_getSumSqr, 202
  - Multispecies\_list, 202
  - Multispecies\_setProperty, 201
  - Multispecies\_unitTest, 203
- Particles
  - Particles\_advance, 351, 356, 365
  - Particles\_computeDt, 356
  - Particles\_dump, 365
  - Particles\_init, 365
  - Particles\_initPositions, 362, 363
  - Particles\_mapToMeshOneBlk, 363
  - Particles\_specifyMethods, 80, 362
  - Particles\_unitTest, 365
  - Particles\_updateAttributes, 364, 365
- PhysicalConstants
  - PhysicalConstants\_get, 206
  - PhysicalConstants\_init, 206
  - PhysicalConstants\_list, 206
  - PhysicalConstants\_listUnits, 206
  - PhysicalConstants\_unitTest, 206, 207
- physics
  - Cosmology
    - Cosmology\_cdmPowerSpectrum, 373
    - Cosmology\_computeDeltaCrit, 374
    - Cosmology\_computeDt, 373
    - Cosmology\_computeVariance, 374
    - Cosmology\_getRedshift, 372
    - Cosmology\_massToLength, 374
    - Cosmology\_redshiftToTime, 374
    - Cosmology\_solveFriedmannEqn, 373
  - Diffuse
    - Diffuse\_solveScalar, 331
    - Diffuse\_therm, 559
  - Eos
    - Eos, 72, 127, 249, 250, 253, 254, 260
    - Eos\_getData, 72
    - Eos\_init, 259
    - Eos\_putData, 72
    - Eos\_wrapped, 7, 72, 73, 249, 254, 260
  - Gravity
    - Gravity\_accelOneRow, 152, 344, 345
    - Gravity\_bhFillBotNode, 147
    - Gravity\_bhNodeContrib, 148, 344
    - Gravity\_init, 114
    - Gravity\_potentialListOfBlocks, 345
  - Hydro
    - Hydro\_computeDt, 103
  - RadTrans
    - RadTrans\_mgdEFromT, 393
    - RadTrans\_sumEnergy, 395
  - sourceTerms
    - EnergyDeposition, 567
  - RuntimeParameters
    - RuntimeParameters\_get, 194, 195
    - RuntimeParameters\_getPrev, 181
    - RuntimeParameters\_mapStrToInt, 113, 114
    - RuntimeParameters\_set, 195
  - Simulation
    - Simulation\_adjustEvolution, 394
    - Simulation\_defineDomain, 112, 114, 115, 483, 498
    - Simulation\_getVarnametype, 111
    - Simulation\_init, 29
    - Simulation\_initBlock, 127, 254, 393
    - Simulation\_initSpecies, 198, 201, 260, 264
    - Simulation\_mapIntToStr, 80
    - Simulation\_mapParticlesVar, 80
    - Simulation\_mapStrToInt, 80

# Index

- `.dump_checkpoint`, *see* IO
- `.dump_restart`, *see* IO
- subunit, 46
  - primary, 46
- adaptive mesh, 118
- adiabatic indices, 250
- angular coordinates, 112
- block, 109
- boundary conditions, 113
- computational domain, 112
- `Config`, 27, 28
- Cpp & Python units, 209
- Docker, 20
- double precision, 33
- energy deposition, 276
  - laser: 3D ray tracing in 2D domain, 291
  - laser: beam cross section power structure, 289
  - laser: beam definition and setup, 283
  - laser: Cell average algorithm, 280
  - laser: Cubic interpolation with piecewise parabolic ray tracing, 282
  - laser: Cubic interpolation with Runge Kutta integration, 282
  - laser: energy density, 279
  - laser: implementation details (algorithm), 276, 279, 296
  - laser: power deposition, 277
  - laser: proper beam placement, 285, 293
  - laser: pulse definition, 283
  - laser: ray definition and setup, 286, 290, 294
  - laser: ray grid inside beam, 286, 288, 289
  - laser: ray tracing in 3D cylindrical domains, 297
  - laser: synchronous and asynchronous ray tracing, 302
- equation of state, 249, 250
  - `Eos`, 260
  - `Eos_wrapped`, 260
  - gamma-law, 250
  - Helmholtz, 250, 251
  - multi-gamma, 250, 251
  - rh-d-gamma, 250
- `fidlr`
  - `xflash3`, 22
- FIXEDBLOCKSIZE mode, 90, 96, 117
- Flash.h, 89
- `flash.par`, 19, 36, 177, 400
- FLASH\_MPI1, 11
- flux conservation, 120
- geometry, *see* mesh
- `gmake`
  - `Makefile.h`, 18
- grid
  - boundary, *see* boundary conditions
  - data averaging, 120
  - guardcells, *see* guardcells
  - interpolation, 117, 120, 121, 152, 162, 168
    - order, 121
  - prolongation, 121, 168
  - restriction, 121, 168
  - `xmin/xmax`, 112
  - `ymin/ymax`, 112
  - `zmin/zmax`, 112
- `Grid_getCellCoords`, 33
- guardcells, 109
- H5\_USE\_16\_API, 78, 184
- HDF5, *see* IO
- I/O, 172
- IDL, 18
- incompressible, 247
- IO
  - `.dump_checkpoint`, 176
  - `.dump_particle_file`, 179
  - `.dump_plotfile`, 178
  - `.dump_restart`, 176
  - checkpoint file, 175
  - forced Plotfile, 178
  - HDF5, 183
  - output file names, 182
  - particle file, 179
  - plotfile, 177

- kernel, 47
- make
  - make clean, 18
  - parallel build, 18
- gmake
  - make realclean, 18
- Makefile.h, 49
- materials
  - viscosity, *see* viscosity
- MAXBLOCKS, 52, 119
- memory, 118, 119
- mesh
  - geometry, 162
    - CARTESIAN, 164
    - Cartesian, 166
    - CYLINDRICAL, 164
    - cylindrical, 166
    - POLAR, 164
    - polar, 167
    - SPHERICAL, 164
    - spherical, 167
  - refinement criteria, 31
- Navier-Stokes, 247
- NONFIXEDBLOCKSIZE mode, 3, 90, 118, 127
- PARAMESH, 118
- paramesh, 110
- Physical Constants, 205
  - CGS,CGS, 205
  - MKS,MKS, 205
- proton emission, 429
  - proton detector screens, 431
  - proton emission boxes, 431
  - proton generation, 429
- proton imaging, 411
  - implementation details (algorithm), 412
- proton imaging: beam definition and setup, 415
- proton imaging: detector screen setup, 417
- proton imaging: moving protons through domain, 415
- proton imaging: proton definition and setup, 415
- proton imaging: proton specifications, 415
- proton imaging: recording protons on detector screen, 417
- proton imaging: time resolved proton imaging, 418
- Runge Kutta, 467
- sedov.dat, 22
- sedov.log, 22
- setup
  - +default, 66
  - [123]d, 52
  - append-parfiles, 56
  - auto, 51
  - cmake, 52
  - curvilinear, 53
  - debug, 52
  - defines, 53
  - fbs, 53
  - geometry, 53
  - gridinterpolation, 53, 152
  - index-reorder, 55
  - library, 57
  - makefile, 55
  - makehide, 55
  - maxblocks, 52
  - n[xyz]b, 52
  - noclobber, 55
  - nofbs, 53
  - objdir, 53
  - opt, 52
  - os, 55
  - parfile, 55
  - particlemethods, 56
  - portable, 56
  - parfile, 55
  - site, 56
  - tau, 57
  - test, 52
  - unit, 53
  - unitsfile, 56
  - verbose, 51
  - with-library, 57
  - with-unit, 53
  - without-library, 57
  - without-unit, 57
  - 3t, 61
  - 8wave, 59
  - cartesian, 60
  - chombo\_amr, 58
  - chombo\_ug, 58
  - cppapi, 62
  - cube16, 59
  - cube32, 59
  - cube64, 59
  - curv-pm2, 60
  - curvilinear, 60
  - cylindrical, 60
  - default, 57
  - ed\_maxBeams, 64
  - ed\_maxPulses, 64
  - fixedBlockSize, 63
  - gravMgrid, 61
  - gravMpole, 61
  - gravPfftNofbs, 61
  - Grid, 62



- grid, 58
- GridIndexOrder, 63
- hdf5, 57
- hdf5TypeIO, 58
- IO, 62
- io, 57
- laser, 62
- longrange, 61
- maxBlocks, 63
- mgd, 62
- mgd\_meshgroups, 64
- mpi1, 61
- mpi2, 61
- mpole, 60
- Mtmmmt, 64
- mtmmmt, 61
- nDim, 63
- newMpole, 61
- noDefaultMpole, 61
- nofbs, 60
- noio, 57
- nolog, 58
- noMgrid, 61
- npg, 60
- nxb, 63
- nyb, 63
- nzb, 63
- parallelIO, 57, 62
- ParameshLibraryMode, 63
- PfftSolver, 64
- pic, 62
- pm2, 58
- pm3, 58
- pm40, 58
- pm4dev, 58
- pm4dev\_clean, 58
- pnetcdf, 57
- pnetcdfTypeIO, 58
- polar, 60
- ptdens, 60
- ptio, 59
- pureHydro, 59
- python, 62
- rnf, 59
- serialIO, 57
- shortcuts, 183, 189
- species, 64
- spherical, 60
- spherical-pm2, 60
- SplitDriver, 64
- splitHydro, 59
- supportPPMUpwind, 59
- threadEosWithinBlock, 65
- threadHydroBlockList, 64
- threadHydroWithinBlock, 65
- threadMpoleBlockList, 65
- threadMpoleWithinBlock, 65
- threadRayTrace, 65
- ug, 58
- uhd, 59
- uhd3t, 61
- unsplitHydro, 59
- usm, 59
- usm3t, 62
- setup, 18, 49
- setup\_params, 37
- setups
  - example, 27
  - sedov, 18
- shortcuts, 174
- Simulation\_data, 29
- Simulation\_init, 29
- Simulation\_initBlock, 27, 32
  - creating new, 27
- Thomson scattering, 435
  - laser: pulse definition, 444
- Thomson scattering: beam definition and setup, 444
- Thomson scattering: detector setup, 445
- Thomson scattering: ray definition and setup, 444
- Thomson scattering: ray specifications, 445
- Thomson scattering: recording rays at detector, 445
- unit, 43
- variable
  - type
    - conserved, 71
    - mass-specific, 71
- variable type, 71
- viscosity, 379
  - spitzer, 379
- XFLASH3\_DIR, 22
- xray imaging, 449
  - implementation details (algorithm), 449
- xray imaging: detector screen setup and recording, 450
- xray imaging: X-ray definition, 450