Flash-X User's Guide

Version alpha
(last updated July 27, 2021)

Contents

Chapter 1

Introduction

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

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

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

1.1 What's New in `Flash-X`

This Guide describes the release version alpha of `Flash-X`. `Flash-X` includes all the well tested capabilities of `Flash-X`. There were a few modules in the official releases of Flash-X2 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 `Flash-X`. Those modules are not included in the official releases of `Flash-X` or `Flash-X`, 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 `Flash-X` 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, `Flash-X` closely follows `Flash-X`. The major changes from `Flash-X` are several new capabilities in both physics solvers and infrastructure. Major effort went into the design of the `Flash-X` architecture to ensure that the code can be easily modified and extended by internal as well as external developers. Each code unit in `Flash-X`, like in `Flash-X` has a well defined interface and follows the

rules for inheritance and encapsulation defined in `Flash-X`. One of the largest achievements of `Flash-X` was the separation of the discretized 'grid' architecture from the actual physics.

Because of the increasing importance of software verification and validation, the Flash code group has developed a test-suite application for `Flash-X`. The application is called FlashTest and can be used to setup, compile, execute, and test a series of Flash-X code simulations on a regular basis. FlashTest is available without a license and can be downloaded from the 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 `Flash-X` are backwards compatible with Flash-X2, and they are all compatible with `Flash-X`. The Flash code group has written extensive documentation detailing how to make the transition from `Flash-X` to `Flash-X` as smooth as possible. The user should follow the "Name changes from `Flash-X` to `Flash-X`" link on the for help on transitioning to `Flash-X` from `Flash-X`. The transition from `Flash-X` to `Flash-X` does not require much effort from the users except in any custom implementation they may have.

The new capabilities in `Flash-X` that were not included in `Flash-X` 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 Flash-X4-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
- Initializaton of 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-X initialization.
- The custom region code and corresponding Grid API routines have been removed.
- PARAMESH4DEV is now the default PARAMESH implementation.

The new capabilities in Flash-X4.2 ... Flash-X4.2.2 since Flash-X4.0.1 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 EnergyDeposition to be called once every n time steps.

The new capabilities in Flash-X4.3 since Flash-X4.2.2 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-X_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 Flash-X4.4 since Flash-X4.3 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 PhysicalCosntants˙init.F90 with an older version; the version from Flash-X4.3 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 implemention (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 aspreconditioner). 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 `Flash-X` 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 Flash-X4.5 since Flash-X4.4 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 **??** 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 `"vanLeeer1.5"` 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-X on Windows, Mac OS, and Linux distributions, using Docker. See **??** in the "Quick Start" chapter.

- **Flash-X 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-X parameters (found in `setup_params`) via categories;
  - Searching for Flash-X 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-X-Interface. Compiled executables for a number of operating systems can be found at http://flash.uchicago.edu/site/flashcode/Flash-X-Interface

- The **PRaLine** code for Proton Radiography Linear reconstruction is now included with the Flash-X 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@uchicago.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 Flash-X4.6 since Flash-X4.5 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 Flash-X4.6 to Flash-X4.6.1:

- 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.

Part I

Getting Started

Chapter 2

Quick Start

This chapter describes how to get up-and-running quickly with Flash-X 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-X source code distribution (as a Unix tar file). To request a copy of the distribution, click on the "Code Request" link on the 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-X.

- A F90 (Fortran 90) compiler and a C compiler. Most of Flash-X is written in F90. Flash-X has been tested with many Fortran compilers. For details of compilers and libraries, see the RELEASE-NOTES available in the Flash-X home directory.

- An installed copy of the Message-Passing Interface (MPI) library.

- 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-X 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-X units are described in detail in **??**.

- 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 **??**.

- 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 `Flash-X`.

- 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-X 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-X for quick start

To begin, unpack the Flash-X source code distribution.

```
tar -xvf Flash-XX.Y.tar
```

where *X.Y* is the Flash-X version number (for example, use `Flash-X4-alpha.tar` for Flash-X version 4-alpha, or `Flash-X3.1.tar` for Flash-X version 3.1). This will create a directory called `Flash-X`*X*`/`. Type '`cd Flash-X`*X*' to enter this directory. Next, configure the Flash-X source tree for the Sedov explosion problem using the `setup` scriptType

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

This configures Flash-X 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-X 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 **??** 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-X 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 `flash`*X* 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-X, but you will not be able to generate either a checkpoint or a plot file. You can setup Flash-X 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-X web page.

Flash-X 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-X. If it is not present, Flash-X 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 **??**.

```
# 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-X parameter file contents for the quick start example.

This example first instructs Flash-X to name output files appropriately (through the `basenm` parameter). Flash-X 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 Running Flash-X

We are now ready to run Flash-X. To run Flash-X 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 `Flash-X` executable accepts an optional command-line argument for the runtime parameters file. If "`-par_file` *filename*" is present, Flash-X 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-X 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-X 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 **??** for more information about IO outputs.

Flash-X 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-X and the sample problems and tools distributed with it.

Part II

The Flash-X Software System

Chapter 3

Overview of Flash-X architecture

The files that make up the Flash-X 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-X can be viewed as a collection of units, which are selectively grouped to form one application.

A typical Flash-X simulation requires only a subset of all of the units in the Flash-X 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-X `Config` files (described in **??**) as a guide. Thus, it is important to distinguish between the entire Flash-X source code and a given Flash-X application. the Flash-X 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-X 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-X 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 **??**, the `Simulation` unit allows the user to overwrite any of Flash-X'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-X units. These helpful features enable users to customize their applications, and are described in more detail below in **??** and online in The simulation unit also provides some useful interaces 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-X Transition

Why the name change from "modules" in `Flash-X` to "units" in `Flash-X`? The term "module" caused confusion among users and developers because it could refer both to a FORTRAN90 module and to the Flash-X-specific code entity. In order to avoid this problem, `Flash-X` started using the word "module" to refer exclusively to an F90 module, and the word "unit" for the basic Flash-X code component. Also, Flash-X no longer uses F90 modules to implement units. Fortran's limitation of one file per module is too restrictive for some of `Flash-X`'s units, which are too complex to be described by a single file. Instead, `Flash-X` 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.

3.1 Flash-X Inheritance

FORTRAN90 is not an object-oriented language like Java or C++, and as such does not implement those languages' characteristic properties of inheritance. But Flash-X 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

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 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.

3.2 Unit Architecture

A Flash-X 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 [[api reference]], [[api reference]], and [[api reference]], while [[api reference]] and
[[api reference]] are examples of API routines which modify data in the `Grid` unit.

A unit can have more than one implementation of its API. The `Grid` Unit, for example, has both an **Adaptive Grid** and a **Uniform Grid** implementation. Although the implementations are different, they both conform to the `Grid` API, and therefore appear the same to the outside units. This feature allows users

to easily swap various unit implementations in and out of a simulation without affecting they way other units communicate. Code does not have to be rewritten if the users decides to implement the uniform grid instead of the adaptive grid.

3.2.1 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 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 [[api reference]].

```
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-X 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 [[api reference]] 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 [[api reference]] 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-X Transition

While the concept of "null" or "stub" functions existed in `Flash-X`, `Flash-X` 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 `Flash-X` is essentially the same as that of Flash-X2, but the introduction of a formal naming convention has clarified it and made it easier to follow. The complete API can be found online at http://flash.uchicago.edu/site/flashcode/user_support/.

3.2.2 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 *Unit*`Main`, 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-X exploits the inheritance rules described in 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 `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 **??**. The kernels are described below in **??**.

3.2.3 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 [[api reference]] 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 *Unit*`Main`, 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, &
```

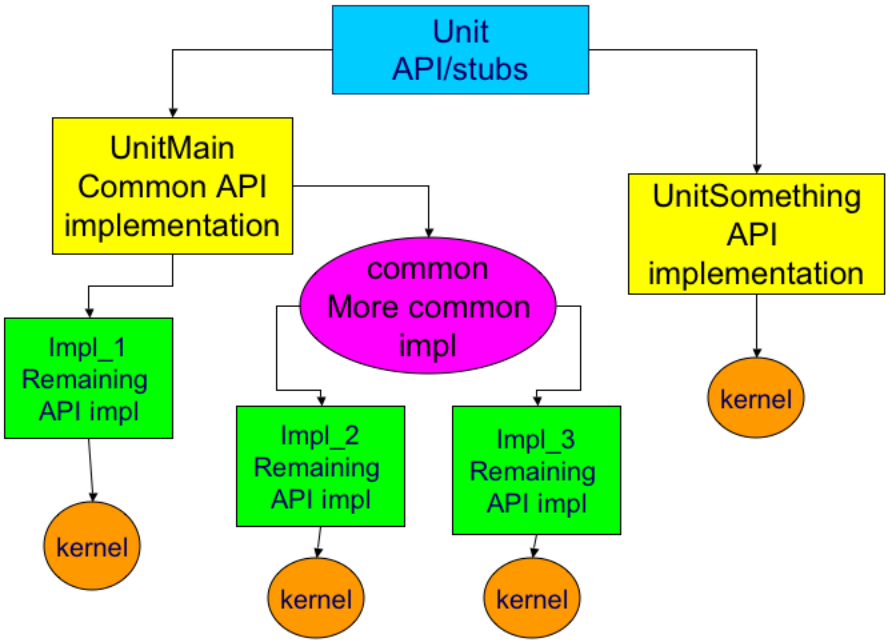

Figure 3.1: The unit hierarchy and inheritance.

```
      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-X run. The finalization routines might deallocate space or write out completion messages.

3.2.4 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-X. The kernels are not required to follow either the naming convention or the inheritance rules of the Flash-X 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-X unit hierarchy. This arrangement allows Flash-X 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 **??**). 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.

3.3 Unit Test Framework

In keeping with good software practice, `Flash-X` 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-X 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 **??**. The `Eos` unit test needs its own version of the routine [[api reference]] 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-X 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 4

The Flash-X configuration script (setup)

The setup scriptdirectory, provides the primary command-line interface to configuring the Flash-X source code. It is important to remember that the Flash-X 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-X source tree and link necessary files for a given application to the object/ directory
- find the target Makefile.h
- generate the Makefile that will build the Flash-X 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 Flash-X architecture interact may be found in ??. 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-X is running.[1] Failing this, it looks in sites/Prototypes/ for a directory with the same name as the output of the uname command. The site and operating system type can be overridden with the -site and -ostype command-line options to the setup command. Only one of these options can be used at one time. The directory for each site and operating system type contains a makefile fragment Makefile.h that sets command names, compiler flags, library paths, and any replacement or additional source files needed to compile Flash-X 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-X application. This file is automatically generated when the user commonly provides the command-line -auto option, although it may be assembled by hand. When -auto option is used, the setup script starts with the Config file of the problem specified, finds its REQUIRED units and then works its way through their Config files. This process continues until all the dependencies are met and a self-consistent set of units has been found. At the end of this automatic generation, the Units file is created and placed in the object/ directory, where it can be edited if necessary. setup also creates the master makefile (object/Makefile) and several FORTRAN include files that are needed by the code in order to parse the runtime parameters. After running setup, the user can create the Flash-X executable by running make in the object directory.

[1] if a machine has multiple hostnames, setup tries them all

Note that the Flash-X 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-X Transition

In `Flash-X`, the `Units` file was located in the Flash-X root directory. In `Flash-X`, 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. `Flash-X` 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 **??**. For detailed examples of I/O shortcuts please see **??**in the I/O chapter.

Reduce compilation time

- To reuse compiled code when changing setup configurations, use the `-noclobber` setup option. For details see **??**.

4.1 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. **??** explains how setup variables can be created and used.

Table 4.1: List of Commonly Used Setup Arguments

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

?? shows a list of some of the basic setup arguments that every Flash-X user should know about. A comprehensive list of all setup arguments can be found in **??** alongside more detailed descriptions of these options.

4.2 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 [2]) that specifies the units to include. A sample `Units` file appears in **??**. 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.

[2]Formerly, (in Flash-X2) it was located in the Flash-X root directory

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

`-[123]d`

By default, `setup` creates a makefile which produces a Flash-X 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=#`

`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 count-

`[-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-X 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 `Flash-X` distribution directory.

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

`<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 `Flash-X`.

- **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 `Flash-X` 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 4.1: 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 `Flash-X` for compatibility with alternative grids. This is supported by both the Uniform Grid as well as PARAMESH.

`-makehide`

Ordinarily, the commands being executed during compilation of the Flash-X 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` .

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

`-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-X. Internal libraries are those libraries whose code is included with Flash-X. 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.

`-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`

`+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`

`+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`

`+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`

`+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-X 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`

`+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 smilulation 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 smilulation 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`

`+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`

`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`

`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 paramters which include the species names.
Name: `species`
Type: `String`, comma seperated list of strings (*e.g.*, `species=air,sf6)`

`ed_maxPulses`

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

`ed_maxBeams`

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

`threadHydroBlockList`

This is used to turn on block list OPENMP threading of hydro routines.
Name: `threadHydroBlockList`
Type: `Boolean`
Values: `True, False`

`threadMpoleBlockList`

This is used to turn on block list OPENMP threading of the multipole routine.
Name: `threadMpoleBlockList`
Type: `Boolean`
Values: `True, False`

`threadRayTrace`

This is used to turn on block list OPENMP threading of Enery 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`

4.3 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 **??** for an example file.

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

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

Some of the more commonly used shortcuts are described below:

```
# 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 4.2: A sample `setup_shortcuts.txt` file

Table 4.5: 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

Table 4.6: 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

4.4 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[3]. 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

```
DEFAULT serial

USESETUPVARS parallelIO

IF parallelIO
   DEFAULT parallel
ENDIF
```

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

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

```
# Requires use of the Grid SetupVariable
USESETUPVARS Grid

DEFAULT paramesh

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

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

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

[3] All non-integral values not equal to True/False/Yes/No/On/Off are considered to be string values

4.5 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.

4.5.1 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.
```

4.5.2 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 typedefined 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
[[rpi reference]] or [[rpi reference]] 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 [[rpi reference]] nor
[[rpi reference]] 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 [[api reference]] interface, via the subroutines [[api reference]] and [[api reference]], to map variables from Grid data structures to the eosData array that [[api reference]] 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 [[api reference]], 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 [[api reference]] 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 [[api reference]] invocation when the optional gridDataStruct argument is absent or has a value of CENTER.

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

  For *eosmap-spec*, see above under VARIABLE. An *eosmap-spec* for FACEVAR is only used when [[api reference]] 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 **??** for more information.

- SCRATCHCENTERVAR *name [eosmap-spec]*
  This keyword is used in connection with the grid scope scratch space for cell-centered data supported by Flash-X. 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 Flash-X2. See **??** 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 [[api reference]] 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 **??**

  For *eosmap-spec*, see above under VARIABLE. An *eosmap-spec* for a MASS_SCALAR may be used in an [[api reference]] 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 **??**), *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 **??**, and in general **??**). 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 **??**) 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 **??** 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 **??**.

- `SPECIES` *name* [TO *number of ions*]
  An application that uses multiple species uses this keyword to define them. See **??** 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-X architecture.

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

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

- `LIBRARY` *name*
  Specifies a library requirement. Different Flash-X 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

- `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-X. 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-X 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-X_GRID_UG
  ug specific code
#endif

#ifdef Flash-X_GRID_PARAMESH3OR4
  pm3+ specific code
#endif
```

  By convention, many preprocessor symbols defined in Config files included in the Flash-X code distribution start with the prefix "Flash-X_".

- 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-X 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 ??.

4.6 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

Compilation warning

The Makefile.h *must* include a compiler flag to promote Fortran Reals to Double Precision. Flash-X 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 IO unit need to be compiled with the preprocessor symbol H5_USE_16_APIdefined. 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
```

4.7 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-X executable, but which contain useful information,

(b) Generated F90 or C code, and

(c) Makefiles required to compile the Flash-X executable.

4.7.1 Informational files

These files are generated before compilation by setup. Each of these files begins with the prefix setup_ for easy identification.

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

4.7.2 Code generated by the setup call

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

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

4.7.3 Makefiles generated by setup

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

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

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

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

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

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

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

4.8 Setup a hybrid MPI+OpenMP Flash-X application

There is the experimental inclusion of Flash-X multithreading with OpenMP in the Flash-X beta release. The units which have support for multithreading are split hydrodynamics **??**, unsplit hydrodynamics **??**, Gamma law and multigamma EOS **??**, Helmholtz EOS **??**, Multipole Poisson solver (improved version (support for 2D cylindrical and 3D cartesian)) **??** and energy deposition **??**.

The Flash-X 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-X 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-X 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-X 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-X 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-X 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-X application - you should build a MPI-2 library with thread support and then rebuild Flash-X.

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 4.7: Log file entries showing safe and unsafe threaded Flash-X applications

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

The Flash-X applications in Table **??** 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-X 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-X 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-X application with `+serialIO`.

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

Chapter 5

The Flash.h file

Flash.his a critical header file in Flash-X 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.

5.1 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 modeFIXEDBLOCKSIZE mode is when the sizes and the block bounds are determined at compile time. In NONFIXEDBLOCKSIZE modethe 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 ?? and ?? for more information.

5.2 Property Variables, Species and Mass Scalars

The unk data structure stores, in order, property variables (like density, pressure, temperature), the mass fraction of species, and mass scalars [1]. However, in Flash-X 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.

[1] See (??) for more information about mass scalars

5.3 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

5.4 Scratch Vars

In `Flash-X` 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.

5.5 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
```

5.6 Particles

5.6.1 Particles Types

Flash-X 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 desribing the type, initialization, and mapping methods must be provided for *each* type of particle included in the simulation.

5.6.2 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)
```

5.7 Non-Replicated Variable Arrays

For each non-replicated variable array defined in the `Config` file (see **??**), 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.

5.7.1 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.

5.7.2 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 elemenets 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.

5.7.3 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
```

5.8 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 **??** for concrete usage examples.

Part III

Driver Unit

Chapter 6

Driver Unit

The Driver unit controls the initialization and evolution of Flash-X 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.

6.1 Driver Routines

The most important routines in the Driver API are those that initialize, evolve, and finalize the Flash-X program. The file Flash.F90 contains the main Flash-X program (equivalent to main() in C). The default top-level program of Flash-X, 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.

6.1.1 Driver_initFlash

The first of these routines is [[api reference]], which initializes the parallel environment for the simulation. New in Flash-X 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 Flash-X, 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 [[api reference]], 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 **??**. 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.

6.1.2 `Driver_evolveFlash`

The next routine is [[api reference]] which controls the timestepping of the simulation, as well as the normal termination of Flash-X 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-X includes versions of `Driver_evolveFlash` for directionally-split and unsplit staggered mesh operators.

6.1.2.1 Strang Split Evolution

The code in the `Driver/DriverMain/Split` unit implementation directory has been the default time update method earlier, and can still be be used for many setups that can be configured with Flash-X. 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.

6.1.2.2 Unsplit Evolution

The driver implementation in the `Driver/DriverMain/Unsplit` directory is the default since `Flash-X`. It is required specifically for the two unsplit solvers: unsplit staggered mesh MHD solver (**??**) and the unsplit gas hydrodynamics solver (**??**). 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.

6.1.2.3 Super-Time-Stepping (STS)

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

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

$$\tau_i = \Delta t_{\text{CFL\_para}} \big[(-1 + \nu_{\text{sts}}) cos\big(\frac{\pi(2j-1)}{2N_{\text{sts}}} + 1 + \nu_{\text{sts}}\big)\big]^{-1}, \tag{6.1}$$

Table 6.1: Runtime parameters for STS

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

where $\Delta t_{\rm CFL\_para}$ is an explicit time step for a given parabolic system based on the CFL stability condition. ν (`nuSTS`) is a free parameter less than unity. For $\nu \to 0$, STS is asymptotically $N_{\rm sts}$ times faster than the conventional explicit scheme based on the CFL condition. During the N_{sts} sub-time steps, the STS method still solves solutions at each intermediate step τ_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-X invokes a single Δt_{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, τ_i is given by

$$\tau_i = \Delta t_{\rm CFL}\big[(-1+\nu_{\rm sts})cos\big(\frac{\pi(2j-1)}{2N_{\rm sts}}+1+\nu_{\rm sts}\big)\big]^{-1}, \tag{6.2}$$

where $\Delta t_{\rm CFL}$ can be an explicit time step for advection ($\Delta t_{\rm CFL\_adv}$) or parabolic ($\Delta t_{\rm CFL\_para}$) systems. In cases of advection-diffusion system, $\Delta t_{\rm CFL}$ takes ($\Delta t_{\rm CFL\_para}$) when it is smaller than ($\Delta t_{\rm CFL\_adv}$); otherwise, Flash-X'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 **??**.

6.1.2.4 Runtime Parameters

The `Driver` unit supplies certain runtime parameters regardless of which type of driver is chosen. These are described in the online [[rpi reference]]Driver/Runtime Parameters Documentation page.

Flash-X 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 `Flash-X`, the `Driver` declared a runtime parameter `iburn` to turn on and off burning.

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

In `Flash-X` 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.

6.1.3 `Driver_finalizeFlash`

Finally, the the `Driver` unit calls [[api reference]] which calls the finalize routines for each unit. Typically this involves deallocating memory and any other necessary cleanup.

6.1.4 Driver accessor functions

In `Flash-X` the `Driver` unit also provides a number of accessor functions to get data stored in the `Driver` unit, for example [[api reference]], [[api reference]], [[api reference]], [[api reference]].

Flash-X Transition

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

The `Driver` unit API also defines two interfaces for halting the code, [[api reference]] and [[C api reference]]`.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`.

6.1.5 Time Step Limiting

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

The implementation of [[api reference]] can lower or increase the time step. It takes various runtime parameters into consideration, including [[rpi reference]], [[rpi reference]], and [[rpi reference]]. For the most part, however, [[api reference]] 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.

Table 6.2: Runtime parameters for positive definiteness.

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

The code units that participate in this negotiation return a Δt, and usually some additional information about which location in the simulaton domain caused the reequired 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 [[api reference]] 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 [[rpi reference]].

6.1.5.1 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 `Flash-X`. 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 **??** 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 [[rpi reference]].

Part IV

Infrastructure Units

Chapter 7

Grid Unit

7.1 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 blocksand using these blocks as the primary computational entity visible to the physics units. A blockcontains a number of computational cells (`nxb` in the x-direction, `nyb` in the y-direction, and `nzb` in the z-direction). A perimeter of guardcellsof 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 **??**. Since the majority of physics solvers used in Flash-X 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-X. 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-X to abstract the grid from physics solvers and facilitates the ability of Flash-X 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-X. 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-X can interchangeably use either a **uniform** or **adaptive grid** for most problems. Additionally, a new feature in `Flash-X` is an option to replicate the mesh; that is processors are assumed to be partitioned into groups, each group gets a copy of the entire domain mesh. This feature is useful when it is possible to decompose the computation based upon certain compute intensive tasks that apply across the domain. One such example is radiation transfer with multigroup flux limited diffusion where each group needs an implicit solve. Here the state variable of the mesh are replicated on each group of processors, while the groups are unique. Thus at the cost of some memory redundancy, it becomes possible to compute a higher fidelity problem (see **??** 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 [[rpi reference]]

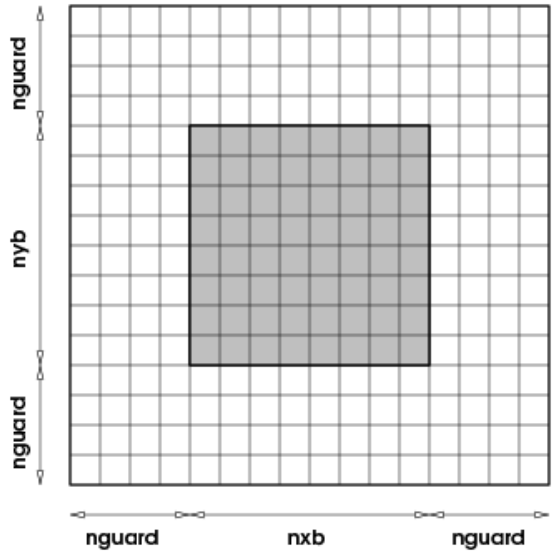

Figure 7.1: A single 2-D block showing the interior cells (shaded) and the perimeter of guard cells.

The uniform grid supported in Flash-X 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-X for providing adaptive grid capbility. The block-structured oct-tree based AMR package, `PARAMESH` has been the work horse since the beginning of the code.

Flash-X 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-X, the user must match up the correct `IO` alternative implementation with the correct `Grid` alternative implementation. Please see **??** 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 **??** for more information. Also see for shortcuts useful for the Grid unit.

7.2 GridMain Data Structures

The Grid unit is the most extensive infrastructure unit in the Flash-X 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-X 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.

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 **??**.

For cell-centered variables, the Grid unit also stores a **variable type** that can be retrieved using the [[api reference]] routine; see **??** 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 Flash-X3 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 **??** 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 **??** and **??**, which describe individual implementations of the Grid unit.

7.3 Computational Domain

The size of the computational domainin physical units is specified at runtime through the ([[rpi reference]], [[rpi reference]])([[rpi reference]], [[rpi reference]])([[rpi reference]], [[rpi reference]])runtime parameters. When working with curvilinear coordinates (see below in **??**), the extrema for angle coordinatesare specified in degrees. Internally all angles are represented in radians, so angles are converted to radians at Grid initialization.

Flash-X Transition

The convention for specifying the ranges for angular coordinates has changed from Flash-X2, which used units of π instead of degrees for angular coordinates.

The physical domain is mapped into a computational domain at problem initialization through routine [[api reference]] in PARAMESH and [[api reference]] in UG.When using the uniform grid UG, the mapping is easy: one block is created for each processor in the run, which can be sized either at build time or runtime depending upon the mode of UG use. [1] Further description can be found in ??. 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 [[rpi reference]], [[rpi reference]], and [[rpi reference]] are runtime parameters which default to 1.[2] The resolution of the computational domain is usually very coarse and unsuitable for computation after the initial mapping. The gr_expandDomain routine remedies the situation by applying the refinement process to the initial domain until a satisfactory level of resolution is reached everywhere in the domain. This method of mapping the physical domain to computational domain is effective because the resultant resolution in any section is related to the demands of the initial conditions there.

Flash-X Transition

Flash-X 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. Flash-X 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.

7.4 Boundary Conditions

Much of the Flash-X 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 bounday condition names as strings (in runtime parameters) and constants (in the source code); these are defined in [[api reference]] 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 [[rpi reference]] × [[rpi reference]] × [[rpi reference]] 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.

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

[2] The gr_createDomain function also can remove certain blocks of this initial mesh, if this is requested by a non-default [[api reference]] implementation.

- 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 [[rpi reference]] *etc.* need to be set to `"periodic"` in that case.)

7.4.1 Boundary Condition Types

Boundary conditions are determined by the physical problem. Within Flash-X, 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 [[rpi reference]] (for the 'left' X–boundary) to one of the supported boundary types (**??**) 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 [[api reference]].

*ab*_**boundary_type**	**Description**
`periodic`	Periodic ('wrap-around')
`reflect`,`reflecting`	Non-penetrating boundaries; plane symmetry, the normal vector components change sign
`outflow`	Zero-gradient boundary conditions; allows shocks to leave the domain
`diode`	like outflow, but fluid velocities are never allowed to let matter flow into the domain: normal velocity components are forced to zero in guard cells if necessary
`axisymmetric`	like `reflect`, but both normal and toroidal vector components change sign. Typically used with cylindrical geometry (R-Z) for the Z symmetry axis.
`eqtsymmetric`	like reflect for velocities but the magnetic field components, poloidal and toroidal, change sign. The sign of the normal magnetic field component remains the same. Typically used with cylindrical geometry (R-Z) for the R axis to emulate equatorial symmetry.
`hydrostatic-f2`	Hydrostatic boundary handling as in `Flash-X`. See remark in text.
`hydrostatic-f2+nvrefl`, `hydrostatic-f2+nvout`, `hydrostatic-f2+nvdiode`	Variants of `hydrostatic-f2`, where the **n**ormal **v**elocity is handled specially in various ways, analogous to `reflect`, `outflow`, and `diode` boundary conditions, respectively. See remark in text.
`user-defined` or `user`	The user must implement the desired boundary behavior; see text.

Table 7.1: Hydrodynamical boundary conditions supported by Flash-X. 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 [[rpi reference]] 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 [[api reference]]. 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).

*ab*_**boundary_type**	**Constant**	**Remark**
`isolated`	—	used by Gravity only for [[rpi reference]]
—	`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 `Flash-X`
`hydrostatic+nvrefl`	`HYDROSTATIC_NVREFL`	Hydrostatic variant, other impl. than `Flash-X`
`hydrostatic+nvout`	`HYDROSTATIC_NVOUT`	Hydrostatic variant, other impl. than `Flash-X`
`hydrostatic+nvdiode`	`HYDROSTATIC_NVDIODE`	Hydrostatic variant, other impl. than `Flash-X`

Table 7.2: Additional boundary condition types recognized by Flash-X. 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-X versions for a specific purpose (as indicate by the remarks), or are for special uses within the `Grid` unit.

7.4.2 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 [[api reference]]. Effectively this creates additional domain boundaries at the interface between blocks removed and regions still included. All boundary conditions other than `periodic` are possible at these additional boundaries, and are handled there in the same way as on external domain boundaries. This feature is only available with `PARAMESH`. See the documentation and example in [[api reference]] for more details and some caveats.

7.4.3 Implementing Boundary Conditions

Users may need to implement boundary conditions beyond those provided with `Flash-X`, 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 [[api reference]]`.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 [[api reference]] which by default is only provided as a stub and is explicitly intended to be implemented by users.

A [[api reference]] implementation gets called before [[api reference]] and can decide to either handle a specific combination of boundary condition type, direction, grid data structure, *etc.*, or leave it to [[api reference]]. These calls operate on a region of one block's cells at a time. Flash-X will pass additional information beyond that needed for handling simple boundary conditions to [[api reference]], 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-X 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: [[api reference]] is given only one fluid variable at a time, but can also handle data structures other than `unk`; whereas [[api reference]] handles all variables of `unk` along a row in one call. Cell coordinate information is included in the call arguments. Flash-X invokes these functions through an implementation of [[api reference]] in `Grid/GridBoundaryConditions/OneRow` which acts as a wrapper. `GridBoundaryConditions/OneRow` also provides a default implementation of [[api reference]] (which implements the simple boundary conditions) and [[api reference]] (which calls `Grid_applyBCEdge`) each. Another implementation of [[api reference]] can be found in `GridBoundaryConditions/OneRow/Flash2HSE`, this one calls `Grid_applyBCEdge` or, for `Flash-X`-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 [[api reference]], [[api reference]], [[api reference]], or
[[api reference]], the implementation does not fill guard cells in permanent data storage (the `unk` array and similar data structures) directly, but operates on buffers. `Flash-X` 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`.

7.4.3.1 Additional Concerns with `PARAMESH` 4

Boundary condition handling has become significantly more complex in `Flash-X`. 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-X): These cells can be filled for all blocks at a time that is predictable to the user code, as a standard part of handling [[api reference]], 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 Flash-X 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 hande" 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.

7.5 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.

7.5.1 FIXEDBLOCKSIZE Mode

FIXEDBLOCKSIZEmode, 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 $\texttt{NXB} * iprocs \times \texttt{NYB} * jprocs \times \texttt{NZB} * kprocs$. 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 interpolationwhile 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-X, this conservation step should be explicitly turned off with a runtime parameter [[rpi reference]] 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.

7.5.2 NONFIXEDBLOCKSIZE mode

Up ot version 2, Flash-X 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-X was limited to use the fixed block size mode described above. With `Flash-X` 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-X. 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. To run an application in "NONFIXEDBLOCKSIZE" modethe "`-nofbs`" option must be used when invoking the setup tool; see **??** 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 [[rpi reference]], [[rpi reference]] and [[rpi reference]]. 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` modeallocation 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 $\texttt{nvar} \times (2 * \texttt{nguard} + \texttt{nxb}) \times (2 * \texttt{nguard} + \texttt{nyb}) \times (2 * \texttt{nguard} + \texttt{nzb})$ 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 $16 \times 32 \times 64$.

7.6 Adaptive Mesh Refinement (AMR) Grid with Paramesh

The default package in Flash-X is PARAMESH (MacNeice *et al.* 1999) for implementing the adaptive mesh refinement (AMR) grid. PARAMESH uses a block-structured adaptive mesh refinement scheme similar to others in the literature (*e.g.*, Parashar 1999; Berger & Oliger 1984; Berger & Colella 1989; DeZeeuw & Powell 1993). It also shares ideas with schemes which refine on an individual cell basis (Khokhlov 1997). In block-structured AMR, the fundamental data structure is a block of cells arranged in a logically Cartesian fashion. "Logically Cartesian" implies that each cell can be specified using a block identifier (processor number and local block number) and a coordinate triple (i, j, k), where $i = 1 \ldots$ nxb, $j = 1 \ldots$ nyb, and $k = 1 \ldots$ nzb refer to the x-, y-, and z-directions, respectively. It does not require a physically rectangular coordinate system; for example a spherical grid can be indexed in this same manner.

The complete computational grid consists of a collection of blocks with different physical cell sizes, which are related to each other in a hierarchical fashion using a tree data structure. The blocks at the root of the tree have the largest cells, while their children have smaller cells and are said to be refined. Three rules govern the establishment of refined child blocks in PARAMESH. First, a refined child block must be one-half as large as its parent block in each spatial dimension. Second, a block's children must be nested; *i.e.*, the child blocks must fit within their parent block and cannot overlap one another, and the complete set of children 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 **??**, illustrating the rules above. Each block contains nxb $\times$ nyb $\times$ nzb interior cells and a set of guard cells. The guard cells contain boundary information needed to update the interior cells. These can be obtained from physically neighboring blocks, externally specified boundary conditions, or both. The number of guard cells needed depends upon the interpolation schemes and the differencing stencils used by the various physics units (usually hydrodynamics). For the explicit PPM algorithm distributed with Flash-X, four guard cells are needed in each direction, as illustrated in **??**. The blocksize while using the adaptive grid is fixed at compile time, resulting in static memoryallocation. The total number of blocks a processor can manage is determined by MAXBLOCKSwhich can be overridden at setup time with the `setup ...-maxblocks=#` argument. The amount of memory consumed by the Grid data structure of code when running with PARAMESH is NUNK_VARS $\times$ (2 $*$ nguard + nxb) $\times$ (2 $*$ nguard + nyb) $\times$ (2 $*$ nguard + nzb) $\times$ MAXBLOCKS. PARAMESH also needs memory to store its tree data structure for adaptive mesh management, over and above what is already mentioned with Uniform Grid. As the levels of refinement increase, the size of the tree also grows.

PARAMESH handles the filling of guard cells with information from other blocks or, at the boundaries of the physical domain, from an external boundary routine (see **??**). If a block's neighbor exists and has the same level of refinement, PARAMESH fills the corresponding guard cells using a direct copy from the neighbor's interior cells. If the neighbor has a different level of refinement, the data from the neighbor's cells must be adjusted by either interpolation(to a finer level of resolution) or averaging (to a coarser level)—see **??** 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 [[api reference]].

Grid Interpolationis also used when filling the blocks of children newly created in the course of automatic refinement. This happens during [[api reference]] processing. Averagingis 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 [[api reference]] processing.

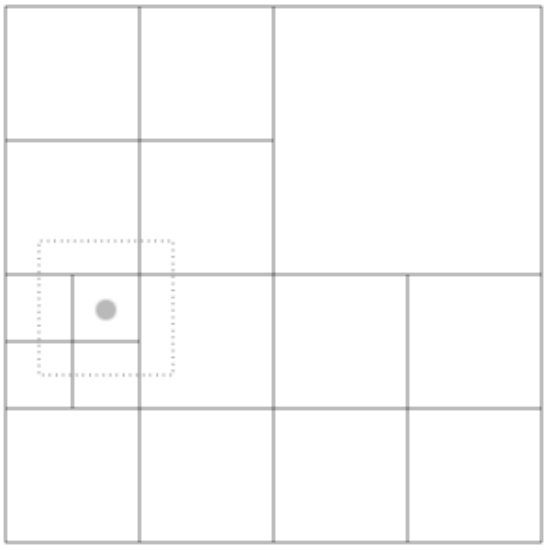

Figure 7.2: A simple computational domain showing varying levels of refinement in a total of 16 blocks. The dotted lines outline the guard cells for the block marked with a circle.

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 **??**). The summing allows a geometrical weighting to be implemented for non-Cartesian geometries, which ensures that the proper volume-corrected flux is computed.

7.6.1 Additional Data Structures

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

Flash-X 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 Flash-X; 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 $1 \leq n \leq lrefine\_max$.

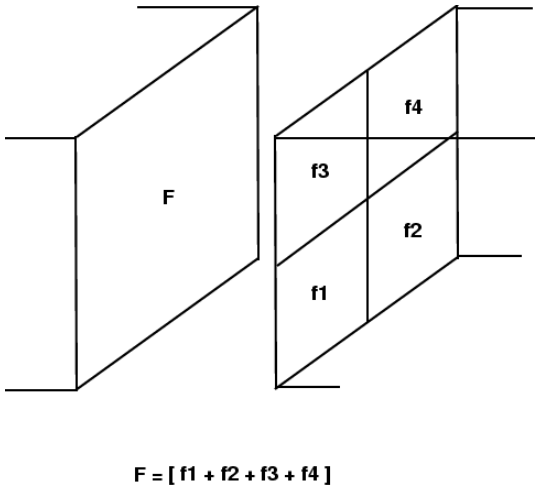

Figure 7.3: Flux conservation at a jump in refinement. The fluxes in the fine cells are added and replace the coarse cell flux (F).

The routine [[api reference]] 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 [[api reference]], [[api reference]], [[api reference]] and [[api reference]], 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.

7.6.2 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. [3] 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. Interpolationis 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 **??**.

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

1. monotonic
2. native

The choice is made at `setup` time, see **??**.

The versions of `PARAMESH` supplied with `Flash-X` supply their own default interpolation scheme, which is used when `Flash-X` is configured with the `-gridinterpolation=native setup` option (see **??**). The default schemes are only appropriate for Cartesian coordinates. If `Flash-X` 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

[3] 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.

data. The order of "monotonic" interpolation can be chosen with the [[rpi reference]]runtime parameter. See **??** for some more details on prolongation for curvilinear coordinates. At setup time, monotonic interpolation is the default interpolation used.

7.6.2.1 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 **??** 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.

`Flash-X` provides three ways to deal with this:

1. Do nothing—*i.e.*, assume that ignoring the difference between mass-specific and conserved form is a reasonable approximation. Depending on the smoothness of solutions in regions where refinement, derefinement, and jumps in refinement level occur, this assumption may be acceptable. This behavior can be forced by setting the [[rpi reference]] 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 [[rpi reference]] runtime parameter and corresponds roughly to `Flash-X` 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 `Flash-X`, this is the default behavior whenever available. It can be enabled explicitly (only necessary in setups that change the default) by setting the [[rpi reference]] runtime parameter.

7.6.3 Refinement

7.6.3.1 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-X Transition

`Flash-X` 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 [[rpi reference]] (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

$$E_i = \frac{\mid u_{i+1} - 2u_i + u_{i-1} \mid}{\mid u_{i+1} - u_i \mid + \mid u_i - u_{i-1} \mid + \epsilon[\mid u_{i+1} \mid + 2 \mid u_i \mid + \mid u_{i-1} \mid]} \ , \tag{7.1}$$

where u_i is the refinement test variable's value in the ith cell. The last term in the denominator of this expression acts as a filter, preventing refinement of small ripples, where ϵ 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

$$E_{i_1 i_2 i_3} = \left\{ \frac{\displaystyle\sum_{pq} \left(\frac{\partial^2 u}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right)^2}{\displaystyle\sum_{pq} \left[\left(\left| \frac{\partial u}{\partial x_p} \right|_{i_p+1/2} + \left| \frac{\partial u}{\partial x_p} \right|_{i_p-1/2} \right) \Delta x_p + \epsilon \frac{\partial^2 |u|}{\partial x_p \partial x_q} \Delta x_p \Delta x_q \right]^2} \right\}^{1/2} \ , \tag{7.2}$$

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 `Flash-X`'s default refinement criterion is a modification of the above, as follows:

$$E_i = \frac{\mid u_{i+2} - 2u_i + u_{i-2} \mid}{\mid u_{i+2} - u_i \mid + \mid u_i - u_{i-2} \mid + \epsilon[\mid u_{i+2} \mid + 2 \mid u_i \mid + \mid u_{i-2} \mid]} \ , \tag{7.3}$$

where again u_i is the refinement test variable's value in the ith cell. The last term in the denominator of this expression acts as a filter, preventing refinement of small ripples, where ϵ 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

$$E_{i_X i_Y i_Z} = \left\{ \frac{\displaystyle\sum_{pq} \left(\frac{\partial^2 u}{\partial x_p \partial x_q} \right)^2}{\displaystyle\sum_{pq} \left[\frac{1}{2\,\Delta x_q} \left(\left| \frac{\partial u}{\partial x_p} \right|_{i_q+1} + \left| \frac{\partial u}{\partial x_p} \right|_{i_q-1} \right) + \epsilon \frac{\bar{|u_{pq}|}}{\Delta x_p \Delta x_q} \right]^2} \right\}^{1/2} \ , \tag{7.4}$$

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

The constant ϵ is by default given a value of 10^{-2}, and can be overridden through the [[rpi reference]] 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 [[rpi reference]], where the number `#` matching the number of the [[rpi reference]] 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 [[rpi reference]].

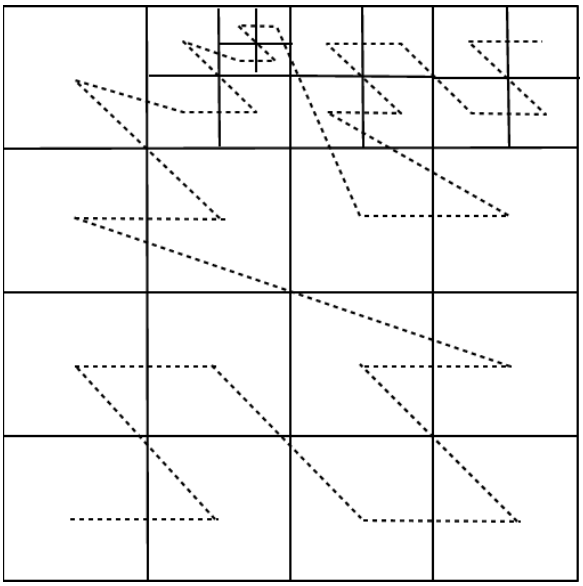

Figure 7.4: Morton space-filling curve for adaptive mesh grids.

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 [[rpi reference]] to true and setting [[rpi reference]] to the desired count.

7.6.3.2 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 **??**.

During the distribution step, each block is assigned a weight which estimates the relative amount of time required to update the block. The Morton number of the block is then computed by interleaving the bits of its integer coordinates, as described by Warren and Salmon (1987). This reordering determines its location along the space-filling curve. Finally, the list of all blocks is partitioned among the processors using the block weights, equalizing the estimated workload of each processor. By default, all leaf-blocks are weighted twice as heavily as all other blocks in the simulation.

7.6.3.3 Specialized Refinement Routines

Sometimes, it may be desirable to refine a particular region of the grid independent of the second derivative of the variables. This criterion might be, for example, to better resolve the flow at the boundaries of the domain, to refine a region where there is vigorous nuclear burning, or to better resolve some smooth initial condition. For curvilinear coordinates, regions around the coordinate origin or the polar z-axis may require special consideration for refinement. A collection of methods that can refine a (logically) rectangular region or a circular region in Cartesian coordinates, or can automatically refine by using some variable threshold, are available through the [[api reference]]. It is intended to be called from the [[api reference]] routine. The interface works by allowing the calling routine to pick one of the routines in the suite through an integer

argument. The calling routine is also expected to populate the data structure `specs` before making the call. A copy of the file `Grid_markRefineDerefine.F90` should be placed in the `Simulation` directory, and the interface file `Grid_interface.F90` should be used in the header of the function.

7.7 GridMain Usage

The `Grid` unit has the largest API of all units, since it is the custodian of the bulk of the simulation data, and is responsible for most of the code housekeeping. The [[api reference]] 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 [[api reference]] also creates the computational domain and the housekeeping data structures and initializes them. If using AMR, the computational domain is created by the [[api reference]] 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 [[api reference]] 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 [[api reference]] 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 [[api reference]], 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 ([[api reference]]), a partial or full plane from a block ([[api reference]]), a partial or full row ([[api reference]]), or a single point ([[api reference]]). 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 [[api reference]] returns a list of blocks that meet the specified criterion such as being "LEAF" blocks in `PARAMESH`, or residing on the physical boundary.

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 [[api reference]]. 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 [[api reference]] 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-X 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 `Flash-X`, [[api reference]] 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 [[api reference]] for documentation. When these optional arguments are absent or when a `Grid` implementation does not support them, Flash-X falls back to safe default behavior which may, however, be needlessly expensive.

Another routine that may change the global state of the grid is [[api reference]]. 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-X Transition

As mentioned in **??**, Flash-X allows every unit to identify scalar variables for checkpointing. In the `Grid` unit, the function that takes care of consolidating user specified checkpoint variable is [[api reference]]. Users can select their own variables to checkpoint by including an implementation of this function specific to their requirements in their Simulation setup directory.

7.8 GridParticles

Flash-X is primarily an Eulerian code, however, there is support for tracing the flow using Lagrangian particles. In `Flash-X` 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-X also uses active particles with mass in cosmological simulations, and charged particles in a hybrid PIC solver. Each particle has an associated data structure, which contains fields such as its physical position and velocity, and relevant physical attributes such as mass or field values in active particles. Depending upon the time advance method, there may be other fields to store intermediate values. Also, depending upon the requirements of the simulation, other physical variables such as temperature *etc.* may be added to the data structure. The `GridParticles` subunit of the `Grid` unit has two sub-subunits of its own. The `GridParticlesMove` sub-subunit moves the data structures associated with individual particles when the particles move between blocks; the actual movement of the particles through time advancement is the responsibility of the `Particles` unit. Particles move from one block to another when their time advance places them outside their current block. In AMR, the particles can also change their block through the process of refinement and derefinement. The `GridParticlesMap` sub-subunit provides mapping between particles data and the mesh variables. The mesh variables are either cell-centered or face-centered, whereas a particle's position could be anywhere in the cell. The `GridParticlesMap` sub-subunit calculates the particle's properties at its position from the corresponding mesh variable values in the appropriate cell . When using active particles, this sub-subunit also maps the mass of the particles onto the specified mesh variable in appropriate cells. The next sections describe the algorithms for moving and mapping particles data.

7.8.1 GridParticlesMove

`Flash-X` has implementations of three different parallel algorithms for moving the particles data when they are displaced from their current block. `Flash-X` 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 `Flash-X`. The simplest algorithm, `Directional algorithm` is applicable only to the uniform grid when it is configured with one block per processor. This algorithm uses directional movement of data, and is easy because the directional neighbors are trivially known. The movement of particles data is much more challenging with AMR even when the grid is not refining. Since the blocks are at various levels of refinement at any given moment, a block may have more than one neighbor along one or more of its faces. The distribution of blocks based on space-filling curve is an added complication since the neighboring blocks along a face may reside at a non-neighboring processor The remaining two algorithmss included in `Flash-X` 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

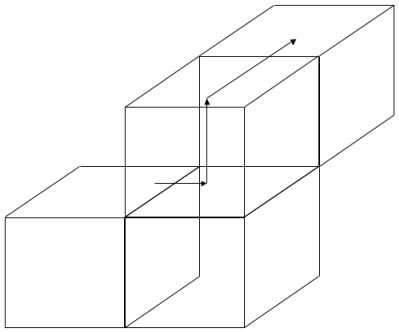

Figure 7.5: Moving one particle to a neighbor on the corner.

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.

7.8.1.1 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 neightbor'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 direaction.

At the end of these steps, all particles will have reached their destination blocks, including those that move to a neighbor on the corner. **??** illustrates the steps in getting a particle to its correct destination.

7.8.1.2 Point To Point Algorithm

As a part of the data cached by Paramesh, there is wealth of information about the neighborhood of all the blocks on a processor. This information includes the processor and block number of all neighbors (face and corners) if they are at the same refinement level. If those neighbors are at lower refinement level, then the neighbor block is in reality the current block's parent's neighbor, and the parent's neighborhood information is part of the cached data. Similarly, if the neighbor is at a higher resolution then the current blocks neighbor is in reality the parent of the neighbor. The parent's metadata is also cached, from which information about all of its children can be derived. Thus it is possible to determine the processor and block number of the destination block for each particle. The PointToPoint implementation finds out the destinations for every particles that is getting displaced from its block. Particles going to local destination blocks are moved first. The remaining particles are sorted based on their destination processor number, followed by a couple of global operations that allow every processor to determine the number of particles it is expected to receive

from all of the other processors. A processor then posts asynchronous receives for every source processor that had at least one particle to send to it. In the next step, the processor cycles through the sorted list of particles and sends them to the appropriate destinations using synchronous mode of communition.

7.8.1.3 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.

7.8.2 GridParticlesMapToMesh

`Flash-X` 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 **??**. 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 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 **??**.

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 **??** and **??**, respectively.

Flash-X 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
```

7.8.2.1 Uniform Grid

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

7.8.2.2 Paramesh Grid

There are two implementations of the AMR algorithms for accumulating particles' attribute on the grid. They are inspired by a particle redistribution algorithms `Sieve` and `Point to Point` described in **??** and **??** 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 **??**. 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 **??** or **??**

The guard cell region decomposition described in Step 2 is illustrated in **??**. Here, the clear regions correspond to guard cells and the gray region corresponds to internal cells. Each guard cell region contains cells which correspond to the internal cells of a single neighboring block at the same refinement.

Figure 7.6: 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 Flash-X 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 **??**). 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 Flash-X.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 Flash-X2, 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 Flash-X.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 Flash-X.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.

7.9 GridSolvers

The `GridSolvers` unit groups together subunits that are used to solve particular types of differential equations. Currently, there are two types of solvers: a parallel Fast Fourier Transform package (**??**) and various solvers for the Poisson equation (**??**).

7.9.1 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 **??** 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-X 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-X uses `fftpack` from NCAR for the 1-D FFTs, since that package also contains transforms that are useful with non-periodic boundary conditions.

The lowest layer has three distinct components. The first component redistributes data. It includes routines for distributed transposes for two and three dimensional data. The second component provides a uniform interface for FFT calls to hide the details of individual libraries. The third component is the data structures. There are global data structures to keep track of the type of transform, number of data dimensions, and physical and transform space information about each dimension. The dimensional information includes the start and end point of data (useful when the dimension is spread over more than one processor), the MPI communicator, the coordinates of the node in the processor grid etc. The structures also include pointers to the trigonometric tables and work space needed by the sequential FFT calls.

The upper layer of PFFT combines the lower layer routines to create end-to-end transforms in a variety of ways. The available one dimensional transforms include real-to-complex, complex-to-complex, sine and cosine transforms. They can be combined to create two or three dimensional tranforms for different configuration of the domain. The two dimensional transforms support parallelization of one dimension (or a one dimensional grid of processors). The three dimensional transforms support one or two dimensional grid of processors. All transforms must have at least one dimension within the processor at all times. The data distribution changes during the computation. However, a complete cycle of forward and inverse transform restores the data distribution.

The computation of a forward three dimensional FFT in parallel involves following steps :

1. Compute 1-D transforms along **x**.
2. Reorder, or transpose from **x-y-z** to **y-z-x**
3. Compute 1-D transforms along **y**. If the transform along **x** was real-to-complex, this one must be a complex-to-complex transform.
4. Transpose from **y-z-x** to **z-x-y**
5. Compute 1-D FFTs along **z**. If the transform along **x** or **y** was real-to-complex, this must be a complex-to-complex transform.

The inverse transform can be computed by carrying out the steps described above in reverse order. The more commonly used domain decomposition in FFT based codes assumes a one dimensional processor grid:

$$N_1 \times N_2 \times N_3/P, \tag{7.5}$$

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

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

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

$$N_1 \times N_2/P_1 \times N_3/P_2 \qquad (7.6)$$
$$P = P_1 \times P_2.$$

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

7.9.1.1 Using `Pfft`

`Pfft` can only be used with a pencil grid, with the constraint that the number of processors along the `IAXIS` must be 1. This is because all one dimensional transforms are computed locally within a processor. However, Flash-X 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 **??**.

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 [[rpi reference]] to true, which enables the `Flash-X` 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-X` 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 [[api reference]] for computing the transforms must be preceded by a call to [[api reference]] and followed by a call to [[api reference]]. In addition, the runtime parameter [[rpi reference]] 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.

7.9.1.2 `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-X` 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-X 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-X 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-X block fragments and one for Pfft block fragments. Metadata information about each Flash-X block fragment is placed in separate messages and sent using MPI_Isend to the appropriate destination pencil grid processor.

Each destination pencil grid processor repeatedly invokes MPI_Iprobe using MPI_ANY_SOURCE, and creates a node in its Pfft list whenever it discovers a message. The MPI message is received into a metadata region of the freshly allocated node, and a communication buffer is also allocated according to the size specified in the metadata. The pencil processor continues probing for messages until the cumulative size of its node's communication buffers is equal to the pencil grid points it has been assigned. At this stage, grid data is communicated by traversing the Pfft list and posting MPI_Irecvs, and then traversing the Flash-X 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.

7.9.1.3 Unit Test

The unit test for Pfft solver solves the following equation:

$$\nabla^2(\mathbf{F}) = -13.0 * \cos 2x * \sin 3y \tag{7.7}$$

The simplest analytical solution of this equation assuming no constants is

$$F = \cos 2x * \sin 3y \tag{7.8}$$

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 (**??**). We compute the forward real-to-complex transform of the solution data variable that is initialized with the right hand side of (**??**). 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.

7.9.2 Poisson equation

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

$$\nabla^2\phi(\mathbf{x}) = \alpha\rho(\mathbf{x}) \ . \tag{7.9}$$

Here α 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.

7.9.2.1 Multipole Poisson solver (original version)

This section describes the multipole Poisson solver that has been included in all the past releases of Flash-X. It is included in the current release also, however, certain limitations found in this solver lead to a new implementation described in **??**. This version is retained in `Flash-X`, 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 $\mathbf{x}_{\rm cm}$

$$\mathbf{x}_{\rm cm} = \frac{\int d^3\mathbf{x}\,\mathbf{x}\rho(\mathbf{x})}{\int d^3\mathbf{x}\,\rho(\mathbf{x})} \ . \tag{7.10}$$

We will take $\mathbf{x}_{\rm cm}$ as our origin. In integral form, Poisson's ((**??**)) is

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi}\int d^3\mathbf{x}'\,\frac{\rho(\mathbf{x}')}{|\mathbf{x}-\mathbf{x}'|} \ . \tag{7.11}$$

The Green's function for this equation satisfies the relationship

$$\frac{1}{|\mathbf{x}-\mathbf{x}'|} = 4\pi\sum_{\ell=0}^{\infty}\sum_{m=-\ell}^{\ell}\frac{1}{2\ell+1}\frac{r_<^\ell}{r_>^{\ell+1}}Y^*_{\ell m}(\theta',\varphi')Y_{\ell m}(\theta,\varphi) \ , \tag{7.12}$$

where the components of $\mathbf{x}$ and $\mathbf{x}'$ are expressed in spherical coordinates (r, θ, φ) about $\mathbf{x}_{\rm cm}$, and

$$\begin{aligned} r_< &\equiv \min\{|\mathbf{x}|, |\mathbf{x}'|\} \\ r_> &\equiv \max\{|\mathbf{x}|, |\mathbf{x}'|\} \ . \end{aligned} \tag{7.13}$$

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

$$Y_{\ell m}(\theta,\varphi) \equiv (-1)^m\sqrt{\frac{2\ell+1}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}P_{\ell m}(\cos\theta)e^{im\varphi} \ . \tag{7.14}$$

$P_{\ell m}(x)$ are Legendre polynomials. Substituting (**??**) into (**??**), we obtain

$$\begin{aligned} \phi(\mathbf{x}) \quad = \quad & -\alpha\sum_{\ell=0}^{\infty}\sum_{m=-\ell}^{\ell}\frac{1}{2\ell+1}\Bigg\{Y_{\ell m}(\theta,\varphi)\times \\ & \left[r^\ell\int_{r<r'}d^3\mathbf{x}'\frac{\rho(\mathbf{x}')Y^*_{\ell m}(\theta',\varphi')}{r'^{\ell+1}} + \frac{1}{r^{\ell+1}}\int_{r>r'}d^3\mathbf{x}'\rho(\mathbf{x}')Y^*_{\ell m}(\theta',\varphi')r'^\ell\right]\Bigg\} \ . \end{aligned} \tag{7.15}$$

In practice, we carry out the first summation up to some limiting multipole $\ell_{\rm max}$. 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 (**??**) is real-valued

$$\sum_{m=-\ell}^{\ell} Y^*_{\ell m}(\theta',\varphi')Y_{\ell m}(\theta,\varphi) = \frac{2\ell+1}{4\pi}\left[P_{\ell 0}(\cos\theta)P_{\ell 0}(\cos\theta') + 2\sum_{m=1}^{\ell}\frac{(\ell-m)!}{(\ell+m)!}P_{\ell m}(\cos\theta)P_{\ell m}(\cos\theta')\cos\left(m(\varphi-\varphi')\right)\right] . \tag{7.16}$$

Using a trigonometric identity to split up the last cosine in this expression and substituting for the inner sums in (**??**), we obtain

$$\begin{aligned}\phi(\mathbf{x}) = & -\frac{\alpha}{4\pi}\sum_{\ell=0}^{\infty}P_{\ell 0}(\cos\theta)\left[r^\ell\mu^{\rm eo}_{\ell 0}(r)+\frac{1}{r^{\ell+1}}\mu^{\rm ei}_{\ell 0}(r)\right] - \\ & \frac{\alpha}{2\pi}\sum_{\ell=1}^{\infty}\sum_{m=1}^{\ell}P_{\ell m}(\cos\theta)\Big[(r^\ell\cos m\varphi)\mu^{\rm eo}_{\ell m}(r)+(r^\ell\sin m\varphi)\mu^{\rm oo}_{\ell m}(r)+ \\ & \frac{\cos m\varphi}{r^{\ell+1}}\mu^{\rm ei}_{\ell m}(r)+\frac{\sin m\varphi}{r^{\ell+1}}\mu^{\rm oi}_{\ell m}(r)\Big] .\end{aligned} \tag{7.17}$$

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

$$\mu^{\rm ei}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!}\int_{r>r'} d^3\mathbf{x}'\, r'^\ell\rho(\mathbf{x}')P_{\ell m}(\cos\theta')\cos m\varphi' \tag{7.18}$$

$$\mu^{\rm oi}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!}\int_{r>r'} d^3\mathbf{x}'\, r'^\ell\rho(\mathbf{x}')P_{\ell m}(\cos\theta')\sin m\varphi' \tag{7.19}$$

$$\mu^{\rm eo}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!}\int_{r<r'} d^3\mathbf{x}'\,\frac{\rho(\mathbf{x}')}{r'^{\ell+1}}P_{\ell m}(\cos\theta')\cos m\varphi' \tag{7.20}$$

$$\mu^{\rm oo}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!}\int_{r<r'} d^3\mathbf{x}'\,\frac{\rho(\mathbf{x}')}{r'^{\ell+1}}P_{\ell m}(\cos\theta')\sin m\varphi' . \tag{7.21}$$

The procedure is thus to compute the moment integrals ((**??**) − (**??**)) for a given source field $\rho(\mathbf{x})$, and then to use these moments in (**??**) 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 $\mathbf{x}_{\rm cm}$, with an implied discretization. The solver allocates storage for moment samples spaced a distance Δ apart in radius

$$\mu^{\rm ei}_{\ell m,q} \equiv \mu^{\rm ei}_{\ell m}(q\Delta) \qquad \mu^{\rm eo}_{\ell m,q} \equiv \mu^{\rm eo}_{\ell m}((q-1)\Delta) \tag{7.22}$$

$$\mu^{\rm oi}_{\ell m,q} \equiv \mu^{\rm oi}_{\ell m}(q\Delta) \qquad \mu^{\rm oo}_{\ell m,q} \equiv \mu^{\rm oo}_{\ell m}((q-1)\Delta) . \tag{7.23}$$

The sample index q varies from 0 to N_q ($\mu^{\rm eo}_{\ell m,0}$ and $\mu^{\rm oo}_{\ell m,0}$ are not used). The sample spacing Δ is chosen to be one-half the geometric mean of the x, y, and z cell spacings at the highest level of refinement, and N_q is chosen to be large enough to span the diagonal of the computational volume with samples.

Determining the contribution of individual cells to the tabulated moments requires some care. To reduce the error caused by the grid geometry, in each cell ijk an optional subgrid can be establish (see [[rpi reference]]) consisting of N' points at the locations $\mathbf{x}'_{i'j'k'}$, where

$$x'_{i'} = x_i + (i' - 0.5(N'-1))\frac{\Delta x_i}{N'} , \qquad i' = 0\ldots N'-1 \tag{7.24}$$

$$y'_{j'} = y_j + (j' - 0.5(N'-1))\frac{\Delta y_j}{N'} , \qquad j' = 0\ldots N'-1 \tag{7.25}$$

$$z'_{k'} = z_k + (k' - 0.5(N'-1))\frac{\Delta z_k}{N'} , \qquad k' = 0\ldots N'-1 , \tag{7.26}$$

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

$$\mu^{\rm ei}_{\ell m,q\geq q'} \leftarrow \mu^{\rm ei}_{\ell m,q\geq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} r'^{\ell}_{i'j'k'} \rho(\mathbf{x}'_{i'j'k'}) P_{\ell m}(\cos\theta'_{i'j'k'}) \cos m\varphi'_{i'j'k'} \tag{7.27}$$

$$\mu^{\rm oi}_{\ell m,q\geq q'} \leftarrow \mu^{\rm oi}_{\ell m,q\geq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} r'^{\ell}_{i'j'k'} \rho(\mathbf{x}'_{i'j'k'}) P_{\ell m}(\cos\theta'_{i'j'k'}) \sin m\varphi'_{i'j'k'} \tag{7.28}$$

$$\mu^{\rm eo}_{\ell m,q\leq q'} \leftarrow \mu^{\rm eo}_{\ell m,q\leq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{r'^{\ell+1}_{i'j'k'}} P_{\ell m}(\cos\theta'_{i'j'k'}) \cos m\varphi'_{i'j'k'} \tag{7.29}$$

$$\mu^{\rm oo}_{\ell m,q\leq q'} \leftarrow \mu^{\rm oo}_{\ell m,q\leq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{r'^{\ell+1}_{i'j'k'}} P_{\ell m}(\cos\theta'_{i'j'k'}) \sin m\varphi'_{i'j'k'} \ , \tag{7.30}$$

where

$$q' = \left\lfloor \frac{|\mathbf{x}'_{i'j'k'}|}{\Delta} \right\rfloor + 1 \tag{7.31}$$

is the index of the radial sample within which the subcell center lies. These expressions introduce (hopefully) small errors when compared to ((**??**)) − (**??**), 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 Δ^2 is also introduced by assuming the source profile within each cell to be flat. Note that the total source computed by this method ($\mu^{\rm ei}_{\ell m,N_q}$) is exactly equal to the total implied by ρ_{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 ((**??**)) − (**??**) 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

$$x'_{i'} = x_i + (i' - 0.5N'))\frac{\Delta x_i}{N'} \ , \qquad i' = 0 \ldots N' \tag{7.32}$$

$$y'_{j'} = y_j + (j' - 0.5N'))\frac{\Delta y_j}{N'} \ , \qquad j' = 0 \ldots N' \tag{7.33}$$

$$z'_{k'} = z_k + (k' - 0.5N'))\frac{\Delta z_k}{N'} \ , \qquad k' = 0 \ldots N' \ . \tag{7.34}$$

The cell-averaged potential in cell ijk is then

$$\phi_{ijk} = \frac{1}{N'^3} \sum_{i'j'k'} \phi(\mathbf{x}'_{i'j'k'}) \ , \tag{7.35}$$

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

7.9.2.2 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 implemention 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 $\mathbf{x}_{\rm cen}$ is determined via density-squared weighted integration over position:

$$\mathbf{x}_{\rm cen} = \frac{\int \mathbf{x}\rho^2(\mathbf{x})\, d\mathbf{x}}{\int \rho^2(\mathbf{x})\, d\mathbf{x}}. \tag{7.36}$$

We will take $\mathbf{x}_{\rm cen}$ as our origin. In integral form, Poisson's equation (**??**) becomes

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \int \frac{\rho(\mathbf{x}')}{|\mathbf{x} - \mathbf{x}'|}\, d\mathbf{x}'. \tag{7.37}$$

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

$$\frac{1}{|\mathbf{x} - \mathbf{x}'|} = \sum_{\ell=0}^{\infty} \frac{x_<^\ell}{x_>^{\ell+1}} P_\ell(\cos\gamma), \tag{7.38}$$

where $x_<$ ($x_>$) indicate the smaller (larger) of the magnitudes and γ 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

$$P_\ell(\cos\gamma) = \frac{4\pi}{2\ell + 1} \sum_{m=-\ell}^{+\ell} Y_{\ell m}^*(\theta', \phi') Y_{\ell m}(\theta, \phi), \tag{7.39}$$

where θ, ϕ and θ', ϕ' are the spherical angular components of $\mathbf{x}$ and $\mathbf{x}'$ about $\mathbf{x}_{\rm cen}$. Defining now the regular $R_{\ell m}$ and irregular $I_{\ell m}$ solid harmonic functions

$$R_{\ell m}(x_<) = \sqrt{\frac{4\pi}{2\ell + 1}} x_<^\ell Y_{\ell m}(\theta, \phi) \tag{7.40}$$

$$I_{\ell m}(x_>) = \sqrt{\frac{4\pi}{2\ell + 1}} \frac{Y_{\ell m}(\theta, \phi)}{x_>^{\ell+1}}, \tag{7.41}$$

we can rewrite Eq.(**??**) in the form

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi} \int \sum_{\ell m} R_{\ell m}(x_<) I_{\ell m}^*(x_>) \rho(\mathbf{x}')\, d\mathbf{x}', \tag{7.42}$$

where the summation sign is a shorthand notation for the double sum over all the allowed ℓ and m values. In Flash-X 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

$$\phi(q) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}(q_<) I^*_{\ell m}(q_>) m(q'), \tag{7.43}$$

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 $\phi_{Self}(q)$ 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.(**??**) by setting $\cos\gamma = 1$:

$$\phi_{Self}(q) \quad = \quad m(q)\frac{L+1}{x_q}, \tag{7.44}$$

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.(**??**) will thus be replaced by

$$\phi(q) = \frac{1}{n_F} \sum_F \phi(\mathbf{x}_F) \tag{7.45}$$

and

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \sum_{q'} \sum_{\ell m} R_{\ell m}([q', x_F]_<) I^*_{\ell m}([q', x_F]_>) m(q'), \tag{7.46}$$

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

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left\{ \sum_{q' \le x_F} \sum_{\ell m} \left[R_{\ell m}(q') m(q')\right] I^*_{\ell m}(\mathbf{x}_F) + \sum_{q' > x_F} \sum_{\ell m} R_{\ell m}(\mathbf{x}_F) \left[I^*_{\ell m}(q') m(q')\right] \right\}, \tag{7.47}$$

and defining the two moments

$$M^R_{\ell m}(\mathbf{x}_F) \quad = \quad \sum_{q' \le x_F} R_{\ell m}(q') m(q') \tag{7.48}$$

$$M^I_{\ell m}(\mathbf{x}_F) \quad = \quad \sum_{q' > x_F} I_{\ell m}(q') m(q'), \tag{7.49}$$

we obtain

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left[\sum_{\ell m} M^R_{\ell m}(\mathbf{x}_F) I^*_{\ell m}(\mathbf{x}_F) + \sum_{\ell m} M^{I*}_{\ell m}(\mathbf{x}_F) R_{\ell m}(\mathbf{x}_F) \right] \tag{7.50}$$

and using vector notation

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left[\mathbf{M}^R(\mathbf{x}_F) \cdot \mathbf{I}^*(\mathbf{x}_F) + \mathbf{M}^{I*}(\mathbf{x}_F) \cdot \mathbf{R}(\mathbf{x}_F) \right]. \tag{7.51}$$

We now change from complex to real formulation. We state this for the regular solid harmonic functions, the same reasoning being applied to the irregular solid harmonic functions and all their derived moments. The regular solid harmonic functions can be split into a real and imaginary part

$$R_{\ell m} = R^c_{\ell m} + i\, R^s_{\ell m}. \tag{7.52}$$

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

$$R^c_{\ell,-m} = (-1)^m R^c_{\ell m} \tag{7.53}$$
$$R^s_{\ell,-m} = -(-1)^m R^s_{\ell m} \tag{7.54}$$

we can restrict ourselves to the following polar angle number ranges

$$c \ : \ \ell \geq 0 \ , \ \ell \geq m \geq 0 \tag{7.55}$$
$$s \ : \ \ell \geq 1 \ , \ \ell \geq m \geq 1. \tag{7.56}$$

The real formulation of (**??**) becomes then

$$\phi(\mathbf{x}_F) = -\frac{\alpha}{4\pi} \left\{ \begin{bmatrix} \mathbf{M}^{Rc}(\mathbf{x}_F) \\ \mathbf{M}^{Rs}(\mathbf{x}_F) \end{bmatrix} \cdot \mathbf{\Delta} \begin{bmatrix} \mathbf{I}^{c}(\mathbf{x}_F) \\ \mathbf{I}^{s}(\mathbf{x}_F) \end{bmatrix} + \begin{bmatrix} \mathbf{M}^{Ic}(\mathbf{x}_F) \\ \mathbf{M}^{Is}(\mathbf{x}_F) \end{bmatrix} \cdot \mathbf{\Delta} \begin{bmatrix} \mathbf{R}^{c}(\mathbf{x}_F) \\ \mathbf{R}^{s}(\mathbf{x}_F) \end{bmatrix} \right\}, \tag{7.57}$$

which, when compared to (**??**), shows, that all vectors now contain a cosine and a sine section. The $\mathbf{\Delta}$ matrix is a diagonal matrix whose elements are equal to 2 for $m \neq 0$ and 1 otherwise, i.e.:

$$\mathbf{\Delta} = diag(2 - \delta_{m0}). \tag{7.58}$$

The recursion relations for calculating the solid harmonic vectors are

$$R^c_{00} = 1 \tag{7.59}$$
$$R^c_{\ell\ell} = -\frac{xR^c_{\ell-1,\ell-1} - yR^s_{\ell-1,\ell-1}}{2\ell} \tag{7.60}$$
$$R^s_{\ell\ell} = -\frac{yR^c_{\ell-1,\ell-1} + xR^s_{\ell-1,\ell-1}}{2\ell} \tag{7.61}$$
$$R^{c/s}_{\ell m} = \frac{(2\ell-1)zR^{c/s}_{\ell-1,m} - r^2 R^{c/s}_{\ell-2,m}}{(\ell+m)(\ell-m)}, \quad 0 \leq m < \ell \tag{7.62}$$

and

$$I^c_{00} = \frac{1}{r} \tag{7.63}$$
$$I^c_{\ell\ell} = -(2\ell-1)\frac{xI^c_{\ell-1,\ell-1} - yI^s_{\ell-1,\ell-1}}{r^2} \tag{7.64}$$
$$I^s_{\ell\ell} = -(2\ell-1)\frac{yI^c_{\ell-1,\ell-1} + xI^s_{\ell-1,\ell-1}}{r^2} \tag{7.65}$$
$$I^{c/s}_{\ell m} = \frac{(2\ell-1)zI^{c/s}_{\ell-1,m} - \left[(\ell-1)^2 - m^2\right] I^{c/s}_{\ell-2,m}}{r^2}, \quad 0 \leq m < \ell \tag{7.66}$$

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-X, the order of the two cosine and sine components for each solid harmonic vector is such that ℓ precedes m. This allows buildup of the vectors with maximum number of unit strides. The same applies of course for the assembly of the moments. For 2D cylindrical and 2D spherical geometries only the $m = 0$ parts of both recursions are needed, involving only the cartesian z coordinate and r^2. Symmetry along the radial axes of these 2D geometries inflicts only the sign change $z \to -z$, resulting in the symmetry relations $R^c_{\ell 0} \to R^c_{\ell 0}$ for even ℓ and $R^c_{\ell 0} \to -R^c_{\ell 0}$ for odd ℓ, 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 ℓ moments by a factor of 2 while setting the odd ℓ 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-X 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 (**??**-**??**), 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 ℓ values this leads to potential computational over- and underflow. To get independent of the size of both the coordinates and ℓ 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:

$$R_{Lm}^{c/s} \approx 1 \tag{7.67}$$

$$I_{Lm}^{c/s} \approx 1. \tag{7.68}$$

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.(**??**). 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:

$$|R_{\ell 0}| = \frac{r^\ell}{\ell!} \tag{7.69}$$

$$|I_{\ell 0}| = \frac{\ell!}{r^{\ell+1}}. \tag{7.70}$$

Since $r = \sqrt{x^2+y^2+z^2}$, damping of the coordinates with D results in a damped radial cell face distance Dr. Inserting this result into (**??**) and (**??**) and imposing conditions (**??**) and (**??**) results in

$$D_R = \frac{1}{r}\sqrt[L]{L!} \approx \frac{1}{r}\frac{L}{e}\sqrt[2L]{2\pi L} \tag{7.71}$$

$$D_I = \frac{1}{r}\sqrt[L+1]{L!} \approx \frac{1}{r}\frac{L}{e}\sqrt[2L+2]{\frac{2\pi e^2}{L}}, \tag{7.72}$$

where the approximate forms are obtained by using Stirling's factorial approximation formula for large L. In Flash-X 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 (**??**) and (**??**) we see, that the moments are sums over subsets of cell center solid harmonic vectors multiplied by the corresponding cell mass. From Eq.(**??**) 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 $q \to Q$ 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

$$\mathbf{M}^R(Q) = \sum_{q \leq Q} \mathbf{R}(q) m(q) \tag{7.73}$$

$$\mathbf{M}^I(Q) = \sum_{q \geq Q} \mathbf{I}(q) m(q), \tag{7.74}$$

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.(**??**) 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 $\mathbf{x} \to Q$. Consider the three consecutive radial bins $Q-1$, Q and $Q+1$, together with their calculated moments:

$$\begin{array}{c|c|c} \mathbf{M}^R(Q-1) & \mathbf{M}^R(Q) & \mathbf{M}^R(Q+1) \\ \mathbf{M}^I(Q-1) & \mathbf{M}^I(Q) & \mathbf{M}^I(Q+1) \end{array} \tag{7.75}$$

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:

$$\mathbf{M}^R(\mathbf{x}) = \mathbf{M}^R(Q-1) + R_{frac}\left[\mathbf{M}^R(Q) - \mathbf{M}^R(Q-1)\right] \tag{7.76}$$

$$\mathbf{M}^I(\mathbf{x}) = \mathbf{M}^I(Q+1) + I_{frac}\left[\mathbf{M}^I(Q) - \mathbf{M}^I(Q+1)\right], \tag{7.77}$$

The extraction of the moments via (**??**) and (**??**) 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-X multipole solver: i) exponentially and/or ii) logarthmically growing:

$$\text{exponential bin size upper radial limit} = s \cdot \Delta r \cdot Q^t \tag{7.78}$$

$$\text{logarithmic bin size upper radial limit} = s \cdot \Delta r \cdot \frac{e^{tQ}-1}{e^t-1}. \tag{7.79}$$

In these definitions, Δr 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 decrase the atomic unit radial measure and $Q = 1, 2, \ldots$ is a local bin index counter for each outer zone. The atomic radial distance Δr is calculated for each individual domain geometry as follows:

Domain Geometry	Δr
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	$\frac{1}{2}\Delta R$

, (7.80)

where $\Delta x, \Delta y, \Delta z$ are the usual cartesian cell dimensions and ΔR is the radial cell dimension. Note, that since Δr 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 Δr 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 (**??**) and (**??**) 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:

$$Q\text{-th exponential bin size} = s \cdot \Delta r \cdot \left[Q^t - (Q-1)^t\right] \tag{7.81}$$

$$Q\text{-th logarithmic bin size} = s \cdot \Delta r \cdot e^{t(Q-1)}. \tag{7.82}$$

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.

7.9.2.3 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 **??**. 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.

7.9.2.4 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 **??**) 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. [[api reference]]).

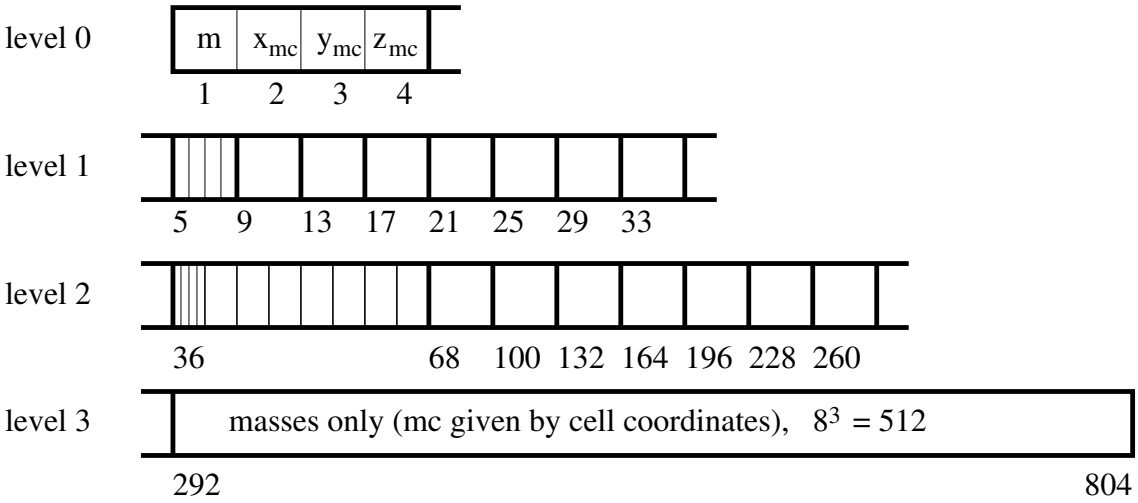

Figure 7.7: Example of a block-tree in case of `nxb`=`nyb`=`nzb`=8 and in case physical units do not store any further information to tree nodes (masses and mass centre positions are included by the tree solver itself).

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

$$\frac{S_{\mathrm{node}}}{D} < \texttt{gr\_bhTreeLimAngle} \ , \tag{7.83}$$

where S_{node} is the node size (defined as the largest edge of the corresponding cuboid) and D is the distance between the node and the *point-of-calculation*. Additionally, `gr_bhMAC` checks that the *point-of-calculation* is not located within the node itself enlarged by factor [[rpi reference]]. 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 [[rpi reference]] 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 [[api reference]], 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).

7.9.2.5 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.

7.9.2.6 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 `Flash-X`. The Flash Center adapted it to `Flash-X`.

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-X 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-X(**??**). This is overcome by the exchange of guardcells from coarse to fine using monotonic interpolation (**??**) and external boundary extrapolation for the calculation of the residual.

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.

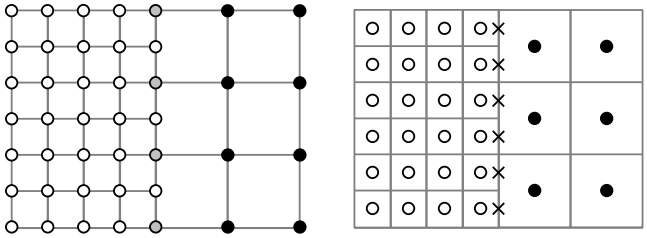

Figure 7.8: Contrast between jumps of refinement in meshes used in the original paper (left) and the oct-tree adapted method (right).

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 ℓ 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 ϕ 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:

$$R(\mathbf{x}) \equiv 4\pi G\rho(\mathbf{x}) - \nabla^2\tilde{\phi}(\mathbf{x}) \ . \tag{7.84}$$

This is accomplished through the application of the finite difference laplacian, defined on level ℓ with length-scales Δx_ℓ, Δy_ℓ and Δz_ℓ.

$$\mathcal{D}_\ell \tilde{\phi}^{b\ell}_{ijk} \equiv \frac{1}{\Delta x_\ell^2}\left(\tilde{\phi}^{b\ell}_{i+1,jk} - 2\tilde{\phi}^{b\ell}_{ijk} + \tilde{\phi}^{b\ell}_{i-1,jk}\right) + \frac{1}{\Delta y_\ell^2}\left(\tilde{\phi}^{b\ell}_{i,j+1,k} - 2\tilde{\phi}^{b\ell}_{ijk} + \tilde{\phi}^{b\ell}_{i,j-1,k}\right) \tag{7.85}$$

$$+\frac{1}{\Delta z_\ell^2}\left(\tilde{\phi}^{b\ell}_{ij,k+1} - 2\tilde{\phi}^{b\ell}_{ijk} + \tilde{\phi}^{b\ell}_{ij,k-1}\right) \ . \tag{7.86}$$

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

$$(\mathcal{R}_\ell \tilde{\phi})^{\mathcal{P}(c),\ell}_{ijk} \equiv \frac{1}{2^d}\sum_{i'j'k'} \tilde{\phi}^{c,\ell+1}_{i'j'k'} \ , \tag{7.87}$$

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:

$$(\mathcal{I}_\ell \tilde{\phi})^{c,\ell+1}_{i'j'k'} \equiv \sum_{p,q,r=-2}^{2} \alpha_{i'j'k'pqr}\tilde{\phi}^{\mathcal{P}(c),\ell}_{i+p,j+q,k+r} \tag{7.88}$$

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 α determine the interpolation scheme. For the $-x$ face in 3D,

$$\alpha_{1/2,j'k'pqr} = \beta_p\gamma_{j'q}\gamma_{k'r} \tag{7.89}$$

$$(\beta_p) = \left(-\frac{1}{12},\frac{7}{12},\frac{7}{12},-\frac{1}{12},0\right)$$

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

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.

7.9.2.7 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:

$$G^\ell_{ijk} = -16\pi G\left[\frac{1}{\Delta x_\ell^2}\sin^2\left(\frac{i\pi}{2n_x}\right) + \frac{1}{\Delta y_\ell^2}\sin^2\left(\frac{j\pi}{2n_y}\right) + \frac{1}{\Delta z_\ell^2}\sin^2\left(\frac{k\pi}{2n_z}\right)\right]^{-1} \ . \tag{7.90}$$

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:

$$\rho_{1jk} \to \rho_{1jk} - \frac{2}{\Delta x_\ell^2}\phi(x_{1/2}, y_j, z_k) \ . \tag{7.91}$$

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:

$$G^\ell_{ijk} = \begin{cases} -16\pi G\left[\frac{1}{\Delta x_\ell^2}\sin^2\left(\frac{(i-1)\pi}{n_x}\right) + \frac{1}{\Delta y_\ell^2}\sin^2\left(\frac{(j-1)\pi}{n_y}\right) + \frac{1}{\Delta z_\ell^2}\sin^2\left(\frac{(k-1)\pi}{n_z}\right)\right]^{-1} & \\ \qquad\qquad i,j, \text{ or } k \neq 1 & \\ 0 \qquad\qquad i=j=k=1 & \end{cases} \tag{7.92}$$

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 ℓ from 1 to $\ell_{\max}$,

   (a) Reset the grid so that ℓ is the maximum refinement level

   (b) Solve $\mathcal{D}_\ell \tilde{\phi}^{b\ell}_{ijk} = 4\pi G\rho^{b\ell}_{ijk}$ for all blocks b on level ℓ.

   (c) Compute the residual $R^{b\ell}_{ijk} = 4\pi G\rho^{b\ell}_{ijk} - \mathcal{D}_\ell \tilde{\phi}^{b\ell}_{ijk}$

   (d) For each block b on level ℓ that has children, prolong face values for $\tilde{\phi}^{b\ell}_{ijk}$ onto each child block.

3. *Residual propagation step:* Restrict the residual $R^{b\ell}_{ijk}$ to all levels.

4. *Correction step:* Compute the discrete L_2 norm of the residual over all leaf-node blocks and divide it by the discrete L_2 norm of the source over the same blocks. If the result is greater than a preset threshold value, proceed with a correction step: for each level ℓ from 1 to $\ell_{\max}$,

   (a) Reset the grid so that ℓ is the maximum refinement level

   (b) Solve $\mathcal{D}_\ell C^{b\ell}_{ijk} = R^{b\ell}_{ijk}$ for all blocks b on level ℓ.

   (c) Overwrite $R^{b\ell}_{ijk}$ with the new residual $R^{b\ell}_{ijk} - \mathcal{D}_\ell C^{b\ell}_{ijk}$ for all blocks b on level ℓ.

   (d) Correct the solution on all leaf-node blocks b on level ℓ: $\tilde{\phi}^{b\ell}_{ijk} \to \tilde{\phi}^{b\ell}_{ijk} + C^{b\ell}_{ijk}$.

   (e) For each block b on level ℓ that has children, interpolate face boundary values of $C^{b\ell}_{ijk}$ for each child.

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

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

7.9.2.8 A Hybrid Poisson Solver: Interfacing PFFT with Multigrid

We can improve the performance of the Multigrid solver in Section **??** 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-X 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.

7.9.3 Using the Poisson solvers

The `GridSolvers` subunit solves the Poisson equation ((**??**)). Two different elliptic solvers are supplied with Flash-X: 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 [[api reference]] 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*`)` ,

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

bcTypes	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 α multiplying the source function in ((**??**)).

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 interpolationat 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 [[rpi reference]] 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 [[api reference]], unphysical forces will be produced at refinement boundaries. Also, using constant or linear grid interpolation may cause the multigrid solver to fail to converge.

7.9.3.1 Multipole (original version)

The `poisson/multipole` sub-module takes two runtime parameters, listed in **??**. 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.

7.9.3.2 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
```

Table 7.3: Runtime parameters used with `poisson/multipole`.

Variable	Type	Default	Description
`mpole_lmax`	integer	10	Maximum multipole moment
`quadrant`	logical	`.false.`	Use symmetry to solve a single quadrant in 2D axisymmetric cylindrical (r, z) coordinates, instead of a half domain.

The improved multipole solver currently accepts only two setup parameters, either one switching on multithreading:

- **threadBlockList**: enables multithreaded compilation and execution.
- **threadWithinBlock**: enables multithreaded compilation and execution.

The names of these two setup parameters are missleading, 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.

- [[rpi reference]]: 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.
- [[rpi reference]]: 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.
- [[rpi reference]]: 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.
- [[rpi reference]]: 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.
- [[rpi reference]]: 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'.
- [[rpi reference]]: 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.

- [[rpi reference]]: 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 (¿ 1000 for domain refinement levels up to 5) one can enforce the code to use only non-statistical radial bins.

- [[rpi reference]]: 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.

- [[rpi reference]]: 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.

- [[rpi reference]]: 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.

- [[rpi reference]]: 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).

- [[rpi reference]]: 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 Δr.

- [[rpi reference]]: 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
[[rpi reference]]	integer	0	> 0
[[rpi reference]]	logical	false	true
[[rpi reference]]	logical	false	true
[[rpi reference]]	logical	false	true
[[rpi reference]]	logical	false	true
[[rpi reference]]	logical	false	true
[[rpi reference]]	integer	16	> 0
[[rpi reference]]	real	0.1	less than 0.1 and > 0.0
[[rpi reference]]	integer	1	> 1
[[rpi reference]]	real	1.0	less than 1.0 and > 0.0
[[rpi reference]]	string	"exponential"	"logarithmic"
[[rpi reference]]	real	1.0	> 0.0
[[rpi reference]]	real	1.0	> 0.0 (exponential)
	real	-	any $\neq 0$ (logarithmic)

7.9.3.3 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 [[rpi reference]] and [[rpi reference]] 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 [[rpi reference]] = `.false.` and [[rpi reference]] = `.true.`.

Runtime parameter [[rpi reference]] 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 [[rpi reference]].

Parameter [[rpi reference]] 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
[[rpi reference]]	logical	.false.	whether physical MAC should be used in tree walk
[[rpi reference]]	logical	.false.	whether physical MAC should be used in communication
[[rpi reference]]	real	0.5	limiting opening angle
[[rpi reference]]	real	1.2	relative size of restricted volume around node where the point-of-calculation is not allowed to be located
[[rpi reference]]	logical	.true.	whether Barnes-Hut like tree walk algorithm should be used
[[rpi reference]]	integer	.true.	maximum length of the priority queue

7.9.3.4 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-X, the multigrid solver will be used to solve for Gravity (see: **??**). 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 7.4: Runtime parameters used with `Grid/GridSolvers/Multigrid`.

Variable	Type	Default	Description
[[rpi reference]]	real	1×10^{-6}	Maximum ratio of the norm of the residual to that of the right-hand side
[[rpi reference]]	integer	100	Maximum number of iterations to take
[[rpi reference]]	real	`.true.`	Print the norm ratio per-iteration
[[rpi reference]]	integer	4	The number of multipole moments used in the isolated case

7.9.3.5 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 [[rpi reference]] should be set to 1.

The `Multigrid` runtime parameters given in the previous section also apply.

7.9.4 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 `Flash-X` the HYPRE linear algebra package is used to solve these systems of equations. Therefore it is necessary to install Hypre if this capability of Flash-X is to be used.

[[api reference]] 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 `Flash-X`)

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 `Flash-X`. Two terms commonly used in HYPRE terminology are part and box. We define these terms in equivalent `Flash-X` terminology. A HYPRE box object maps directly to a leaf block in `Flash-X`. The block is then defined by it's extents. In `Flash-X` 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 `Flash-X` 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 then 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 `Flash-X` grid, whenever the `Flash-X` grid changes the HYPRE grid object needs to be updated. Consequentlywith 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 **??** for details). UG does not require any graph relationships.

Whether a block needs a graph relationship depends on the refinement level of it's 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 to way to achieve the desired effect.

With the computation of Vector B (RHS), the system can be solved using HYPRE and UNK can be updated.

7.9.4.1 HYPRE Solvers

In `Flash-X` we use the HYPRE PARCSR storage format as this exposes the maximum number of iterative solvers.

Table 7.5: 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 `Flash-X` 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 `Flash-X` 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 7.6: Runtime parameters used with `Grid/GridSolvers/HYPRE`.

Variable	Type	Default	Description
`gr_hyprePCType`	string	`"hypre_amg"`	Algebraic Multigrid as Preconditioner
`gr_hypreMaxIter`	integer	10000	Maximum number of iterations
`gr_hypreRelTol`	real	1×10^{-8}	Maximum ratio of the norm of the residual to that of the initial residue
`gr_hypreSolverType`	string	`"hypre_pcg"`	Type of linear solver, Preconditioned Conjugate gradient
`gr_hyprePrintSolveInfo`	boolean	FALSE	enables/disables some basic solver information (for e.g number of iterations)
`gr_hypreInfoLevel`	integer	1	Verbosity level of solver diagnostics (refer HYPRE manual).
`gr_hypreFloor`	real	1×10^{-12}	Used to floor the end solution.
`gr_hypreUseFloor`	boolean	TRUE	whether to apply gr_hypreFloor to floor results from HYPRE.

7.10 Grid Geometry

Flash-X 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 (**??**). 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 geometryrequires 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-X are finite-volume methods, so the interpolationmust also be conservative in the given geometry. The default mesh refinement criteria of `Flash-X` also currently take geometry into account, see **??** above.

Table 7.7: Different geometry types. For each geometry/dimensionality combination, the "support" column lists the `Grid` implementations that support it: pm4 stands for `PARAMESH` 4.0 and `PARAMESH` 4dev, pm2 for `PARAMESH` 2, UG for Uniform Grid implementations, respectively.

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	ϕ
spherical	1	pm4,pm2,UG	y	r		
spherical	2	pm4,pm2,UG	y	r	θ	
spherical	3	pm4,pm2,UG	n	r	θ	ϕ
polar	1	pm4,UG	y	r		
polar	2	pm4,pm2,UG	n	r	ϕ	
"polar + z" (cylindrical with a different ordering of coordinates)	3	—	n	r	ϕ	z

A **convention:** in this section, Small letters x, y, and z are used with their usual meaning in designating coordinate directions for specific coordinate systems: *i.e.*, x and y refer to directions in Cartesian coordinates, and z may refer to a (linear) direction in either Cartesian or cylindrical coordinates.

On the other hand, capital symbols X, Y, and Z are used to refer to the (up to) three directions of any coordinate system, *i.e.*, the directions corresponding to the `IAXIS`, `JAXIS`, and `KAXIS` dimensions in `Flash-X`, respectively. Only in the Cartesian case do these correspond directly to their small-letter counterparts. For other geometries, the correspondence is given in **??**.

7.10.1 Understanding 1D, 2D, and Curvilinear Coordinates

In the context of Flash-X, 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 threedimensional reality? Physical reality has three spatial dimensions (as far as the physical problems simulated with Flash-X are concerned). In Cartesian coordinates, it is relatively straightforward to understand what a 2-d (or 1-d) simulation means: "Just leave out one (or two) coordinates." This is less obvious for other coordinate systems, therefore some fundamental discussion follows.

A reduced dimensionality (RD) simulation can be naively understood as taking a cut (or, for 1-d, a linear probe) through the real 3-d problem. However, there is also an assumption, not always explicitly stated, that is implied in this kind of simulation: namely, that the cut (or line) is representative of the 3-d problem. This can be given a stricter meaning: it is assumed that the physics of the problem do not depend on the omitted dimension (or dimensions). A RD simulation can be a good description of a physical system only to the degree that this assumption is warranted. Depending on the nature of the simulated physical system, non-dependence on the omitted dimensions may mean the absence of force and/or momenta vector components in directions of the omitted coordinate axes, zero net mass and energy flow out of the plane spanned by the included coordinates, or similar.

For omitted dimensions that are lengths — z and possibly y in Cartesian, and z in cylindrical and polar RD simulations — one may think of a 2-d cut as representing a (possibly very thin) layer in 3-d space sandwiched between two parallel planes. There is no *a priori* definition of the thickness of the layer, so it is not determined what 3-d volume should be asigned to a 2-d cell in such coordinates. We can thus arbitrarily assign the length "1" to the edge length of a 3-d cell volume, making the volume equal to the 2-d area. We can understand generalizations of "volume" to 1-d, and of "face areas" to 2-d and 1-d RD simulations with omitted linear coordinates, in an equivalent way: just set the length of cell edges along omitted dimensions to 1.

For omitted dimensions that are angles — the θ and ϕ coordinates on spherical, cylindrical, and polar geometries — it is easier to think of omitting an angle as the equivalent of integrating over the full range of that angle coordinate (under the assumption that all physical solution variables are independent of that angle). Thus omitting an angle ϕ in these geometries implies the assumption of axial symmetry, and this is noted in **??**. Similarly, omitting both ϕ and θ in spherical coordinates implies an assumption of complete spherical symmetry. When ϕ is omitted, a 2-d cell actually represents the 3-d object that is generated by rotating the 2-d area around a z-axis. Similarly, when only r is included, 1-d cells (*i.e.*, r intervals) represent hollow spheres or cylinders. (If the coordinate interval begins at $r_l = 0.0$, the sphere or cylinder is massive instead of hollow.)

As a result of these considerations, the measures for cell (and block) volumes and face areas in a simulation depends on the chosen geometry. Formulas for the volume of a cell dependent on the geometry are given in the geometry-specific sections further below.

As discussed in **??**, 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.

7.10.1.1 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 ρ, including their physical values and units, so that material density ρ 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-X.

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 threedimensional cell of volume $\Delta x \Delta y \Delta z$ where the cell height Δz 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-X 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).

7.10.2 Choosing a Geometry

The user chooses a geometry by setting the [[rpi reference]] 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 **??**.

The [[rpi reference]] 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 [[rpi reference]] runtime parameter is inconsistent with a geometry specified at setup time, Flash-X 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.

7.10.3 Getting Geometry Information in Program Code

The `Grid` unit provides an accessor [[api reference]] 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 **??**). 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 [[api reference]] routine. This routine can provide the coordinates of cells at the left edge, right edge, or center. The width of cells can be determined via the [[api reference]] routine. Angle values and differences are given in radians. Coordinate information for a block of cells as a whole is available through [[api reference]] and [[api reference]].

The volume of a single cell can be obtained via the [[api reference]] or the [[api reference]] routine. Use the [[api reference]], [[api reference]], or [[api reference]] 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.

7.10.4 Available Geometries

Currently, all of Flash-X'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-X `Grid` unit and PPM hydrodynamics unit (**??**), support additional geometries, such as two-dimensional cylindrical (r, z) grids, one/two-dimensional spherical $(r)/(r, \theta)$ grids, and two-dimensional polar (r, ϕ) 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 $\Delta r = r_r - r_l$;

```
#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 7.9: Branching based on geometry type

and similar for the other coordinates.

7.10.4.1 Cartesian geometry

Flash-X 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	Δx
2-d	$\Delta x \Delta y$
3-d	$\Delta x \Delta y \Delta z$

7.10.4.2 Cylindrical geometry

To run Flash-X with cylindrical coordinates, the `geometry` parameter must be set thus:

```
geometry = "cylindrical"
```

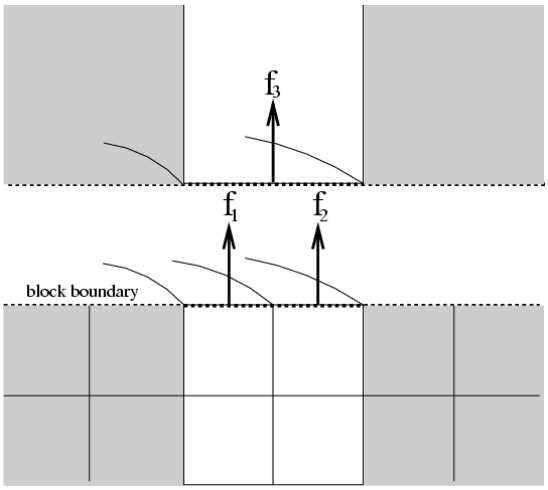

Figure 7.10: Diagram showing two fine cells and a coarse cell at a jump in refinement in the cylindrical 'z' direction. The block boundary has been cut apart here for illustrative purposes. The fluxes out of the fine blocks are shown as f_1 and f_2. These will be used to compute a more accurate flux entering the coarse flux f_3. The area that the flux passes through is shown as the annuli at the top of each fine cell and the annulus below the coarse cell.

Cell Volume in Cylindrical Coordinates

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, **??** 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

$$A = \pi(r_r^2 - r_l^2) \ . \tag{7.93}$$

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

$$f_3 = \frac{A_1 f_1 + A_2 f_2}{A_3} \ . \tag{7.94}$$

For fluxes in the radial direction, the cross-sectional area is independent of the height, z, so the corrected flux is simply taken as the average of the flux densities in the adjacent finer cells.

7.10.4.3 Spherical geometry

One or two dimensional spherical problems can be performed by specifying

```
geometry = "spherical"
```

in the runtime parameter file.

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.

7.10.4.4 Polar geometry

Polar geometry is a 2-D subset of 3-D cylindrical configuration without the "z" coordinate. Such geometry is natural for studying objects like accretion disks. This geometry can be selected by specifying

```
geometry = "polar"
```

in the runtime parameter file.

Cell Volume in Polar Coordinates

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.

7.10.5 Conservative Prolongation/Restriction on Non-Cartesian Grids

When blocks are refined, we need to initialize the child data using the information in the parent cell in a manner which preserves the cell-averages in the coordinate system we are using. When a block is derefined, the parent block (which is now going to be a leaf block) needs to be filled using the data in the child blocks (which are soon to be destroyed). The first procedure is called prolongation.The latter is called restriction.Both of these procedures must respect the geometry in order to remain conservative. Prolongation and restriction are also used when filling guard cells at jumps in refinement.

7.10.5.1 Prolongation

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

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

These paths will be be automatically added by the `setup` script when the `-gridinterpolation=monotonic` option is in effect (which is the case by default, unless `-gridinterpolation=native` was specified). The "monotonic" interpolationscheme used in both cases is taking geometry into consideration and is appropriate for all supported geometries.

Flash-X Transition

Some more specific PARAMESH 2 interpolation schemes are included in the distribution and might be useful for compatibility with Flash-X:

- 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.

7.10.5.2 Restriction

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

$$\langle f \rangle_{i,j} = \frac{V_{ic,jc} \langle f \rangle_{ic,jc} + V_{ic+1,jc} \langle f \rangle_{ic+1,jc} + V_{ic,jc+1} \langle f \rangle_{ic,jc+1} + V_{ic+1,jc+1} \langle f \rangle_{ic+1,jc+1}}{V_{i,j}} \ , \tag{7.95}$$

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

7.11 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 8

IO Unit

Flash-X 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-X data-structures to records in these files is controlled by the Flash-X IO unit. Flash-X can output data with two parallel IO libraries, HDF5and Parallel-NetCDF. The data layout is different for each of these libraries. Since `Flash-X` we also offer direct serial FORTRAN IO, which can be used as a last resort if no parallel library is available. However, Flash-X post-processing tools such as fidlr (**??**) and sfocu (**??**) 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-X 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.

8.1 IO Implementations

`Flash-X` supports multiple IO implementations: direct, serial and parallel implementations as well as support for different parallel libraries. In addition, `Flash-X` also supports multiple (**??**) `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 `Flash-X` is quite 'smart' and will not let the user setup a problem with incompatible `IO` and `Grid` unit implementations. **??** summarizes the different implementation of the Flash-X IO unit in the current release.

Table 8.1: IO implementations available in Flash-X. All implementations begin at the /source directory.

Implementation Path	Description
IO/IOMain/HDF5/parallel/PM	Hierarchical Data Format (HDF) 5 output. A single HDF5 file is created, with each processor writing its data to the same file simultaneously. This relies on the underlying MPIIO layer in HDF5. This particular implementation only works with the PARAMESH grid package.
IO/IOMain/hdf5/parallel/UG	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.
IO/IOMain/hdf5/parallel/NoFbs	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.
IO/IOMain/hdf5/serial/PM	Hierarchical Data Format (HDF) 5 output. Each processor passes its data to processor 0 through explicit MPI sends and receives. Processor 0 does all of the writing. The resulting file format is identical to the parallel version; the only difference is how the data is moved during the writing. This particular implementation only works with the PARAMESH grid package.
IO/IOMain/hdf5/serial/UG	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.
IO/IOMain/pnetcdf/PM	ParallelNetCDF output. A single file is created with each processor writing its data to the same file simultaneously. This relies on the underlying MPI-IO layer in PNetCDF. This particular implementation only works with the PARAMESH grid package.
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-X tools are not compatible with the direct IO unit.

Table 8.1: Flash-X IO implementations (continued).

Implementation path	Description
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-X tools are not compatible with the direct IO unit.

Flash-X also comes with some predefined setup **shortcuts**which make choosing the correct IO significantly easier; see **??** for more details about shortcuts. In Flash-X HDF5 serial IO is included by default. Since PARAMESH 4.0 is the default grid, the included IO implementations will be compatible with PARAMESH 4.0. For clarity, a number or examples are shown below.

An example of a basic setup with HDF5 serial IO and the PARAMESH grid, (both defaults) is:

```
./setup Sod -2d -auto
```

To include a parallel implementation of HDF5 for a PARAMESH grid the setup syntax is:

```
./setup Sod -2d -auto -unit=IO/IOMain/hdf5/parallel/PM
```

using the already defined shortcuts the setup line can be shortened to

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

To set up a problem with the Uniform Grid and HDF5 serial IO, the setup line is:

```
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/serial/UG
```

using the already defined shortcuts the setup line can be shortened to

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

To set up a problem with the Uniform Grid and HDF5 parallel IO, the complete setup line is:

```
./setup Sod -2d -auto -unit=Grid/GridMain/UG -unit=IO/IOMain/hdf5/parallel/UG
```

using the already defined shortcuts the setup line can be shortened to

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

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

```
./setup Sod -2d -auto +noio
```

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

```
-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-X code architecture, the F90 module `IO_data` stores all the data with `IO` unit scope. The routine [[api reference]] is called once by [[api reference]] and initializes `IO` data and stores any runtime parameters. See **??**.

8.2 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-X also outputs a logfile, but this file is controlled by the Logfile Unit; see **??** 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 [[rpi reference]]IO/all of them. Additional description is located in **??** for checkpoint parameters, **??** for plotfile parameters, **??** for particle file parameters, **??** for flash.dat parameters, and **??** for genereal IO parameters.

8.2.1 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-X. Once the problem is fixed, a simulation can be restarted from the last checkpoint file rather than the beginning of the run. A checkpoint filecontains 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 [[api reference]]. Users usually will not need to call this routine directly because the Flash-X IO unit calls `IO_writeCheckpoint` from the routine [[api reference]] 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-X 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 [[rpi reference]], the minimum cosmological redshift, or the total number of steps [[rpi reference]] 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-X 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-X 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 8.2: Checkpoint IO parameters.

Parameter	Type	Default value	Description
checkpointFileNumber	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.
checkpointFileIntervalStep	INTEGER	0	The number of timesteps desired between subsequent checkpoint files.
checkpointFileIntervalTime	REAL	1.	The amount of simulation time desired between subsequent checkpoint files.
checkpointFileIntervalZ	REAL	HUGE(1.)	The amount of cosmological redshift change that is desired between subsequent checkpoint files.
rolling_checkpoint	INTEGER	10000	The number of checkpoint files to keep available at any point in the simulation. If a checkpoint number is greater than rolling_checkpoint, then the checkpoint number is reset to 0. There will be at most rolling_checkpoint checkpoint files kept. This parameter is intended to be used when disk space is at a premium.
wall_clock_checkpoint	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 wall_clock_checkpoint seconds elapse over the course of the simulation, a checkpoint file is stored. This is useful for ensuring that a checkpoint file is produced before a queue closes.
restart	BOOLEAN	.false.	A logical variable indicating whether the simulation is restarting from a checkpoint file (.true.) or starting from scratch (.false.).

Flash-X 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-X to restart, set the [[rpi reference]] runtime parameter to .true. in the flash.par. Also, set [[rpi reference]] to the number of the file from which you wish to restart. If plotfiles or particle files are being produced set [[rpi reference]] and [[rpi reference]] to the number of the *next* plotfile and particle file you want Flash-X to output. In Flash-X 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 **??** for more details on plotfiles.

8.2.2 Plotfiles

A plotfilecontains all the information needed to interpret the grid data maintained by Flash-X. 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 [[rpi reference]], [[rpi reference]], *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 `??`. At this time, the plotting of face centered quantities is not supported.

Flash-X Transition

In `Flash-X` a few variables like density and pressure were output to the plotfiles by default. Because `Flash-X` supports a wider range of simulations, it makes no assumptions that density or pressure variables are even included in the simulation. In `Flash-X` 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 [[api reference]]. As with checkpoint files, the user will not need to call this routine directly because it is invoked indirectly through calling [[api reference]] when, based on runtime parameters, `Flash-X` needs to write a plotfile. Flash-X 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-X 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 8.3: Plotfile IO parameters.

Parameter	Type	Default value	Description
`plotFileNumber`	`INTEGER`	`0`	The number of the starting (or restarting) plotfile. This number is appended to the filename.
`plotFileIntervalTime`	`REAL`	`1.`	The amount of simulation time desired between subsequent plotfiles.
`plotFileIntervalStep`	`INTEGER`	`0`	The number of timesteps desired between subsequent plotfiles.

Table 8.3: Plotfile IO parameters (continued).

Parameter	Type	Default value	Description
plotFileIntervalZ	INTEGER	HUGE(1.)	The change in cosmological redshift desired between subsequent plotfiles.
plot_var_1, ..., plot_var_n	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.
ignoreForcedPlot	BOOLEAN	.false.	A logical variable indicating whether or not to denote certain plotfiles as forced.
forcedPlotfileNumber	INTEGER	0	An integer that sets the starting number for a forced plotfile.
plotfileMetadataDP	BOOLEAN	.false.	A logical variable indicating whether or or not to output the normally single-precision grid metadata fields as double precision in plotfiles. This specifically affects coordinates, block size, and bounding box.
plotfileGridQuantityDP	BOOLEAN	.false.	A logical variable that sets whether or not quantities stored on the grid, such as those stored in unk, are output in single precision or double precision in plotfiles.

8.2.3 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 [[api reference]]. 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 8.4: Particle File IO runtime parameters.

Parameter	Type	Default value	Description
particleFileNumber	INTEGER	0	The number of the starting (or restarting) particle file. This number is appended to the end of the filename.
particleFileIntervalTime	REAL	1.	The amount of simulation time desired between subsequent particle file dumps.

Table 8.4: Particle File IO parameters (continued).

Parameter	Type	Default value	Description
particleFileIntervalStep	INTEGER	0	The number of timesteps desired between subsequent particle file dumps.
particleFileIntervalZ	REAL	HUGE(1.)	The change in cosmological redshift desired between subsequent particle file dumps.

Flash-X Transition

From Flash-X on each particle dump is written to a separate file. In Flash-X the particles data structure was broken up into real and integer parts, where as in Flash-X all particle properties are real values. See ?? and ?? for more information about the particles data structure in Flash-X. Additionally, filtered particles are not implemented in Flash-X.

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

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

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

8.2.4 Integrated Grid Quantities – flash.dat

At each simulation time step, values which represent the overall state (*e.g.*, total energy and momentum) are computed by calculating over all cells in the computations domain. These integral quantities are written to the ASCI file flash.dat. A default routine [[api reference]] 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 8.5: flash.dat runtime parameters.

Parameter	Type	Default value	Description
stats_file	STRING	"flash.dat"	Name of the file to which the integral quantities are written.
wr_integrals_freq	INTEGER	1	The number of timesteps to elapse between outputs to the scalar/integral data file (flash.dat)

8.2.5 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 8.6: General IO runtime parameters.

Parameter	Type	Default value	Description
basenm	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 **??** for a description of how Flash-X output files are named.
output_directory	STRING	""	Output directory for plotfiles, particle files and checkpoint files. The default is the directory in which the executable sits. output_directory can be an absolute or relative path.
memory_stat_freq	INTEGER	100000	The number of timesteps to elapse between memory statistic dumps to the log file (flash.log).
useCollectiveHDF5	BOOLEAN	.true.	When using the parallel HDF5 implementation of IO, will enable collective mode for HDF5.
summaryOutputOnly	BOOLEAN	.false.	When set to .true. write an integrated grid quantities file only. Checkpoint, plot and particle files are not written unless the user creates a .dump_plotfile, .dump_checkpoint, .dump_restart or .dump_particle file.

8.3 Restarts and Runtime Parameters

Flash-X 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 Flash-X and any simulation parameter files (*e.g.*, flash.par). If needed, the previous values from the checkpoint file can be obtained using the routines [[api reference]].

8.4 Output Scalars

In Flash-X, each unit has the opportunity to request scalar data to be output to checkpoint or plotfiles. Because there is no central database, each unit "owns" different data in the simulation. For example, the Driver unit owns the timestep variable dt, the simulation variable simTime, and the simulation step number nStep. The Grid unit owns the sizes of each block, nxb, nyb, and nzb. The IO unit owns the variable

checkpointFileNumber. Each of these quantities are output into checkpoint files. Instead of hard coding the values into checkpoint routines, Flash-X 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.*, [[api reference]] and [[api reference]]. These routines in turn call [[api reference]]. For example, the routine [[api reference]] 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 [[api reference]] 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 [[api reference]]. 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 [[api reference]] and terminate the run.

8.5 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 Flash-X any additional information had to be hard coded into the simulation. In Flash-X, we have provided a general interface [[api reference]] and [[api reference]] 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.

8.6 Output Scratch Variables

In Flash-X 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 **??**). To output these scratch variables, the user only needs to set the values of the runtime parameters [[rpi reference]], [[rpi reference]], *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.

8.7 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.

8.8 Output Filenames

Flash-X constructs the output filenamesbased 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:

$$\texttt{basename\_s0000\_}\left\{\begin{array}{c}\texttt{hdf5}\\ \texttt{ncmpi}\end{array}\right\}\texttt{\_chk\_0000}\ ,$$

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:

$$\texttt{basename\_s0000\_}\left\{\begin{array}{c}\texttt{hdf5}\\ \texttt{ncmpi}\end{array}\right\}\texttt{\_plt\_}\left\{\begin{array}{c}\texttt{crn}\\ \texttt{cnt}\end{array}\right\}\texttt{\_0000}\ ,$$

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 `Flash-X`'s IO.

Flash-X Transition

In `Flash-X` the correct format of the names of the checkpoint, plotfile and particle file were necessary in order to read the files with the Flash-X fidlr visualization tool. In `Flash-X` the name of the file is irrelevant to fidlr3.0 (see **??**). 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.

8.9 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 `Flash-X` 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 (**??**), do not currently support the direct IO implementation, and the output files from this mode are not portable across platforms.

8.9.1 HDF5

HDF5is 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 `Flash-X` 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 `HDF5IO` 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-X 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 `Flash-X`, the user can set up shortcutsSee **??** 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-X 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 **??** 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: [[rpi reference]] and [[rpi reference]]. When these runtime parameters are true, the `Flash-X` 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 `Flash-X`.

Flash-X Transition

We recommend that HDF5 version 1.6 or later be used with the HDF5 IO implementations with `Flash-X`. 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-X 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
```

8.9.1.1 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. `Flash-X` can make use of this mode by setting the runtime parameter `useCollectiveHDF5` to true.

8.9.1.2 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.

8.9.1.3 HDF5 Data Format

The HDF5 data format for `Flash-X` is identical to `Flash-X` 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-X'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 [[api reference]]. See **??** for more details. The `Flash-X` HDF5 format is summarized in **??**.

Table 8.7: Flash-X HDF5 file format.

Record label	Description of the record
Simulation Meta Data: included in all files	
sim info	Stores simulation meta data in a user defined C structure. Structure datatype and attributes of the structure are described below.

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

sim_info_t sim_info;
```

`sim_info.file_format_version`:	An integer giving the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file.
`sim_info.setup_call`:	The complete syntax of the `setup` command used when creating the current Flash-X executable.
`sim_info.file_creation_time`:	The time and date that the file was created.

Table 8.7: HDF5 format (continued).

Record label	Description of the record
`sim_info.flash_version`:	The version of Flash-X used for the current simulation. This is returned by routine `setup_flashVersion`.
`sim_info.build_date`:	The date and time that the Flash-X executable was compiled.
`sim_info.build_dir`:	The complete path to the Flash-X root directory of the source tree used when compiling the Flash-X executable. This is generated by the subroutine `setup_buildstats` which is created at compile time by the Makefile.
`sim_info.build_machine`:	The name of the machine (and anything else returned from `uname -a`) on which Flash-X was compiled.
`sim_info.cflags`:	The c compiler flags used in the given simulation. The routine `setup_buildstats` is written by the `setup` script at compile time and also includes the `fflags` below.
`sim_info.fflags`:	The f compiler flags used in the given simulation.
`sim_info.setup_time_stamp`:	The date and time the given simulation was setup. The routine `setup_buildstamp` is created by the `setup` script at compile time.
`sim_info.build_time_stamp`:	The date and time the given simulation was built. The routine `setup_buildstamp` is created by the `setup` 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.

Table 8.7: HDF5 format (continued).

Record label	Description of the record

	`typedef struct int_list_t {`
	`  char name[MAX_STRING_LENGTH];`
	`  int value;`
	`} int_list_t;`
	
	`typedef struct real_list_t {`
	`  char name[MAX_STRING_LENGTH];`
	`  double value;`
	`} real_list_t;`
	
	`typedef struct str_list_t {`
	`  char name[MAX_STRING_LENGTH];`
	`  char value[MAX_STRING_LENGTH];`
	`} str_list_t;`
	
	`typedef struct log_list_t {`
	`  char name[MAX_STRING_LENGTH];`
	`  int value;`
	`} log_list_t;`
	
	`int_list_t  *int_list;`
	`real_list_t  *real_list;`
	`str_list_t  *str_list;`
	`log_list_t  *log_list;`

integer runtime parameters	`int_list_t int_list(numIntParams)` A linked list holding the names and values of all the integer runtime parameters.
real runtime parameters	`real_list_t real_list(numRealParams)` A linked list holding the names and values of all the real runtime parameters.
string runtime parameters	`str_list_t str_list(numStrParams)` A linked list holding the names and values of all the string runtime parameters.
logical runtime parameters	`log_list_t log_list(numLogParams)` A linked list holding the names and values of all the logical runtime parameters.
integer scalars	`int_list_t int_list(numIntScalars)` A linked list holding the names and values of all the integer scalars.
real scalars	`real_list_t real_list(numRealScalars)`

Table 8.7: HDF5 format (continued).

Record label	Description of the record
	A linked list holding the names and values of all the real scalars.
string scalars	`str_list_t str_list(numStrScalars)`
	A linked list holding the names and values of all the string scalars.
logical scalars	`log_list_t log_list(numLogScalars)`
	A linked list holding the names and values of all the logical scalars.
Grid data: included only in checkpoint files and plotfiles	
unknown names	`character*4 unk_names(nvar)`
	This array contains four-character names corresponding to the first index of the `unk` array. They serve to identify the variables stored in the 'unknowns' records.
refine level	`integer lrefine(globalNumBlocks)`
	This array stores the refinement level for each block.
node type	`integer nodetype(globalNumBlocks)`
	This array stores the node type for a block. Blocks with node type 1 are leaf nodes, and their data will always be valid. The leaf blocks contain the data which is to be used for plotting purposes.
gid	`integer gid(nfaces+1+nchild,globalNumBlocks)`
	This is the global identification array. For a given block, this array gives the block number of the blocks that neighbor it and the block numbers of its parent and children.
coordinates	`real coord(mdim,globalNumBlocks)`
	This array stores the coordinates of the center of the block. `coord(1,blockID)` = x-coordinate `coord(2,blockID)` = y-coordinate `coord(3,blockID)` = z-coordinate
block size	`real size(mdim,globalNumBlocks)`
	This array stores the dimensions of the current block. `size(1,blockID)` = x size `size(2,blockID)` = y size `size(3,blockID)` = z size
bounding box	`real bnd_box(2,mdim,globalNumBlocks)`

Table 8.7: HDF5 format (continued).

Record label	Description of the record
	This array stores the minimum (`bnd_box(1,:,:)`) and maximum (`bnd_box(2,:,:)`) coordinate of a block in each spatial direction.
which child (*Paramesh4.0 and Paramesh4dev only!*)	`integer which_child(globalNumBlocks)` An integer array identifying which part of the parents' volume this child corresponds to.
variable	`real unk(nxb,nyb,nzb,globalNumBlocks)` `nx` = number of cells/block in x `ny` = number of cells/block in y `nz` = number of cells/block in z This array holds the data for a single variable. The record label is identical to the four-character variable name stored in the record *unknown names*. Note that, for a plot file with `CORNERS=.true.` in the parameter file, the information is interpolated to the cell corners and stored.
Particle Data: included in checkpoint files and particle files	
localnp	`integer localnp(globalNumBlocks)` This array holds the number of particles on each processor.
particle names	`character*24 particle_labels(NPART_PROPS)` 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.
tracer particles	`real particles(NPART_PROPS, globalNumParticles` Real array holding the particles data structure. The first dimension holds the various particle properties like, velocity, tag etc. The second dimension is sized as the total number of particles in the simulation. Note that all the particle properties are real values.

8.9.1.4 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 [[rpi reference]] 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.

8.9.2 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-X 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 shortcutsfor various implementations. See **??** for information about creating shortcuts.

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

8.9.3 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
```

8.9.4 Output Side Effects

In `Flash-X` when plotfiles or checkpoint files are output by [[api reference]], the grid is fully restricted and user variables are computed prior to writing the file. [[api reference]] and [[api reference]] by default, do not do this step themselves. The restriction can be forced for all writes by setting runtime parameter [[rpi reference]] to true and the user variables can always be computed prior to output by setting [[rpi reference]] to true.

8.10 Working with Output Files

The checkpoint file output formats offer great flexibility when visualizing the data. The visualization program does not have to know the details of how the file was written; rather it can query the file to find the number of dimensions, block sizes, variable data etc that it needs to visualize the data. `IDL` routines for reading HDF5 and PnetCDF formats are provided in `tools/fidlr3/`. These can be used interactively though the `IDL` command line (see **??**). In addition, ViSit version 10.0 and higher (see **??**) can natively read `Flash-X` HDF5 output files by using the command line option `-assume_format Flash-X`.

8.11 Unit Test

The IO unit test is provided to test IO performance on various platforms with the different Flash-X IO implementations and parallel libraries.

Flash-X Transition

The IO unit test replaces the simulation setup io_benchmark in Flash-X.

The unitTest is setup like any other Flash-X 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-X simulation:

```
mpirun -np numProcs flash3
```

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

- The Config file in unitTest/IO declares some dummy grid scope variables which are stored in the unk array. If the user wants a more intensive IO test, more variables can be added. Variables are initialized to dummy values in Driver_evolveFlash.
- Variables will only be output to the plotfile if they are declared in the flash.par (see the example flash.par in the unit test).
- The only units besides the simulation unit included in this simulation are Grid, IO, Driver, Timers, Logfile, RuntimeParameters and PhysicalConstants.
- If the PARAMESH Grid implementation is being used, it is important to note that the grid will not refine on its own. The user should set lrefine_min to a value > 1 to create more blocks. The user could also set the runtime parameters nblockx, nblocky, nblockz to make a bigger problem.
- Just like any other simulation, the user can change the number of zones in a simulation using -nxb=-numZones on the setup line.

8.12 Derived data type I/O

In `Flash-X` 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-X 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-X application by adding the setup shortcuts `+hdf5TypeIO` for HDF5 and `+pnetTypeIO` for Parallel-NetCDF to your setup line. If you are using the HDF5 implementation then you need a parallel installation of HDF5. All of the runtime parameters introduced in this chapter should be compatible with derived data type I/O.

A nice property of derived data type I/O is that it eliminates a lot of the I/O code duplication which has been spreading in the Flash-X I/O unit over the last decade. The same code is used for UG, NoFBS and Paramesh Flash-X 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-X Astrophysics Code" `www.mcs.anl.gov/uploads/cels/papers/P1819.pdf`

The new I/O code has been tested in our internal Flash-X regression tests from before the `Flash-X` release and there are no known issues, however, it will probably be in the release following `Flash-X` 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-X applications, however, the code still needs a clean up and exhaustive testing with all the Flash-X runtime parameters introduced in this chapter.

Chapter 9

Runtime Parameters Unit

The `RuntimeParameters` Unit stores and maintains a global linked lists of runtime parameters that are used during program execution. Runtime parameters can be added to the lists, have their values modified, and be queried. This unit handles adding the default runtime parameters to the lists as well as reading any overwritten parameters from the `flash.par` file.

9.1 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 **??** for more information on declaring parameters in a `Config` file.

9.2 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 simulation's output log. For example, the RuntimeParameters section of the output log shown in **??** indicates that [[rpi reference]] and [[rpi reference]] have been read in from `flash.par` and are different than the default values, whereas the runtime parameter [[rpi reference]] 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 **??** for an example listing.

9.3 Routine Descriptions

The Runtime Parameters unit is included by default in all of the provided Flash-X simulation examples, through a dependence within the `Driver` unit. The main Flash-X initialization routine ([[api reference]]) and

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

Figure 9.1: 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 9.2: Portion of a `setup_params` file from an object directory.

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, [[api reference]]. This routine retrieves the value of a parameter stored in the linked list in the `RuntimeParameters_data` module. In `Flash-X` 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 [[api reference]]. For more documentation on the Flash-X code architecture please see **??**. 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 [[api reference]] 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-X code will call [[api reference]] 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-X need to access parts of the RuntimeParameters interface. For example, the input/output unit `IO` needs [[api reference]]. There are no user-defined parameters which affect the `RuntimeParameters` unit.

Flash-X Transition

Flash-X no longer distinguishes between contexts as in `Flash-X`. All runtime parameters are stored in the same context, so there is no need to pass a 'context' argument.

9.4 Example Usage

An implementation example from the [[api reference]] 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 RuntimeParamters_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 [[rpi reference]], 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 10

Multispecies Unit

Flash-X 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 **??**.

10.1 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 **??**, `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 whehter the simulation includes the `Multispecies` unit or not). However, if `NSPECIES`= 0, `SPECIES_END` will be less than `SPECIES_BEGIN`, and then neither of them should be used as an index into solution vectors.

As an illustration, Figures **??** and **??** are snippets from a configuration file and the corresponding section of the Flash-X header file, respectively. For more information on `Config` files, see **??**; for more information on the `setup` procedure, see **??**; for more information on the structure of the main header file `Flash.h`, see **??**.

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

Figure 10.1: 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 10.2: Sample excerpt from header file `Flash.h` showing integer definition of fluid species.

Table 10.1: 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

Flash-X Transition

In `Flash-X`, you found the integer index of a species by using `find_fluid_index`. In `Flash-X`, 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 **??** 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.

10.2 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 [[api reference]]. An example procedure is shown below and

```
  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 10.3: A `Simulation_initSpecies.F90` file showing Multispecies initialization

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 **??** is a valid input to the Multispecies unit.

Flash-X 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.

10.3 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-X 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. **??** 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.

10.4 Alternative Method for Defining Species

?? described how species can be defined by using the `SPECIES` keyword in the `Config` file. **??** then described how the properties of the species can be set using various subroutines defined in the `Multispecies` unit.

```
#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 10.4: A `Simulation_initSpecies.F90` file showing Multispecies initialization

Table 10.2: 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

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.

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 ??. The runtime parameter names contain the species name. ?? 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.

10.5 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.

- [[api reference]] This routine sets the value species property. It should be called within the subroutine [[api reference]] 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-X Transition

In `Flash-X`, you could set multiple properties at once in a call to `add_fluid_to_db`. In `Flash-X`, individual calls are required. If you are setting up a nuclear network, there is a precoded [[api reference]] to easily initialize all necessary species. It is located in the unit `Simulation/SimulationComposition`, which must be listed in your simulation `Config` file.

- [[api reference]] Returns the value of a requested property.

- [[api reference]] 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, ..., γ_i) is denoted $\mathcal{P}_i$, the sum calculated is
$$\sum_i X_i \mathcal{P}_i \quad .$$

- [[api reference]] 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, ..., γ_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.

- [[api reference]] 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, ..., γ_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
$$\frac{1}{\bar{A}} = \sum_i \frac{X_i}{A_i} \ , \tag{10.1}$$
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-X. This method allows some of the mass fractions to be zero.

- [[api reference]] 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, ..., γ_i) is denoted $\mathcal{P}_i$, the sum calculated is
$$\sum_i \frac{X_i}{A_i} \mathcal{P}_i \quad .$$

- [[api reference]] 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, ..., γ_i) is denoted $\mathcal{P}_i$, the sum calculated is

$$\sum_i^N w_i {\mathcal{P}_i}^2 \quad .$$

- [[api reference]] List the contents of the multifluid database in a snappy table format.

10.6 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
...
```

10.7 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 [[api reference]], incorrect usage is also indicated within commented-out statements.

Chapter 11

Physical Constants Unit

The Physical Constants unit provides a set of common constants, such as Pi and the gravitational constant, in various systems of measurement units. The default system of units is CGS, so named for having a length unit in centimeters, a mass unit in grams, and a time unit in seconds. In CGS, the charge unit is the esu, and the temperature unit is the Kelvin. The constants can also be obtained in the standard MKS system of units, where length is in meters, mass in kilograms, and time in seconds. For MKS units, charge is in Coloumbs, and temperature in Kelvin.

Flash-X Transition

For ease of usage, the constant PI=3.14159.... is defined in the header file `constants.h`. Including this file with #include "constants.h" is an alternate way to access the value of π, 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 $\mathrm{Mpc}^3\ \mathrm{Gyr}^{-2} M_\odot^{-1}$ 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, 4.4983×10^{-15} (to five significant figures).

Physical constants are taken from K. Nahamura *et al.* (Particle Data Group), J. Phys. G **37**, 075021 (2010).

11.1 Available Constants and Units

There are many constants and units available within `Flash-X`, see **??** and **??**. Should the user wish to add additional constants or units to a particular setup, the routine [[api reference]] should be overridden and the new constants added within the directory of the setup.

11.2 Applicable Runtime Parameters

There is only one runtime parameter used by the Physical Constants unit: [[rpi reference]] selects the default system of units for returned constants. It is a three-character string set to "CGS" or "MKS"; the default is CGS.

Table 11.1: 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

11.3 Routine Descriptions

The following routines are supplied by this unit.

- [[api reference]] 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.
- [[api reference]] Initializes the Physical Constants Unit by loading all constants. This routine is called by [[api reference]] and must be called before the first invocation of `PhysicalConstants_get`. In general, the user does not need to invoke this call.
- [[api reference]] Lists the available physical constants in a snappy table.
- [[api reference]] Lists all the units available for optional conversion.
- [[api reference]] Lists all physical constants and units, and tests the unit conversion routines.

Flash-X Transition

The header file `PhysicalConstants.h` must be included in the calling routine due to the optional arguments of `PhysicalConstants_get`.

11.4 Unit Test

The `PhysicalConstants` unit test [[api reference]] 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.

Table 11.2: 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

Part V

Physics Units

Chapter 12

Hydrodynamics Units

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

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{12.1}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) + \nabla P = \rho \mathbf{g} \tag{12.2}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P)\,\mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g}\ , \tag{12.3}$$

where ρ is the fluid density, $\mathbf{v}$ is the fluid velocity, P is the pressure, E is the sum of the internal energy ϵ and kinetic energy per unit mass,

$$E = \epsilon + \frac{1}{2}|\mathbf{v}|^2\ , \tag{12.4}$$

$\mathbf{g}$ is the acceleration due to gravity, and t is the time coordinate. The pressure is obtained from the energy and density using the equation of state. For the case of an ideal gas equation of state, the pressure is given by

$$P = (\gamma - 1)\rho\epsilon\ , \tag{12.5}$$

where γ is the ratio of specific heats. More general equations of state are discussed in **??** and **??**.

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

$$\epsilon = E - \frac{1}{2}|\mathbf{v}|^2 \tag{12.6}$$

can lead to unphysical values, primarily due to truncation error. This results in inaccurate pressures and temperatures. To avoid this problem, we can separately evolve the internal energy according to

$$\frac{\partial \rho\epsilon}{\partial t} + \nabla \cdot [(\rho\epsilon + P)\,\mathbf{v}] - \mathbf{v} \cdot \nabla P = 0\ . \tag{12.7}$$

If the internal energy is a small fraction of the kinetic energy (determined via the runtime parameter [[rpi reference]]), then the total energy is recomputed using the internal energy from (**??**) and the velocities from the momentum equation. Numerical experiments using the PPM solver included with Flash-X showed that using (**??**) when the internal energy falls below 10^{-4} of the kinetic energy helps avoid the truncation errors while not affecting the dynamics of the simulation.

For reactive flows, a separate advection equation must be solved for each chemical or nuclear species

$$\frac{\partial \rho X_\ell}{\partial t} + \nabla \cdot (\rho X_\ell \mathbf{v}) = 0\ , \tag{12.8}$$

where X_ℓ is the mass fraction of the ℓth species, with the constraint that $\sum_\ell X_\ell = 1$. Flash-X will enforce this constraint if you set the runtime parameter `irenorm` equal to 1. Otherwise, Flash-X will only restrict

Table 12.1: Runtime parameters used with the hydrodynamics (`Hydro`) unit.

Variable	Type	Default	Description
`eintSwitch`	real	0	If $\epsilon <$ `eintSwitch` $\cdot \frac{1}{2}\|\mathbf{v}\|^2$, use the internal energy equation to update the pressure
`irenorm`	integer	0	If equal to one, renormalize multifluid abundances following a hydro update; else restrict their values to lie between `smallx` and 1.
`cfl`	real	0.8	Courant-Friedrichs-Lewy (CFL) factor; must be less than 1 for stability in explicit schemes

Table 12.2: Solution variables used with the hydrodynamics (`Hydro`) unit.

Variable	Type	Description
`dens`	PER_VOLUME	density
`velx`	PER_MASS	x-component of velocity
`vely`	PER_MASS	y-component of velocity
`velz`	PER_MASS	z-component of velocity
`pres`	GENERIC	pressure
`ener`	PER_MASS	specific total energy $(T + U)$
`temp`	GENERIC	temperature

the abundances to fall between `smallx` and 1. The quantity ρX_ℓ represents the partial density of the ℓth 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,

$$\frac{\partial \rho\phi_\ell}{\partial t} + \nabla \cdot (\rho\phi_\ell \mathbf{v}) = 0 \ , \tag{12.9}$$

where ϕ_ℓ is the ℓth 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 **??** for more details.

12.1 Gas hydrodynamics

12.1.1 Usage

The two gas hydrodynamic solvers supplied in the release of `Flash-X` 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 **??** and **??** and implemented in the directory tree under `physics/Hydro/HydroMain/split/` and `physics/Hydro/HydroMain/unsplit/Hydro_Unsplit`.

Current and future implementations of `Hydro` use the runtime parameters and solution variables described in **??** and **??**. Additional runtime parameters used either solely by the PPM method or the unsplit hydro solver are described in [[rpi reference]].

12.1.2 The piecewise-parabolic method (PPM)

Flash-X 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-X 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-X 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-X 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 [[rpi reference]] `= .false.`.

The version of PPM in the Flash-X 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 [[rpi reference]] `= .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 [[rpi reference]] `= .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 [[rpi reference]] `= .true.`.

12.1.3 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 (USM) MHD solver (see details in **??**) that has been available in previous `Flash-X` 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 `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-X versions any more.

- [Reduced CTU vs. Full CTU for 3D in the unsplit hydro (UHD) and staggered mesh (USM) solvers]: `use_3dFullCTU` is a new switch that enhances a numerical stability for 3D simulations in the unsplit

Table 12.3: Additional runtime parameters for *Interpolation Schemes* in the unsplit hydro solver (`physics/Hydro/HydroMain/unsplit/Hydro_Unsplit`)

Variable	Type	Default	Description
`order`	integer	2	Order of method in data reconstruction: 1st order Godunov (FOG), 2nd order MUSCL-Hancock (MH), 3rd order PPM, 5th order WENO.
`transOrder`	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 `transOrder`=4 adopts a slope limiter between the 1st and 3rd order accurate methods to minimize oscillations in upwinding at discontinuities.
`slopeLimiter`	string	"vanLeer"	Slope limiter: "MINMOD", "MC", "VANLEER", "HYBRID", "LIMITED"
`LimitedSlopeBeta`	real	1.0	Slope parameter specific for the "LIMITED" slope by Toro
`charLimiting`	logical	.true.	Enable/disable limiting on characteristic variables (.false. will use limiting on primitive variables)
`use_steepening`	logical	.false.	Enable/disable contact discontinuity steepening for PPM and WENO
`use_flattening`	logical	.false.	Enable/disable flattening (or reducing) numerical oscillations for MH, PPM, and WENO
`use_avisc`	logical	.false.	Enable/disable artificial viscosity for FOG, MH, PPM, and WENO
`cvisc`	real	0.1	Artificial viscosity coefficient
`use_upwindTVD`	logical	.false.	Enable/disable upwinded TVD slope limiter PPM. NOTE: This requires NGUARD=6
`use_hybridOrder`	logical	.false.	Enable an adaptively varying reconstruction order scheme reducing its order from a high-order to first-order depending on monotonicity constraints
`use_gravHalfUpdate`	logical	.false.	On/off gravitational acceleration source terms at the half time Riemann state update
`use_3dFullCTU`	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.

Table 12.4: Additional runtime parameters for *Riemann Solvers* in the unsplit hydro solver (`physics/Hydro/HydroMain/unsplit/Hydro_Unsplit`)

Variable	Type	Default	Description
`RiemannSolver`	string	"Roe"	Different choices for Riemann solver. "LLF (local Lax-Friedrichs)", "HLL", "HLLC", "HYBRID", "ROE", and "Marquina"
`shockDetect`	logical	.false.	On/off attempting to detect strong shocks/rarefactions (and saving flag in `"shok"` variable)
`shockLowerCFL`	logical	.false.	On/off lowering of CFL factor where strong shocks are detected, automatically sets `shockDetect` if on.
`EOSforRiemann`	logical	.false.	Enable/disable calling EOS in computing each Godunov flux
`entropy`	logical	.false.	On/off entropy fix algorithm for Roe solver
`entropyFixMethod`	string	``HARTENHYMAN''	Entropy fix method for the Roe solver. "HARTEN", "HARTENHYMAN"

solvers using the corner transport upwind (CTU) algorithm by Colella. The unsplit solvers of FlashX 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 stabilty 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_3dFullCTU=.true.` (i.e., full-CTU) will make the solution evolution scheme similar to 12-CTU while requiring to solve three Riemann problems only (again, except for the CT update in USM). On the other hand, `use_3dFullCTU=.false.` (i.e., reduced-CTU) will be similar to the 6-CTU integration algorithm with a reduced CFL limit (*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 `Flash-X`. 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 **??**.

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 **??**.

12.1.3.1 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 regualar box-like grid structure in Flash-X, 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 **??**) 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 Flash-X4.3, a reduced CFL factor is automatically used in such cases; the theoretical reduced CFL limit of 1/`NDIM` is further adjusted by [[rpi reference]].

Two example simulations can be found in **??** and **??**.

12.2 Magnetohydrodynamics (MHD)

12.2.1 Description

The `Flash-X` 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 `Flash-X`. 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 `Flash-X`. 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 4π or μ_0 coefficients) conservation form, these equations are

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{12.10}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v} - \mathbf{B}\mathbf{B}) + \nabla p_* = \rho \mathbf{g} + \nabla \cdot \tau \tag{12.11}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\mathbf{v}(\rho E + p_*) - \mathbf{B}(\mathbf{v} \cdot \mathbf{B})) = \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot (\mathbf{v} \cdot \tau + \sigma \nabla T) + \nabla \cdot (\mathbf{B} \times (\eta \nabla \times \mathbf{B})) \tag{12.12}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) = -\nabla \times (\eta \nabla \times \mathbf{B}) \tag{12.13}$$

where

$$p_* = p + \frac{B^2}{2}, \tag{12.14}$$

$$E = \frac{1}{2}v^2 + \epsilon + \frac{1}{2}\frac{B^2}{\rho}, \tag{12.15}$$

$$\tau = \mu\left((\nabla\mathbf{v}) + (\nabla\mathbf{v})^{\mathrm{T}} - \frac{2}{3}(\nabla\cdot\mathbf{v})\mathbf{I}\right) \tag{12.16}$$

are total pressure, specific total energy and viscous stress respectively. Also, ρ is the density of a magnetized fluid, $\mathbf{v}$ is the fluid velocity, p is the fluid thermal pressure, T is the temperature, ϵ 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. τ is the viscosity tensor, μ is the coefficient of viscosity (dynamic viscosity), $\mathbf{I}$ is the unit (identity) tensor, σ is the heat conductivity, and η 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 **??**). In addition, both the eight-wave and the unsplit staggered mesh units declare additional plasma variables and runtime parameters, which are listed in **??** and **??**.

12.2.2 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 12.5: Additional solution variables used in the MHD units.

Variable	Type	Description
`magx`	PER_VOLUME	x-component of magnetic field
`magy`	PER_VOLUME	y-component of magnetic field
`magz`	PER_VOLUME	z-component of magnetic field
`magp`	(GENERIC)	magnetic pressure
`divb`	(GENERIC)	divergence of magnetic field

Table 12.6: Additional runtime parameters used in the MHD units.

Variable	Type	Default	Description
`UnitSystem`	string	"none"	System of units in which MHD calculations are to be performed. Acceptable values are "none" "CGS" and "SI".
`killdivb`	logical	.true.	Enable/disable divergence cleaning.
`flux_correct`	logical	.true.	Enable/disable flux correction on AMR grid.

12.2.3 Algorithm: The Unsplit Staggered Mesh Solver

A directionally unsplit staggered mesh algorithm (USM), which solves ideal and non-ideal MHD governing equations (**??**) $\sim$ (**??**) in multiple dimensions, is a new MHD solver. Since `Flash-X`, 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 ρ, plasma momentum density $\rho\mathbf{v}$ and total plasma energy ρE 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 (**??**) $\sim$ (**??**) 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 oulined scheme. The choices of TVD slope limiters available in the unsplit staggered mesh scheme (see **??**) 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 β (`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):

$$b^{n+1}_{x,i+1/2,j} = b^{n}_{x,i+1/2,j} - \frac{\Delta t}{\Delta y}\Big\{E^{n+1/2}_{z,i+1/2,j+1/2} - E^{n+1/2}_{z,i+1/2,j-1/2}\Big\}, \tag{12.17}$$

$$b^{n+1}_{y,i,j+1/2} = b^{n}_{y,i,j+1/2} - \frac{\Delta t}{\Delta x}\Big\{-E^{n+1/2}_{z,i+1/2,j+1/2} + E^{n+1/2}_{z,i-1/2,j+1/2}\Big\}. \tag{12.18}$$

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 **??**.

One of the main advantages of using the CT-type of scheme is that the cell face-centered magnetic fields $b^{n+1}_{x,i+1/2,j}$ and $b^{n+1}_{y,i,j+1/2}$, which are updated via the above induction equations, satisfy the divergence-free constraint locally. The numerical divergence of the magnetic fields is defined as

$$(\nabla \cdot \mathbf{B})^{n+1}_{i,j} = \frac{b^{n+1}_{x,i+1/2,j} - b^{n+1}_{x,i-1/2,j}}{\Delta x} + \frac{b^{n+1}_{y,i,j+1/2} - b^{n+1}_{y,i,j-1/2}}{\Delta y} \tag{12.19}$$

and it remains zero to the accuracy of machine round-off errors, provided that $nabla \cdot \mathbf{B})^{n}_{i,j} = 0$.

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 **??** below).

To solve the above induction equations (**??**) and (**??**) 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:

$$\begin{aligned} E^{n+1/2}_{z,i+1/2,j+1/2} &= \frac{1}{4}\Big\{ - F^{n+1/2}_{B_y,i+1/2,j} - F^{n+1/2}_{B_y,i+1/2,j+1} + G^{n+1/2}_{B_x,i,j+1/2} + G^{n+1/2}_{B_x,i+1,j+1/2}\Big\} \\ &= \frac{1}{4}\Big\{ E^{n+1/2}_{z,i+1/2,j} + E^{n+1/2}_{z,i+1/2,j+1} + E^{n+1/2}_{z,i,j+1/2} + E^{n+1/2}_{z,i+1,j+1/2}\Big\}, \end{aligned} \tag{12.20}$$

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 $E^{n+1/2}_{z,i+1/2,j+1/2}$ 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 β 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 **??** and **??**. Handling of different geometries will be available in future releases.

12.2.3.1 Slowly moving shock handling in PPM

A new dissipative mechanism is a hybridized slope limiter for PPM that combines a new upwind biased slope limiter with a conventional TVD slope limiter (Lee, D., Astronum Proc. 2010). This hybrid upwind limiter reduces spurious numerical oscillations near discontinuities, and therefore can compute sharp, monotonized profiles in compressible flows when using PPM, especially in Magnetohydrodynamics (MHD) slowly moving shock regions. (See more in Chapter **??**.)

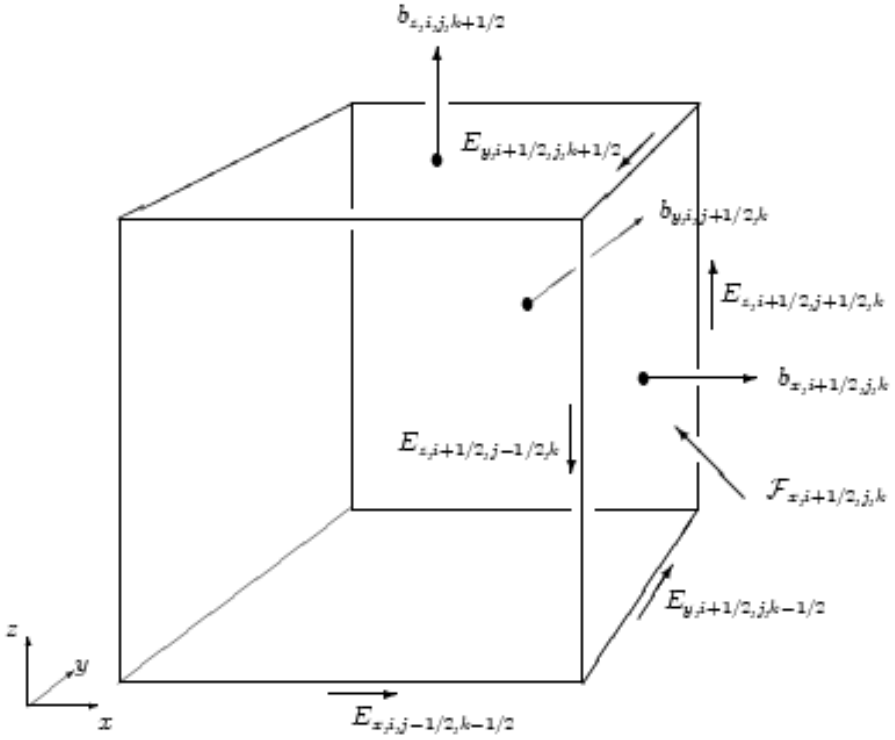

Figure 12.1: A 3D control volume on the staggered grid with the cell center at (i, j, k). The magnetic fields are collocated at the cell face centers and the electric fields at the cell edge centers. The line integral of the electric fields $\int \_\partial\mathcal{F}\_n\mathbf{E} \cdot \mathbf{T}dl$ 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 **??**.

Table 12.7: 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
`killdivb`	logical	.true.	On/off $\nabla \cdot \mathbf{B} = 0$ handling on the staggered grid
`E_modification`	logical	.true.	Enable/disable high-order electric field construction
`E_upwind`	logical	.false.	Enable/disable an upwind update for induction equations
`energyFix`	logical	.true.	Enable/disable energy correction
`facevar2ndOrder`	logical	.true.	Turn on/off a second-order facevar update
`ForceHydroLimit`	logical	.false.	On/off pure Hydro mode
`prolMethod`	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
`RiemannSolver`	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.*, 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/`.

Chapter 13

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 con 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.

Chapter 14

Equation of State Unit

14.1 Introduction

The `Eos` unit implements the equation of state needed by the hydrodynamics and nuclear burning solvers. The function [[api reference]] 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 ([[api reference]]) is provided, which takes a section of a block and translates it into the data format required by the [[api reference]] function, then calls the function. Upon return from the [[api reference]] 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 `Flash-X`: `Gamma` which implements a perfect-gas equation of state; `Gamma/RHD`which implements a perfect-gas equation taking relativistic effects into account; `Multigamma`which implements a perfect-gas equation of state with multiple fluids, each of which can have its own adiabatic index (γ); and `Helmholtz`which uses a fast Helmholtz free-energy table interpolation to handle degenerate/relativistic electrons/positrons and includes radiation pressure and ions (via the perfect gas approximation).

As described in previous sections, Flash-X 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-X is capable of using any general equation of state, we discuss here the three equation of state routines that are supplied: an ideal-gas or gamma-law EOS, an EOS for a fluid composed of multiple gamma-law gases, and a tabular Helmholtz free energy EOS appropriate for stellar interiors. The two gamma-law EOSs consist of simple analytic expressions that make for a very fast EOS routine both in the case of a single gas or for a mixture of gases. The Helmholtz EOS includes much more physics and relies on a table look-up scheme for performance.

14.2 Gamma Law and Multigamma

Flash-X 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 indicesThe first is necessary to calculate the adiabatic sound speed and is given by

$$\gamma_1 = \frac{\rho}{P}\frac{\partial P}{\partial \rho} \ . \tag{14.1}$$

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

$$\gamma_4 = 1 + \frac{P}{\rho\epsilon} \ . \tag{14.2}$$

These two adiabatic indices are stored as the mesh-based variables `GAMC_VAR` and `GAME_VAR`. All EOS routines must return γ_1, and γ_4 is calculated from (**??**).

The gamma-lawEOS models a simple ideal gas with a constant adiabatic index γ. Here we have dropped the subscript on γ, because for an ideal gas, all adiabatic indices are equal. The relationship between pressure P, density ρ, and specific internal energy ϵ is

$$P = (\gamma - 1)\,\rho\epsilon \ . \tag{14.3}$$

We also have an expression relating pressure to the temperature T

$$P = \frac{N_a k}{\bar{A}}\rho T \ , \tag{14.4}$$

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

$$\frac{1}{\bar{A}} = \sum_i \frac{X_i}{A_i} \ , \tag{14.5}$$

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

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

The relativistic variant of the ideal gas equation is explained in more detail in **??**.

Simulations are not restricted to a single ideal gas; the multigamma EOSsimulations with several species of ideal gases each with its own value of γ. In this case the above expressions hold, but γ represents the weighted average adiabatic index calculated from

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

We note that the analytic expressions apply to both the forward (internal energy as a function of density, temperature, and composition) and backward (temperature as a function of density, internal energy and composition) relations. Because the backward relation requires no iteration in order to obtain the temperature, this EOS is quite inexpensive to evaluate. Despite its fast performance, use of the gamma-law EOS is limited, due to its restricted range of applicability for astrophysical problems.

14.3 Helmholtz

The Helmholtz EOSprovided with the Flash-X 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

$$P_{\rm tot} = P_{\rm rad} + P_{\rm ion} + P_{\rm ele} + P_{\rm pos} + P_{\rm coul} \tag{14.8}$$

$$\epsilon_{\rm tot} = \epsilon_{\rm rad} + \epsilon_{\rm ion} + \epsilon_{\rm ele} + \epsilon_{\rm pos} + \epsilon_{\rm coul} \ . \tag{14.9}$$

Here the subscripts “rad,” “ion,” “ele,” “pos,” and “coul” represent the contributions from radiation, nuclei, electrons, positrons, and corrections for Coulomb effects, respectively. The radiation portion assumes a

blackbody in local thermodynamic equilibrium, the ion portion (nuclei) is treated as an ideal gas with $\gamma = 5/3$, and the electrons and positrons are treated as a non-interacting Fermi gas.

The blackbody pressure and energy are calculated as

$$P_{\rm rad} = \frac{aT^4}{3} \tag{14.10}$$

$$\epsilon_{\rm rad} = \frac{3P_{\rm rad}}{\rho} \tag{14.11}$$

where a is related to the Stephan-Boltzmann constant $\sigma_B = ac/4$, and c is the speed of light. The ion portion of each routine is the ideal gas of (Equations **??** – **??**) with $\gamma = 5/3$. The number densities of free electrons $N_{\rm ele}$ and positrons $N_{\rm pos}$ in the noninteracting Fermi gas formalism are given by

$$N_{\rm ele} = \frac{8\pi\sqrt{2}}{h^3}\ m_{\rm e}^3\ c^3\ \beta^{3/2}\ \left[F_{1/2}(\eta,\beta)\ +\ F_{3/2}(\eta,\beta)\right] \tag{14.12}$$

$$N_{\rm pos} = \frac{8\pi\sqrt{2}}{h^3}\ m_{\rm e}^3\ c^3\ \beta^{3/2}\left[F_{1/2}\left(-\eta-2/\beta,\beta\right)\ +\ \beta\ F_{3/2}\left(-\eta-2/\beta,\beta\right)\right]\ , \tag{14.13}$$

where h is Planck's constant, $m_{\rm e}$ is the electron rest mass, $\beta = kT/(m_{\rm e}c^2)$ is the relativity parameter, $\eta = \mu/kT$ is the normalized chemical potential energy μ for electrons, and $F_k(\eta,\beta)$ is the Fermi-Dirac integral

$$F_k(\eta,\beta) = \int_0^\infty \frac{x^k\ (1+0.5\ \beta\ x)^{1/2}\ dx}{\exp(x-\eta)+1}\ . \tag{14.14}$$

Because the electron rest mass is not included in the chemical potential, the positron chemical potential must have the form $\eta_{\rm pos} = -\eta - 2/\beta$. For complete ionization, the number density of free electrons in the matter is

$$N_{\rm ele,matter} = \frac{\bar{Z}}{\bar{A}}\ N_a\ \rho = \bar{Z}\ N_{\rm ion}\ , \tag{14.15}$$

and charge neutrality requires

$$N_{\rm ele,matter} = N_{\rm ele} - N_{\rm pos}\ . \tag{14.16}$$

Solving this equation with a standard one-dimensional root-finding algorithm determines η. Once η 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 γ_1 and γ_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-X provides a table of the Helmholtz free energy (hence the name) and makes use of a thermodynamically consistent interpolation scheme obviating the need to perform the complex calculations required of the above formalism during the course of a simulation. The interpolation scheme uses a bi-quintic Hermite interpolant resulting in an accurate EOS that performs reasonably well.

The Helmholtz free energy,

$$F = \epsilon - T\ S \tag{14.17}$$

$$dF = -S\ dT + \frac{P}{\rho^2}\ d\rho\ , \tag{14.18}$$

is the appropriate thermodynamic potential for use when the temperature and density are the natural thermodynamic variables. The free energy table distributed with Flash-X was produced from the Timmes EOS (Timmes & Arnett 1999). The Timmes EOS evaluates the Fermi-Dirac integrals (**??**) and their partial derivatives with respect to η and β to machine precision with the efficient quadrature schemes of Aparicio (1998) and uses a Newton-Raphson iteration to obtain the chemical potential of (**??**). 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.* $F(\rho, \mathrm{T})$. It is made with $\bar{\mathrm{A}} = \bar{\mathrm{Z}} = 1$ (pure hydrogen), with an electron fraction $Y_\mathrm{e} = 1$. 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 **??** – **??**), 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 $Y_\mathrm{e} = 1$ 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^{-3}) and $10^4 < T < 10^{11}$ (K). As with the gamma-law EOS, the Helmholtz EOS provides both forward and backward relations. In the case of the forward relation (ρ, T, given along with the composition) the table lookup scheme and analytic formulae directly provide relevant thermodynamic quantities. In the case of the backward relation (ρ, ϵ, and composition given), the routine performs a Newton-Rhaphson iteration to determine temperature. It is possible for the input variables to be changed in the iterative modes since the solution is not exact. The returned quantities are thermodynamically consistent.

14.4 Usage

14.4.1 Initialization

The initialization function of the Eos unit [[api reference]] 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 [[api reference]] routine is a little more complex. The `Helmholtz` EOS requires an input file `helm_table.dat` that contains the lookup table for the electron contributions. This table is currently stored in ASCII for portability purposes. When the table is first read in, a binary version called `helm_table.bdat` is created. This binary format can be used for faster subsequent restarts on the same machine but may not be portable across platforms. The `Eos_init` routine reads in the table data on processor 0 and broadcasts it to all other processors.

14.4.2 Runtime Parameters

Runtime parameters for the `Gamma` unit require the user to set the thermodynamic properties for the single gas. [[rpi reference]], [[rpi reference]], [[rpi reference]] 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 [[api reference]]`.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 [[rpi reference]] and [[rpi reference]] which define the maximum number of iterations and convergence tolerance. An additional runtime parameter for `Helmholtz`, [[rpi reference]], indicates whether or not to apply Coulomb corrections. In some regions of the ρ-T plane, the approximations made in the Coulomb corrections may be invalid and result in negative pressures. When the parameter `eos_coulombMult` is set to zero, the Coulomb corrections are not applied.

14.4.3 Direct and Wrapped Calls

The primary function in the `Eos` unitoperates on a vector, taking density, composition, and either temperature, internal energy, or pressure as input, and returning γ_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 [[api reference]] 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 [[api reference]].

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, [[api reference]] 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 [[api reference]] interface, and upon return translates the vector format back to the block section. This wrapper routine cannot calculate the optional derivative quantities. If they are needed, call the `Eos` routine directly with the optional mask set to true and space allocated for the returned quantities.

14.5 Unit Test

The unit test of the Eos function can exercise all three implementations. Because the Gamma law allows only one species, the setup required for the three implementations is specific. To invoke any three-dimensional `Eos` unit test, the command is:

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

where *implementation* is one of `Gamma`, `Multigamma`, `Helmholtz`. The `Eos` unit test works on the assumption that if the four physical variables in question (density, pressure, energy and temperature) are in thermal equilibrium with one another, then applying the equation of state to any two of them should leave the other two completely unchanged. Hence, if we initialize density and temperature with some arbitrary values, and apply the equation of state to them in `MODE_DENS_TEMP`, then we should get pressure and energy values that

are thermodynamically consistent with density and temperature. Now after saving the original temperature value, we apply the equation of state to density and newly calculated pressure. The new value of the temperature should be identical to the saved original value. This verifies that the `Eos` unit is computing correctly in `MODE_DENS_PRES` mode. By repeating this process for the remaining two modes, we can say with great confidence that the `Eos` unit is functioning normally.

In our implementation of the Eos unit test, the initial conditions applied to the domain create a gradient for density along the x axis and gradients for temperature and pressure along the y axis. If the test is being run for the Multigamma or Helmholtz implementations, then the species are initialized to have gradients along the z axis.

Chapter 15

Local Source Terms

The `physics/sourceTerms` organizational directory contains several units that implement forcing terms. The `Burn`, `Stir`, `Ionize`, and `Diffuse` units contain implementations in `Flash-X`. Two other units, `Cool` and `Heat`, contain only stub level routines in their API.

15.1 Burn Unit

The nuclear burning implementation of the `Burn` unit uses a sparse-matrix semi-implicit ordinary differential equation (ODE) solver to calculate the nuclear burning rate and to update the fluid variables accordingly (Timmes 1999). The primary interface routines for this unit are [[api reference]], which sets up the nuclear isotope tables needed by the unit, and [[api reference]], which calls the ODE solver and updates the hydrodynamical variables in a single row of a single block. The routine [[api reference]] may limit the computational timestep because of burning considerations. There is also a helper routine `Simulation/SimulationComposition/Simulation_initSpecies` (see [[api reference]]) which provides the properties of ions included in the burning network.

15.1.1 Algorithms

Modeling thermonuclear flashes typically requires the energy generation rate due to nuclear burning over a large range of temperatures, densities and compositions. The average energy generated or lost over a period of time is found by integrating a system of ordinary differential equations (the nuclear reaction network) for the abundances of important nuclei and the total energy release. In some contexts, such as supernova models, the abundances themselves are also of interest. In either case, the coefficients that appear in the equations are typically extremely sensitive to temperature. The resulting stiffness of the system of equations requires the use of an implicit time integration scheme.

A user can choose between two implicit integration methods and two linear algebra packages in Flash-X. The runtime parameter [[rpi reference]] controls which integration method is used in the simulation. The choice `odeStepper = 1` is the default and invokes a Bader-Deuflhard scheme. The choice `odeStepper = 2` invokes a Kaps-Rentrop or Rosenbrock scheme. The runtime parameter [[rpi reference]] 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-X. 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-X code.

15.1.2 Reaction networks

We begin by describing the equations solved by the nuclear burning unit. We consider material that may be described by a density ρ 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 ρ_i denote the number and mass density, respectively, of the ith isotope, and let X_i denote its mass fraction, so that

$$X_i = \rho_i/\rho = n_i A_i/(\rho N_A) \ , \tag{15.1}$$

where N_A is Avogadro's number. Let the molar abundance of the ith isotope be

$$Y_i = X_i/A_i = n_i/(\rho N_A) \ . \tag{15.2}$$

Mass conservation is then expressed by

$$\sum_{i=1}^{N} X_i = 1 \ . \tag{15.3}$$

At the end of each timestep, Flash-X checks that the stored abundances satisfy (**??**) 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 ith isotope is given in Lagrangian formulation by

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

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

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

where λ_{kj} and λ_{jk} are the reverse (creation) and forward (destruction) nuclear reaction rates, respectively. Contributions from three-body reactions, such as the triple-α reaction, are easy to append to (**??**). The mass diffusion velocities $\mathbf{V}_i$ in (**??**) 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-X distribution. Setting $\mathbf{V}_i \equiv 0$ transforms (**??**) into

$$\frac{dY_i}{dt} = \dot{R}_i \ , \tag{15.6}$$

which may be written in the more compact, standard form

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

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

Because of the highly nonlinear temperature dependence of the nuclear reaction rates and because the abundances themselves often range over several orders of magnitude in value, the values of the coefficients which appear in (**??**) and (**??**) 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 (**??**) and the associated Jacobian matrix are formed. Consider the ^{12}C(α,γ)^{16}O reaction, which competes with the triple-α 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 (**??**) through the terms

$$\begin{aligned} \dot{Y}(^4He) &= -Y(^4He)\ Y(^{12}C)\ R + \ldots \\ \dot{Y}(^{12}C) &= -Y(^4He)\ Y(^{12}C)\ R\ + \ldots \\ \dot{Y}(^{16}O) &= +Y(^4He)\ Y(^{12}C)\ R\ + \ldots, \end{aligned} \tag{15.8}$$

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

$$\begin{aligned} J(^4He,^4He) &= -Y(^{12}C)\ R\ + \ldots & J(^4He,^{12}C) &= -Y(^4He)\ R\ + \ldots \\ J(^{12}C,^4He) &= -Y(^{12}C)\ R\ + \ldots & J(^{12}C,^{12}C) &= -Y(^4He)\ R\ + \ldots \\ J(^{16}O,^4He) &= +Y(^{12}C)\ R\ + \ldots & J(^{16}O,^{12}C) &= +Y(^4He)\ R\ + \ldots. \end{aligned} \tag{15.9}$$

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

This release of `Flash-X` 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 α-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 α-chain and heavy-ion reactions as the 13-isotope network, but it includes additional isotopes to accommodate some types of hydrogen burning (PP chains and steady-state CNO cycles), along with some aspects of photo-disintegration into ^{54}Fe. This 19 isotope reaction network is described in Weaver, Zimmerman, & Woosley (1978).

The networks supplied with Flash-X are examples of a "hard-wired" reaction network, where each of the reaction sequences are carefully entered by hand. This approach is suitable for small networks, when minimizing the CPU time required to run the reaction network is a primary concern, although it suffers the disadvantage of inflexibility.

15.1.2.1 Two linear algebra packages: MA28 and GIFT

As mentioned in the previous section, the Jacobian matrices of nuclear reaction networks tend to be sparse, and they become more sparse as the number of isotopes increases. Since implicit or semi-implicit time integration schemes generally require solving systems of linear equations involving the Jacobian matrix, taking advantage of the sparsity can significantly reduce the CPU time required to solve the systems of linear equations.

The MA28 sparse matrix package used by Flash-X 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 $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ into a symbolic LU decomposition, a numerical LU decomposition, and a backsubstitution phase. In the symbolic LU decomposition phase, the pivot order of a matrix is determined, and a sequence of decomposition operations that minimizes the amount of fill-in is recorded. Fill-in refers to zero matrix elements which become nonzero (*e.g.*, a sparse matrix times a sparse matrix is generally a denser matrix). The matrix is not decomposed; only the steps to do so are stored. Since the nonzero pattern of a chosen nuclear reaction network does not change, the symbolic LU decomposition is a one-time initialization cost for reaction networks. In the numerical LU decomposition phase, a matrix with the same pivot order and nonzero pattern as a previously factorized matrix is numerically decomposed into its lower-upper form. This phase must be done only once for each set of linear equations. In the backsubstitution phase, a set of linear equations is solved with the factors calculated from a previous numerical decomposition. The backsubstitution phase may be performed with as many right-hand sides as needed, and not all of the right-hand sides need to be known in advance.

MA28 uses a combination of nested dissection and frontal envelope decomposition to minimize fill-in during the factorization stage. An approximate degree update algorithm that is much faster (asymptotically and in practice) than computing the exact degrees is employed. One continuous real parameter sets the amount of searching done to locate the pivot element. When this parameter is set to zero, no searching is done and the diagonal element is the pivot, while when set to unity, partial pivoting is done. Since the matrices generated by reaction networks are usually diagonally dominant, the routine is set in Flash-X 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 $\tilde{\mathbf{A}}$ is reduced to upper triangular form, and backsubstitution with the right-hand side $\mathbf{b}$ yields the solution to $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$. GIFT generated routines skip all calculations with matrix elements that are zero; in this restricted sense, GIFT generated routines are sparse, but the storage of a full matrix is still required. It is assumed that the pivot element is located on the diagonal and no row or column interchanges are performed, so GIFT generated routines may become unstable if the matrices are not diagonally dominant. These routines must decompose the matrix for each right-hand side in a set of linear equations. GIFT writes out (in Fortran code) the sequence of Gaussian elimination and backsubstitution steps without any do loop constructions on the matrix $A(i, j)$. As a result, the routines generated by GIFT can be quite large. For the 489 isotope network discussed by Timmes (1999), GIFT generated $\sim 5.0{\times}10^7$ lines of code! Fortunately, for small reaction networks (less than about 30 isotopes), GIFT generated routines are much smaller and generally faster than other linear algebra packages.

The Flash-X runtime parameter [[rpi reference]] 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.

15.1.2.2 Two time integration methods

One of the time integration methods used by Flash-X 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

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \sum_{i=1}^{4} b_i \Delta_i \ , \tag{15.10}$$

where the four vectors Δ^i are found from successively solving the four matrix equations

$$
\begin{aligned}
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_1 &= \mathbf{f}(\mathbf{y}^n) && (15.11)\\
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_2 &= \mathbf{f}(\mathbf{y}^n + a_{21}\Delta_1) + c_{21}\Delta_1/h && (15.12)\\
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_3 &= \mathbf{f}(\mathbf{y}^n + a_{31}\Delta_1 + a_{32}\Delta_2) + (c_{31}\Delta_1 + c_{32}\Delta_2)/h && (15.13)\\
(\tilde{\mathbf{1}}^\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_4 &= \mathbf{f}(\mathbf{y}^n + a_{31}\Delta_1 + a_{32}\Delta_2) + (c_{41}\Delta_1 + c_{42}\Delta_2 + c_{43}\Delta_3)/h\ . && (15.14)
\end{aligned}
$$

b_i, γ, 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 (Δ_4 depends on $\Delta_3 \ldots$ depends on Δ_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-X makes use of the routine GRK4T given by Kaps & Rentrop (1979).

Another time integration method used by Flash-X 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,

$$
\begin{aligned}
h &= H/m\\
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \Delta_0 &= h\mathbf{f}(\mathbf{y}^n) && (15.15)\\
\mathbf{y}_1 &= \mathbf{y}^n + \Delta_0\ .
\end{aligned}
$$

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

$$
\begin{aligned}
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \mathbf{x} &= h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}\\
\Delta_k &= \Delta_{k-1} + 2\mathbf{x} && (15.16)\\
\mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k\ ,
\end{aligned}
$$

and closure is obtained by the last stage

$$
\begin{aligned}
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \Delta_m &= h[\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}]\\
\mathbf{y}^{n+1} &= \mathbf{y}_m + \Delta_m\ . && (15.17)
\end{aligned}
$$

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-X) and the smoothness of the solution. Estimates of the accuracy of an integration step are made by comparing the solutions derived from different orders. The minimum cost of this method — which applies for a single timestep that met or exceeded the specified integration accuracy — is one Jacobian evaluation, eight evaluations of the right-hand side, two matrix decompositions, and ten backsubstitutions. This minimum cost can be increased at a rate of one decomposition (the expensive part) and m backsubstitutions (the inexpensive part) for every increase in the order $2k+1$. The cost of increasing the order is compensated for, hopefully, by being able to take correspondingly larger (but accurate) timestep. The controls for order versus step size are a built-in part of the Bader-Deuflhard method. The cost per step of this integration method is at least twice as large as the cost per step of either a traditional first-order accurate Euler method or the fourth-order accurate Kaps-Rentrop discussed above. However, if the Bader-Deuflhard method can take accurate timesteps that are at least twice as large, then this method will be more efficient globally. Timmes (1999) shows that this is typically (but not always!) the case. Note that in Equations **??** – **??**, 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-X runtime parameter [[rpi reference]] controls which integration method is used in the simulation. The choice `odeStepper = 1` is the default and invokes the variable order Bader-Deuflhard scheme. The choice `odeStepper = 2` invokes the fourth order Kaps-Rentrop / Rosenbrock scheme.

15.1.3 Detecting shocks

For most astrophysical detonations, the shock structure is so thin that there is insufficient time for burning to take place within the shock. However, since numerical shock structures tend to be much wider than their physical counterparts, it is possible for a significant amount of burning to occur within the shock. Allowing this to happen can lead to unphysical results. The burner unit includes a multidimensional shock detection algorithm that can be used to prevent burning in shocks. If the [[rpi reference]] parameter is set to `.false.`, this algorithm is used to detect shocks in the Burn unit and to switch off the burning in shocked cells.

Currently, the shock detection algorithm supports Cartesian and 2-dimensional cylindrical coordinates. The basic algorithm is to compare the jump in pressure in the direction of compression (determined by looking at the velocity field) with a shock parameter (typically 1/3). If the total velocity divergence is negative and the relative pressure jump across the compression front is larger than the shock parameter, then a cell is considered to be within a shock.

This computation is done on a block by block basis. It is important that the velocity and pressure variables have up-to-date guard cells, so a guard cell call is done for the burners only if we are detecting shocks (*i.e.* `useShockBurning = .false.`).

15.1.4 Energy generation rates and reaction rates

The instantaneous energy generation rate is given by the sum

$$\dot{\epsilon}_{\rm nuc} = N_A \sum_i \frac{dY_i}{dt} \ . \tag{15.18}$$

Note that a nuclear reaction network does not need to be evolved in order to obtain the instantaneous energy generation rate, since only the right hand sides of the ordinary differential equations need to be evaluated. It is more appropriate in the Flash-X program to use the average nuclear energy generated over a timestep

$$\dot{\epsilon}_{\rm nuc} = N_A \sum_i \frac{\Delta Y_i}{\Delta t} \ . \tag{15.19}$$

In this case, the nuclear reaction network does need to be evolved. The energy generation rate, after subtraction of any neutrino losses, is returned to the Flash-X program for use with the operator splitting technique.

The tabulation of Caughlan & Fowler (1988) is used in Flash-X 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-X. The runtime parameter [[rpi reference]] 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-X 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 $\dot{\epsilon}_{\rm nuc}$ due to neutrino losses as given by Itoh *et al.* (1996) are included in Flash-X.

15.1.5 Temperature-based timestep limiting

When using explicit hydrodynamics methods, a timestep limiter must be used to ensure the stability of the numerical solution. The standard CFL limiter is always used when an explicit hydrodynamics unit is included in Flash-X. 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 [[rpi reference]] controls this ratio. The timestep limiter is calculated as

$$\Delta t_{burn} = \texttt{enucDtFactor} \cdot \frac{E_{int}}{E_{nuc}} \tag{15.20}$$

where E_{nuc} is the nuclear energy, expressed as energy per volume per time, and E_{int} is the internal energy per volume. For good coupling between the hydrodynamics and burning, `enucDtFactor` should be < 1. The default value is kept artificially high so that in most simulations the time limiting due to burning is turned off. Care must be exercised in the use of this routine.

15.2 Ionization Unit

The analysis of UV and X-ray observations, and in particular of spectral lines, is a powerful diagnostic tool of the physical conditions in astrophysical plasmas (*e.g.*, the outer layers of the solar atmosphere, supernova remnants, *etc.*). Since deviation from equilibrium ionization may have a non-negligible effect on the UV and X-ray lines, it is crucial to take into account these effects in the modeling of plasmas and in the interpretation of the relevant observations.

In light of the above observations, Flash-X 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

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

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

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

$$\frac{\partial n_i^Z}{\partial t} + \nabla \cdot n_i^Z \mathbf{v} = R_i^Z \quad (i = 0, \ldots, Z)\ , \tag{15.24}$$

where ρ 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

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

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-X species, will be given by

$$N_{\rm spec} = 2 + \sum_Z (Z + 1);$$

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.

Table 15.1: Runtime parameters used with the `Ionize` unit.

Variable	Type	Default	Description
[[rpi reference]]	real	1.0×10^4	Min nei temperature
[[rpi reference]]	real	1.0×10^7	Max nei temperature
[[rpi reference]]	real	1.0	Min nei electron number density
[[rpi reference]]	real	1.0×10^{12}	Max nei electron number density

15.2.1 Algorithms

A fractional step method is required to integrate the equations and in particular to decouple the NEI solver from the hydro solver. For each timestep, the homogeneous hydrodynamic transport equations given by (**??**) are solved using the Flash-X 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

$$\frac{\partial n_i^Z}{\partial t} = R_i^Z \ (i = 0, \ldots, Z) \tag{15.26}$$

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

Note that in the present version, the contribution of the ionization and recombination to the energy equation (the bracketed term in (**??**)) is not accounted for. Also, it should be noted that the source term in the NEI unit implementation is adequate to solve the problem for optically thin plasma in the "coronal" approximation; just collisional ionization, auto-ionization, radiative recombination, and dielectronic recombination are considered.

15.2.2 Usage

In order to run a Flash-X 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 **??**. There are two implementations of `physics/sourceTerms/Ionize/IonizeMain`: the default implementation, `Nei` (tested using `Neitest` (see **??**)), and `Eqi` (untested in `Flash-X`). The former computes ion species for non-equilibrium ionization, and the latter computes ion species in the approximation of ionization equilibrium.

The `Ionize` unit requires that the subunit implementation `Simulation/SimulationComposition/Ionize` be used to set up the ion species of the fluid. The ions are defined in a file `Simulation/SimulationComposition/Ionize/SpeciesList.txt`, however, the `Config` file in the simulation directory (e.g. `Simulation/SimulationMain/Neitest/Config`) defines which subset of these elements are to be used.

15.3 Stir Unit

The addition of driving terms in a hydrodynamical simulation can be a useful feature, for example, in generating turbulent flows or for simulating the addition of power on larger scales (*e.g.*, supernova feedback into the interstellar medium). The `Stir` unit 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-X 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.

15.3.1 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 [[rpi reference]]) divided by the decay time τ ([[rpi reference]]). In the limit that the timestep $\Delta t \to 0$, it is easily seen that the algorithm represents a linearly-weighted summation of the old state with the new Gaussian random variable.

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

The velocities are then converted to physical space by a direct Fourier transform – *i.e.*, adding the sin and cos terms explicitly. Since most drivings involve a fairly small number of modes, this is more efficient than an FFT, since the FFT would need large numbers of modes (equal to six times the number of cells in the domain), the vast majority of which would have zero amplitude.

15.3.2 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):

$$\mathrm{d}\widehat{\vec{f}}(\vec{k},t) = f_0\,(\vec{k})\;\underline{\mathcal{P}}^{\zeta}(\vec{k})\;\mathrm{d}\vec{\mathcal{W}}(t) \;-\; \widehat{\vec{f}}\,(\vec{k},t)\;\frac{\mathrm{d}t}{T}\;. \tag{15.27}$$

The first term on the right hand side is a diffusion term. This term is modeled by a Wiener process $\vec{\mathcal{W}}(t)$, which adds a Gaussian random increment to the vector field given in the previous time step $\mathrm{d}t$. Wiener processes are random processes, such that

$$\vec{\mathcal{W}}(t) - \vec{\mathcal{W}}(t-\mathrm{d}t) = \vec{\mathcal{N}}(0,\mathrm{d}t)\;, \tag{15.28}$$

where $\vec{\mathcal{N}}(0, \mathrm{d}t)$ denotes the 3D, 2D, or 1D version of a Gaussian distribution with zero mean and standard deviation $\mathrm{d}t$. 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

$$\mathcal{P}_{ij}^{\zeta}\left(\vec{k}\right) = \zeta\,\mathcal{P}_{ij}^{\perp}\left(\vec{k}\right) + (1-\zeta)\,\mathcal{P}_{ij}^{\parallel}\left(\vec{k}\right) = \zeta\,\delta_{ij} + (1-2\zeta)\,\frac{k_i k_j}{|k|^2}\;, \tag{15.29}$$

where δ_{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 ζ, 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 (**??**) by evaluating the norm of the compressive component of the projection tensor,

$$\left|(1-\zeta)\,\mathcal{P}_{ij}^{\parallel}\right|^2 = (1-\zeta)^2\;, \tag{15.30}$$

and by evaluating the norm of the full projection tensor

$$\left|\mathcal{P}_{ij}^{\zeta}\right|^2 = 1 - 2\zeta + D\zeta^2\;. \tag{15.31}$$

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 (**??**) and (**??**) gives the relative power in compressive modes, $F_{\rm long}/F_{\rm tot}$, as a function of ζ:

$$\frac{F_{\rm long}}{F_{\rm tot}} = \frac{(1-\zeta)^2}{1 - 2\zeta + D\zeta^2}\;. \tag{15.32}$$

Figure **??** 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 ζ 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 $F_{\rm long}/F_{\rm tot} = 1/3$ for 3D turbulence, and $F_{\rm long}/F_{\rm tot} = 1/2$ 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 (**??**) 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, $T = L_{\rm peak}/V$, on the scale of energy injection, $L_{\rm peak}$. 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 **??** section.

15.3.3 Using the StirFromFile Unit

15.3.3.1 Runtime Parameters

Table **??** 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 ([[rpi reference]] is switched off by default, because it is normally sufficient to restrict the timestep based on the gas velocity). Finally, [[rpi reference]] 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

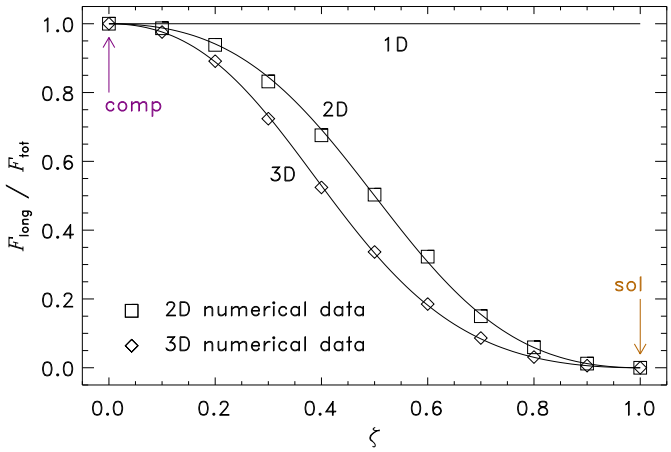

Figure 15.1: Ratio of compressive to total power of the turbulent stirring field, reprinted from Federrath et al. (2010, A&A, 512, A81) with permission by Astronomy & Astrophysics. The solid lines labelled with 1D, 2D, and 3D show the analytical expectation for this ratio, equation (**??**), as a function of the stirring parameter ζ 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 $F\_long/F\_tot = 1$, independent of ζ.

Table 15.2: Runtime parameters for the stirring module.

Variable	Type	Default	Description
[[rpi reference]]	boolean	.true.	switch stirring on/off
[[rpi reference]]	boolean	.false.	restrict timestep based on stirring
[[rpi reference]]	string	"forcingfile.dat"	file containing the stirring time sequence

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-X 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 $\vec{f}$.

15.3.3.2 Preparing the Stirring Sequence (`st_infilename`)

The code requires a time sequence of stirring modes at runtime, which have to be prepared with the standalone Fortran program `forcing_generator.F90` in `SimulationMain/StirFromFile/forcing_generator/`. A Makefile is provided in the same directory. This program prepares the time sequence of Fourier modes, which is then read by Flash-X 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 15.3: Parameters in `forcing_generator.F90` to prepare a stirring sequence.

Variable	Type	Default	Description
`ndim`	integer	3	The dimensionality of the simulation (1, 2, or 3)
`xmin, xmax`	real	-0.5, 0.5	Domain boundary coordinates in x direction
`ymin, ymax`	real	-0.5, 0.5	Domain boundary coordinates in y direction
`zmin, zmax`	real	-0.5, 0.5	Domain boundary coordinates in z direction
`st_spectform`	integer	1	Spectral shape (0: band, 1: paraboloid)
`st_decay`	real	0.5	Autocorrelation time of the OU process, $T = L_{\rm peak}/V$
`st_energy`	real	2e-3	Determines the driving amplitude
`st_stirmin`	real	6.283	Minimum wavenumber stirred (e.g., $k_{\rm min} \lesssim 2\pi/L_{\rm box}$)
`st_stirmax`	real	18.95	Maximum wavenumber stirred (e.g., $k_{\rm max} \gtrsim 6\pi/L_{\rm box}$)
`st_solweight`	real	1.0	Mode mixture $\zeta = [0, 1]$ in Eq. (**??**). Typical values are 1.0: solenoidal; 0.0: compressive; 0.5: natural mixture.
`st_seed`	integer	140281	Random seed for stirring sequence
`end_time`	real	5.0	Final time in stirring sequence
`nsteps`	integer	100	Number of realizations between $t = 0$ and `end_time`
`outfilename`	string	"forcingfile.dat"	Output name (input file `st_infilename` for Flash-X)

Table **??** 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, ...`[1]), 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 $k_{\rm min}$ = `st_stirmin` and $k_{\rm max}$ = `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 $k_{\rm peak} = (k_{\rm min} + k_{\rm max})/2$ and falls off quadratically towards both wavenumber ends, normalized such that the injected power at $k_{\rm min}$ and $k_{\rm max}$ vanishes. This has the advantage of defining a characteristic peak injection scale $k_{\rm peak}$ 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 $L_{\rm peak} = 2\pi/k_{\rm peak}$, the autocorrelation time should be set equal to the turbulent crossing time, $T = L_{\rm peak}/V$. 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 (**??**). 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 ζ is presented in Federrath et al. (2010, A&A, 512, A81).

`st_seed` is the random seed for the OU sequence and determines the pseudo random number sequence for the integrated Box-Muller random number generator.

Finally, `end_time` and `nsteps` determine the final physical time for stirring and the number of driving patterns to be prepared within the time period from $t = 0$ to $t =$ `end_time`. This sets the number of equally-spaced times at which Flash-X 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` $= 10 \times$ `end_time`/`st_decay`. This will sample the smooth changes in the OU driving sequence sufficiently well for most applications.

[1] Note that we typically assume a cubic box with side length $L\_{\rm box}$ = `xmax` − `xmin` = `ymax` − `ymin` = `zmax` − `zmin`

15.3.4 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-X 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 **??**). 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 used as basic code checks.

Chapter 16

Gravity Unit

16.1 Introduction

The `Gravity` unit supplied with `Flash-X` 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 $\mathbf{g}(\mathbf{x})$,

$$\mathbf{g}(\mathbf{x}) = -\nabla\phi(\mathbf{x}) \ . \tag{16.1}$$

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

$$\nabla^2\phi(\mathbf{x}) = 4\pi G\rho(\mathbf{x}) \ , \tag{16.2}$$

where G is Newton's gravitational constant. In the latter case, either periodic or isolated boundary conditions can be applied.

16.2 Externally Applied Fields

The Flash-X 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.

16.2.1 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`.

16.2.2 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.

16.2.3 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.

16.2.4 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-X_GRAVITY_TIMEDEP` in your setup's Config file.

16.3 Self-gravity

The self-consistent gravity algorithm supplied with Flash-X computes the Newtonian gravitational field produced by the matter. The produced potential function satisfies Poisson's equation (**??**). This unit's implementation can also return the acceleration field $\mathbf{g}(\mathbf{x})$ computed by finite-differencing the potential using the expressions

$$\begin{aligned} g_{x;ijk} &= \tfrac{1}{2\Delta x}\left(\phi_{i-1,j,k} - \phi_{i+1,j,k}\right) + \mathcal{O}(\Delta x^2) \\ g_{y;ijk} &= \tfrac{1}{2\Delta y}\left(\phi_{i,j-1,k} - \phi_{i,j+1,k}\right) + \mathcal{O}(\Delta y^2) \\ g_{z;ijk} &= \tfrac{1}{2\Delta z}\left(\phi_{i,j,k-1} - \phi_{i,j,k+1}\right) + \mathcal{O}(\Delta z^2)\ . \end{aligned} \tag{16.3}$$

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

The tree solver described in **??** is appropriate for general mass distributions and can solve problems with both isolated and periodic boundary conditions set independently in individual directions.

16.3.1 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

$$\rho E \equiv \frac{1}{2}\rho v^2 + \rho\epsilon\ , \tag{16.4}$$

where ϵ 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 ϕ 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-X uses the following update steps to obtain the momentum and energy in cell i at timestep $n+1$

$$\begin{aligned}(\rho v)_i^{n+1} &= (\rho v)_i^n + \tfrac{\Delta t}{2} g_i^{n+1}\left(\rho_i^n + \rho_i^{n+1}\right) \\ (\rho E)_i^{n+1} &= (\rho E)_i^n + \tfrac{\Delta t}{4} g_i^{n+1}\left(\rho_i^n + \rho_i^{n+1}\right)\left(v_i^n + v_i^{n+1}\right) \ .\end{aligned} \tag{16.5}$$

Here g_i^{n+1} is obtained by extrapolation from ϕ_i^{n-1} and ϕ_i^n. The `Poisson` gravity implementation supplies a mesh variable to contain the potential from the previous timestep; future releases of Flash-X 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

$$\begin{aligned}v_{L,i+1/2} &\rightarrow v_{L,i+1/2} + \frac{\Delta t}{4}\left(g^+_{L,i+1/2} + g^-_{L,i+1/2}\right) \\ v_{R,i+1/2} &\rightarrow v_{R,i+1/2} + \frac{\Delta t}{4}\left(g^+_{R,i+1/2} + g^-_{R,i+1/2}\right) \ .\end{aligned} \tag{16.6}$$

Here $g^{\pm}_{X,i+1/2}$ 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.

16.3.2 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 [[api reference]] 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

$$D^{m+2} > \frac{GMS^m_{\rm node}}{\Delta a_{\rm p,APE}} \tag{16.7}$$

where D is distance between the *point-of-calculation* and the node, M is the node mass, $S_{\rm node}$ is the node size, m is a degree of the multipole approximation and $\Delta a_{\rm p,APE}$ is the requested maximum error in acceleration (controlled by runtime parameter [[rpi reference]]). 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 $\Delta a_{\rm p,MPE}$ according to formula 9 from Salmon&Warren (1994; see this paper for details):

$$\Delta a_{\rm p,MPE} \le \frac{1}{D^2}\frac{1}{(1 - S_{\rm node}/D)^2}\left(\frac{3\lceil B_2\rceil}{D^2} - \frac{2\lfloor B_3\rfloor}{D^3}\right) \tag{16.8}$$

where $B_n = \Sigma_i m_i |\mathbf{r}_i - \mathbf{r}_0|^n$ 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 [[rpi reference]] and [[rpi reference]] (see **??**).

During the tree walk, subroutine [[api reference]] adds contributions of tree nodes to the gravitational potential or acceleration fields. In case of the potential, the contribution is

$$\Phi = -\frac{GM}{|\vec{r}|} \tag{16.9}$$

if `isolated` boundary conditions are used, or

$$\Phi = -GM f_{\mathrm{EF},\Phi}(\vec{r}) \tag{16.10}$$

if periodic `periodic` or `mixed` boundary conditions are used. In case of the acceleration, the contributions are

$$\vec{a}_g = \frac{GM\vec{r}}{|\vec{r}|^3} \tag{16.11}$$

for `isolated` boundary conditions, or

$$\vec{a}_g = GM f_{\mathrm{EF,a}}(\vec{r}) \tag{16.12}$$

for `periodic` or `mixed` boundary conditions. In 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 $f_{\mathrm{EF},\Phi}$ and $f_{\mathrm{EF,a}}$ 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, $Gm/r = Gm\,\mathrm{erf}(\alpha r)/r + Gm\,\mathrm{erfc}(\alpha r)/r$ (α is an arbitrary constant) and then by applying Poisson summation formula on erfc terms, gravitational field at position $\vec{r}$ can be written in the form

$$\phi(\vec{r}) = -G\sum_{a=1}^{N} m_a \left(\sum_{i_1,i_2,i_3} A_S(\vec{r},\vec{r}_a,\vec{l}_{i_1,i_2,i_3}) + A_L(\vec{r},\vec{r}_a,\vec{l}_{i_1,i_2,i_3}) \right) = -G\sum_{a=1}^{N} m_a f_{\mathrm{EF},\Phi}(\vec{r}_a - \vec{r}) \ , \tag{16.13}$$

the first sum runs over whole computational domain, where at position $\vec{r}_a$ is mass m_a. Second sum runs over all neigbouring 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. **??**. The Ewald field for the acceleration, $f_{\mathrm{EF,a}}$, 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 if mapped to a grid using a linear interpolation each time the gravitational acceleration is calculated (in subroutine [[api reference]]).

16.4 Usage

To include the effects of gravity in your Flash-X 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 **??** explicitly compute the potential, hence they inherit the null implementation from the API for accessing potential. The gravitation acceleration can be obtained either on the whole domain, a single block or a single row at a time.

When building an application that solves the Possion equation for the gravitational potential, additional storage is needed in `unk` for holding the last, as well as (usually) the previous, gravitational potential field; and, depending on the Poisson solver used, additional variables may be needed. The variables `GPOT_VAR` and `GPOT_VAR`, and others as needed, will be automatically defined in `Flash.h` in those cases. See [[api reference]] for more information.

16.4.1 Tree Gravity Unit Usage

Calculation of gravitational potential can be enabled by compiling in this unit and setting the runtime parameter [[rpi reference]] true. The constant of gravity can be set independently by runtime parameter [[rpi reference]]; if it is not positive, the constant `Newton` from the Flash-X `PhysicalConstants` database is used. If parameters [[rpi reference]] or [[rpi reference]] are set, the gravity unit MAC is used and it can be chosen by setting [[rpi reference]] to either `ApproxPartialErr` or `MaxPartialErr`. If the first one is used, the order of the multipole approximation is given by [[rpi reference]].

The maximum allowed partial error in gravitational acceleration is set with the runtime parameter [[rpi reference]]. 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 [[rpi reference]] 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 [[rpi reference]] and they can be `isolated`, `periodic` or `mixed`. In the case of mixed boundary conditions, runtime parameters [[rpi reference]], [[rpi reference]] and [[rpi reference]] 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 [[rpi reference]] is `periodic` or `mixed`, then the Ewald field for appropriate symmetry is calculated at the beginning of the simulation. Parameter [[rpi reference]] controls the range of indices i_1, i_2, i_3 in (eq. **??**). 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 [[rpi reference]] 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 [[rpi reference]] 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 [[rpi reference]]. If [[rpi reference]] is too low to cover origin (where is the Ewald field discontinuous), then the run is terminated. Each grid is composed of [[rpi reference]] $\times$ [[rpi reference]] $\times$ [[rpi reference]] 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 [[rpi reference]] (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 [[rpi reference]] true. The parameter [[rpi reference]] gives the name of the file with the external potential and [[rpi reference]] 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 [[rpi reference]], [[rpi reference]] and [[rpi reference]] specify the position (in the simulation coordinate system) of the external field origin (the point where the radial or z-coordinate is zero).

Table 16.1: Tree gravity unit parameters controlling the accuracy of calculation.

Variable	Type	Default	Description
[[rpi reference]]	real	-1.0	constant of gravity; if < 0, it is obtained from internal physical constants database
[[rpi reference]]	string	"ApproxPartialErr"	MAC, other option: "MaxPartialErr"
[[rpi reference]]	integer	2	degree of multipole in error estimate in APE MAC
[[rpi reference]]	logical	.false.	if .true., grv_bhAccErr has meaning of relative error, otherwise absolute
[[rpi reference]]	real	0.1	maximum allowed error in gravitational acceleration

Table 16.2: Tree gravity unit parameters controlling the boundary conditions.

Variable	Type	Default	Description
[[rpi reference]]	string	"isolated"	or "periodic" or "mixed"
[[rpi reference]]	string	"isolated"	or "periodic"
[[rpi reference]]	string	"isolated"	or "periodic"
[[rpi reference]]	string	"isolated"	or "periodic"
[[rpi reference]]	boolean	true	whether Ewald field should be regenerated
[[rpi reference]]	integer	10	number of terms in the Ewald series
[[rpi reference]]	integer	32	number of points+1 of the Taylor expansion
[[rpi reference]]	string	"ewald_coeffs"	file with coefficients of the Ewald field Taylor expansion
[[rpi reference]]	integer	32	size of the Ewald field grid in x-direction
[[rpi reference]]	integer	32	size of the Ewald field grid in y-direction
[[rpi reference]]	integer	32	size of the Ewald field grid in z-direction
[[rpi reference]]	integer	-1	number of refinement levels (nested grids) for the Ewald field; if < 0, determined automatically
[[rpi reference]]	logical	.true.	if .false., semi-quadratic interpolation is used for interpolation in the Ewald field
[[rpi reference]]	string	"ewald_field_acc"	file with the Ewald field for acceleration
[[rpi reference]]	string	"ewald_field_pot"	file with coefficients of the Ewald field for potential

Tree gravity unit parameters controlling the external gravitational field.

Variable	Type	Default	Description
[[rpi reference]]	logical	.false.	whether to use external field
[[rpi reference]]	logical	.true.	whether to use gravitational field calculated by the tree solver
[[rpi reference]]	string	"external_potential.dat"	file containing the external gravitational field
[[rpi reference]]	string	"planez"	type of the external field: planar or spherical symmetry
[[rpi reference]]	real	0.0	x-coordinate of the center of the external field
[[rpi reference]]	real	0.0	y-coordinate of the center of the external field
[[rpi reference]]	real	0.0	z-coordinate of the center of the external field

16.5 Unit Tests

There are two unit tests for the gravity unit. `Poisson3` is essentially the Maclaurin spheroid problem described in **??**. 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 [[api reference]]. 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 **??**. `PoisParticles` is based on the Huang-Greengard Poisson gravity test described in **??**.

Chapter 17

Particles Unit

The support for particles in `Flash-X` 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 ith passive particle is particularly simple and requires only the time integration of interpolated mesh velocities.

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i \tag{17.1}$$

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

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n \ . \tag{17.2}$$

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 ith active particle include the equation (**??**) and another describing the effects of forces.

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{F}_{\mathrm{lr},i} + \mathbf{F}_{\mathrm{sr},i} \ , \tag{17.3}$$

Here, $\mathbf{F}_{\mathrm{lr},i}$ represents the sum of all long-range forces (coupling all particles, except possibly those handled by the short-range term) acting on the ith particle and $\mathbf{F}_{\mathrm{sr},i}$ 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 (**??**) forward through time. Many alternative integration methods are described in Section **??** below. Additional information about the mesh to particle mapping is described in **??**. An introduction to the particle techniques used in Flash-X is given by R. W. Hockney and J. W. Eastwood in *Computer Simulation using Particles* (Taylor and Francis, 1988).

Flash-X Transition

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

New since Flash-X

Since release 3.1 of Flash-X, a single simulation can have both active and passive particles defined. Flash-X and Flash-X 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 **??** for command line options, **??** for Config sytax, and **??** below for more details.

Flash-X 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 **??** below for information specific to sink particles.

17.1 Time Integration

The active and passive particles have many different time integration schemes available. The subroutine [[api reference]] handles the movement of particles through space and time. Because Flash-X has support for including different types of both active and passive particles in a single simulation, the implementation of [[api reference]] may call several helper routines of the form pt_advance*METHOD* (*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_advance*METHOD* subroutine call.

17.1.1 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 **??** are provided by this subunit. A specific implementation of the active portion of [[api reference]] 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 is **??** 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:

$$\begin{aligned} \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \\ \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n \ . \end{aligned} \tag{17.4}$$

  Here $\mathbf{a}_i$ is the particle acceleration. Within Flash-X, 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

$$\begin{aligned} \mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\ \mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \tfrac{1}{2}\mathbf{a}_i^0 \Delta t^0 \\ \mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2} + C_n \mathbf{a}_i^n + D_n \mathbf{a}_i^{n-1} \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \Delta t^n \ . \end{aligned} \tag{17.5}$$

The coefficients C_n and D_n are given by

$$\begin{aligned} C_n &= \tfrac{1}{2}\Delta t^n + \tfrac{1}{3}\Delta t^{n-1} + \tfrac{1}{6}\left(\tfrac{\Delta t^{n\,2}}{\Delta t^{n-1}}\right) \\ D_n &= \tfrac{1}{6}\left(\Delta t^{n-1} - \tfrac{\Delta t^{n\,2}}{\Delta t^{n-1}}\right) . \end{aligned} \tag{17.6}$$

By using time-centered velocities and stored accelerations, this method achieves second-order time accuracy even with variable timesteps. Within `Flash-X`, 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

$$\begin{aligned} \mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\ \mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \frac{1}{2}\mathbf{a}_i^0 \Delta t^0 \\ \mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2}\left[1 - \frac{A^n}{2}\Delta t^n + \frac{1}{3!}\Delta t^{n\,2}\left(A^{n\,2} - \dot{A}^n\right)\right]\left[1 - \Delta t^{n-1}\frac{A^n}{2} + \Delta t^{n-1\,2}\frac{A^{n\,2} + 2\dot{A}^n}{12}\right] \\ &\quad + \mathbf{a}_i^n\left[\frac{\Delta t^{n-1}}{2} + \frac{\Delta t^{n\,2}}{6\Delta t^{n-1}} + \frac{\Delta t^{n-1}}{3} - \frac{A^n \Delta t^n}{6}\left(\Delta t^n + \Delta t^{n-1}\right)\right] \\ &\quad + \mathbf{a}_i^{n-1}\left[\frac{\Delta t^{n-1\,2} - \Delta t^{n\,2}}{6\Delta t^{n-1}} - \frac{A^n \Delta t^{n-1}}{12}(\Delta t^n + \Delta t^{n-1})\right] \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2}\Delta t^n . \end{aligned}$$

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 **??**.

The leapfrog-based integrators implemented under `Particles/ParticlesMain/active/massive` supply the additional particle attributes listed in **??**.

Table 17.1: 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 17.2: 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

17.1.2 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 mean 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).

17.1.2.1 The hybrid equations

In the hybrid approximation, ions are treated as particles, and electrons as a massless fluid. In what follows we use SI units. We have N_I ions at positions $\mathbf{r}_i(t)$ [m] with velocities $\mathbf{v}_i(t)$ [m/s], mass m_i [kg] and charge q_i [C], $i = 1, \ldots, N_I$. By spatial averaging we can define the charge density $\rho_I(\mathbf{r}, t)$ [Cm^{-3}] of the ions, their average velocity $\mathbf{u}_I(\mathbf{r}, t)$ [m/s], and the corresponding current density $\mathbf{J}_I(\mathbf{r}, t) = \rho_I \mathbf{u}_I$ [$\mathrm{Cm}^{-2}\mathrm{s}^{-1}$]. 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,

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i, \quad \frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i}\left(\mathbf{E} + \mathbf{v}_i \times \mathbf{B}\right), \quad i = 1, \ldots, N_I$$

where $\mathbf{E} = \mathbf{E}(\mathbf{r}, t)$ is the electric field, and $\mathbf{B} = \mathbf{B}(\mathbf{r}, t)$ is the magnetic field. The electric field is given by

$$\mathbf{E} = \frac{1}{\rho_I}\left(-\mathbf{J}_I \times \mathbf{B} + \mu_0^{-1}\left(\nabla \times \mathbf{B}\right) \times \mathbf{B} - \nabla p_e\right) + \eta \mathbf{J} - \eta_h \nabla^2 \mathbf{J},$$

where ρ_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 η_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,

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}.$$

17.1.2.2 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-X implementation of the hybrid solver with periodic boundary conditions.

17.1.2.3 Hybrid solver implementation

The two basic operations needed for a PIC code are provided as standard operations in Flash-X:

- 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 [[rpi reference]]. 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 `cden`. The magnetic field is represented by (`grbx`, `grby`, `grbz`). A background field can be set during initialization in (`gbx1`, `gby1`, `gbz1`). We then solve for the deviation from this background field. Care must be take so that the background field is divergence free, using the discrete divergence operator. The easiest way to ensure this is to compute the background field as the rotation of a potential. The electric field is stored in (`grex`, `grey`, `grez`), the current density in (`grjx`, `grjy`, `grjz`) , and the ion current density in (`gjix`, `gjiy`, `gjiz`). A resistivity is stored in `gres`, thus it is possible to have a non-uniform resistivity in the simulation domain, but the default is to set the resistivity to the constant value of `sim_resistivity` everywhere. For post processing, separate fields are stored for charge density and ion current density for 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-X particles they also each have a position $\mathbf{r}_i$=(`posx`, `posy`, `posz`) and a velocity $\mathbf{v}_i$=(`velx`, `vely`, `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 $\mathbf{B}(\mathbf{r}_i)$, respectivly.

Regarding the choice of time step. The timestep must be constant, Δt =`dtinit = dtmin = dtmax`, since the leap frog solver requires this. For the solution to be stable the time step, Δt, must be small enough. We will try and quantify this, and here we asume that the grid cells are cubes, $\Delta x = \Delta y = \Delta z$, and that we have a single species plasma.

Table 17.3: 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
`pt_picPname_1`	STRING	"H+"	Species 1 name
`pt_picPmass_1`	REAL	1.0	Species 1 mass, m_i [amu]
`pt_picPcharge_1`	REAL	1.0	Species 1 charge, q_i [e]
`pt_picPdensity_1`	REAL	1.0	Initial n_I species 1 [m^{-3}]
`pt_picPtemp_1`	REAL	1.5e5	Initial T_I species 1 [K]
`pt_picPvelx_1`	REAL	0.0	Initial $\mathbf{u}_I$ species 1 [m/s]
`pt_picPvely_1`	REAL	0.0	
`pt_picPvelz_1`	REAL	0.0	
`pt_picPpc_1`	REAL	1.0	Number of macro-particle of species 1 per cell
`sim_bx`	REAL	0.0	Initial $\mathbf{B}$ (at $t = 0$) [T]
`sim_by`	REAL	0.0	
`sim_bz`	REAL	0.0	
`sim_te`	REAL	0.0	Initial T_e [K]
`pt_picResistivity`	REAL	0.0	Resistivity, η [Ω m]
`pt_picResistivityHyper`	REAL	0.0	Hyperresistivity, η_h
`sim_gam`	REAL	-1.0	Adiabatic exponent for electrons
`sim_nsub`	INTEGER	3	Number of CL B-field update subcycles (must be odd)
`sim_rng_seed`	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-X (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),

$$\Delta t < \frac{\Omega_i^{-1}}{\pi}\left(\frac{\Delta x}{\delta_i}\right)^2 \sim \frac{n}{B}\left(\Delta x\right)^2, \qquad \delta_i = \frac{1}{|q_i|}\sqrt{\frac{m_i}{\mu_0\, n}},$$

where δ_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, Δt 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.

17.1.3 Passive Particles

Passive particles may be moved using one of several different methods available in `Flash-X`. 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 **??**.

Numerically solving Equation (**??**) 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 17.4: 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
`cden`	PER_VOLUME	Total charge density, ρ [C/m^3]
`grbx`	GENERIC	Magnetic field, $\mathbf{B}$ [T]
`grby`	GENERIC	
`grbz`	GENERIC	
`grex`	GENERIC	Electric field, $\mathbf{E}$ [V/m]
`grey`	GENERIC	
`grez`	GENERIC	
`grjx`	PER_VOLUME	Current density, $\mathbf{J}$ [A/m^2]
`grjy`	PER_VOLUME	
`grjz`	PER_VOLUME	
`gjix`	PER_VOLUME	Ion current density, $\mathbf{J}_I$ [A/m^2]
`gjiy`	PER_VOLUME	
`gjiz`	PER_VOLUME	
`cde1`	PER_VOLUME	Species 1 charge density, ρ_I [C/m^3]
`jix1`	PER_VOLUME	Species 1 ion current density, $\mathbf{J}_I$ [A/m^2]
`jiy1`	PER_VOLUME	
`jiz1`	PER_VOLUME	

cannot be arbitrarily chosen by a particle motion ODE solver[1]. Statements about the order of a method in this context should be understood as referring to the same method if it were applied in a hypothetical simulation where evaluations of velocities $\mathbf{v}_i$ could be performed at arbitrary times (and with unlimited accuracy). Note that `Flash-X` 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 [[api reference]] is called, the fluid fields have already been updated to t^{n+1}, as is the case with the [[api reference]] implementations provided with `Flash-X`. A specific implementation of the passive portion of [[api reference]] 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 is **??** 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:

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \quad . \tag{17.7}$$

  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.

[1] Even though it is possible to do so, see [[api reference]], one does not in general wish to let particles integration dictate the time step of the simulation.

Table 17.5: Important particle properties for the hybrid solver. Note that this is for the computational macro-particles.

Variable	Type	Description
`specie`	REAL	Particle type (an integer number 1,2,3,...)
`mass`	REAL	Mass of the particle, m_i [kg]
`charge`	REAL	Charge of the particle, q_i [C]
`jx`	REAL	Particle ion current, $\mathbf{J}_{Ii} = q_i \mathbf{v}_i$ [A m]
`jy`	REAL	
`jz`	REAL	
`bx`	REAL	Magnetic field at particle, $\mathbf{B}(\mathbf{r}_i)$ [T]
`by`	REAL	
`bz`	REAL	
`ex`	REAL	Electric field at particle, $\mathbf{E}(\mathbf{r}_i)$ [V/m]
`ey`	REAL	
`ez`	REAL	

- **Two-Stage Runge-Kutta (`Particles/ParticlesMain/passive/RungeKutta`).** This 2-stage Runge-Kutta scheme is the preferred choice in `Flash-X`. 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:

$$\begin{aligned} \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \frac{\Delta t^n}{2}\left[\mathbf{v}_i^n + \mathbf{v}_i^{*,n+1}\right] , \\ \text{where} \quad \mathbf{v}_i^{*,n+1} &= \mathbf{v}(\mathbf{x}_i^{*,n+1}, t^{n+1}) , \\ \mathbf{x}_i^{*,n+1} &= \mathbf{x}_i^n + \Delta t^n \mathbf{v}_i^n \quad . \end{aligned} \tag{17.8}$$

  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 [2]; 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

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^{n-1} + \mathbf{v}_i^n(\Delta t^{n-1} + \Delta t^n) \quad . \tag{17.9}$$

  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).

  Particle advancement follows the equation

[2] They can be considered "predicted" positions and velocities.

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t^n \, \mathbf{v}_i^{\text{comb}} \ , \tag{17.10}$$

where

$$\mathbf{v}_i^{\text{comb}} = c_1 \, \mathbf{v}(\mathbf{x}_i^{*,n+\frac{1}{2}}, t^n) + c_2 \, \mathbf{v}(\mathbf{x}_i^{*,n+\frac{1}{2}}, t^{n+1}) + c_3 \, \mathbf{v}(\mathbf{x}_i^n, t^n) + c_4 \, \mathbf{v}(\mathbf{x}_i^n, t^{n+1}) \tag{17.11}$$

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 mipoint 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

$$\mathbf{v}(\mathbf{x}_i^{\text{E},n+\frac{1}{2}}, t^{n+\frac{1}{2}}) = \mathbf{v}(\mathbf{x}_i^n, t^n) + \mathbf{v}_t(\mathbf{x}_i^n, t^n)\frac{1}{2}\Delta t^n + (\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}})\mathbf{v}(\mathbf{x}_i^n, t^n)\frac{1}{2}\Delta t^n + O((\frac{1}{2}\Delta t^n)^2) \tag{17.12}$$

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

On the other hand, expansion of (**??**) gives

$$\begin{aligned} \mathbf{v}_i^{\text{comb}} &= (c_1 + c_2 + c_3 + c_4)\mathbf{v}(\mathbf{x}_i^n, t^n) \\ &\quad + (c_2 + c_4)\mathbf{v}_t(\mathbf{x}_i^n, t^n)\Delta t \ + \ (c_1 + c_2)(\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}})\mathbf{v}(\mathbf{x}_i^n, t^n)\Delta t_*^n \\ &\quad + \ \text{ higher order terms in } \Delta t \text{ and } \Delta t_*^n. \end{aligned}$$

After introducing a time step factor f defined by $\Delta t_*^n = f \, \Delta t^n$, this becomes

$$\begin{aligned} \mathbf{v}_i^{\text{comb}} &= (c_1 + c_2 + c_3 + c_4)\mathbf{v}(\mathbf{x}_i^n, t^n) \\ &\quad + (c_2 + c_4)\mathbf{v}_t(\mathbf{x}_i^n, t^n)\Delta t \ + \ (c_1 + c_2)(\mathbf{v}_i^n \cdot \frac{\partial}{\partial \mathbf{x}})\mathbf{v}(\mathbf{x}_i^n, t^n) \, f \, \Delta t \\ &\quad + O((\Delta t)^2) \quad . \end{aligned} \tag{17.13}$$

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

$$\mathbf{v}_i^{\text{comb}} = \mathbf{v}(\mathbf{x}_i^{\text{E},n+\frac{1}{2}}, t^{n+\frac{1}{2}}) + O((\Delta t)^2) \quad . \tag{17.14}$$

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:

$$\begin{aligned} c_1 + c_2 + c_3 + c_4 &= 1 \quad \text{(otherwise the scheme will not even be of first order)} , \\ c_2 + c_4 &= \frac{1}{2} \quad \text{(and thus also } c_1 + c_3 = \tfrac{1}{2}) , \\ c_1 + c_2 &= \frac{\Delta t^n}{\Delta t^{n-1}} \quad . \end{aligned}$$

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:

$$\begin{aligned} c_1 &= \frac{\Delta t^n}{\Delta t^{n-1}} - \frac{1}{2} , \\ c_2 &= \frac{1}{2} , \\ c_3 &= 1 - \frac{\Delta t^n}{\Delta t^{n-1}} , \\ c_4 &= 0 \quad . \end{aligned}$$

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 (**??**) simplifies significantly.

An Euler step, as described for `passive/Euler` in (**??**), 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:

$$\Delta t^n > \Delta t^{n-1} \quad \text{and} \quad \left|\Delta t^n - \Delta t^{n-1}\right| \geq \texttt{pt\_dtChangeToleranceUp} \times \Delta t^{n-1}$$

or

$$\Delta t^n < \Delta t^{n-1} \quad \text{and} \quad \left|\Delta t^n - \Delta t^{n-1}\right| \geq \texttt{pt\_dtChangeToleranceDown} \times \Delta t^{n-1},$$

where [[rpi reference]] and [[rpi reference]] 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 **??**.

17.2 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 `Flash-X`, 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

$$f_i(x) \equiv \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i-1/2}} f(x')\,dx' \tag{17.15}$$

where i is the cell index and Δx 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 **??** for more details.

17.2.1 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

$$f(x) = A + B(x - x_i) + C(x - x_i)^2 \ . \tag{17.16}$$

Then integrating gives

$$\begin{aligned} f_{i-1} &= \frac{1}{\Delta x}\left[A + \frac{1}{2}B(x-x_i)^2\Big|_{x_{i-3/2}}^{x_{i-1/2}} + \frac{1}{3}C(x-x_i)^3\Big|_{x_{i-3/2}}^{x_{i-1/2}}\right] \\ &= A - B\Delta x + \frac{13}{12}C\Delta x^2 \ , \end{aligned} \tag{17.17}$$

$$\begin{aligned} f_{i} &= \frac{1}{\Delta x}\left[A + \frac{1}{2}B(x-x_i)^2\Big|_{x_{i-1/2}}^{x_{i+1/2}} + \frac{1}{3}C(x-x_i)^3\Big|_{x_{i-1/2}}^{x_{i+1/2}}\right] \\ &= A + \frac{1}{12}C\Delta x^2 \ , \end{aligned} \tag{17.18}$$

and

$$\begin{aligned} f_{i+1} &= \frac{1}{\Delta x}\left[A + \frac{1}{2}B(x-x_i)^2\Big|_{x_{i+1/2}}^{x_{i+3/2}} . + \frac{1}{3}C(x-x_i)^3\Big|_{x_{i-3/2}}^{x_{i-1/2}}\right] \\ &= A - B\Delta x + \frac{13}{12}C\Delta x^2 \ , \end{aligned} \tag{17.19}$$

We may write these as

$$\begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & -1 & \frac{13}{12} \\ 1 & 0 & \frac{1}{12} \\ 1 & 1 & \frac{13}{12} \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \end{bmatrix} . \tag{17.20}$$

Inverting this gives expressions for A, B, and C,

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & \frac{13}{12} & -\frac{1}{24} \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i+1} \\ f_i \\ f_{i-1} \end{bmatrix} . \tag{17.21}$$

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

$$f(x,y) = A + B(x - x_i) + C(x - x_i)^2 + D(y - y_j) + E(y - y_j)^2 + F(x - x_i)(y - y_j) \ . \tag{17.22}$$

In this case, the cell averaged quantities are given by

$$f_{i,j}(x,y) \equiv \frac{1}{\Delta y}\Delta x \int_{x_{i-1/2}}^{x_{i+1/2}} dx' \int_{y_{j-1/2}}^{x_{j-1/2}} dy' f(x',y') \ . \tag{17.23}$$

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

$$\begin{bmatrix} f_{i-1,j-1} \\ f_{i,j-1} \\ f_{i+1,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i-1,j+1} \\ f_{i,j+1} \\ f_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} 1 & -1 & \frac{13}{12} & -1 & \frac{13}{12} & 1 \\ 1 & 0 & \frac{1}{12} & -1 & \frac{13}{12} & 0 \\ 1 & 1 & \frac{13}{12} & -1 & \frac{13}{12} & -1 \\ 1 & -1 & \frac{13}{12} & 0 & \frac{1}{12} & 0 \\ 1 & 0 & \frac{1}{12} & 0 & \frac{1}{12} & 0 \\ 1 & 1 & \frac{13}{12} & 0 & \frac{1}{12} & 0 \\ 1 & -1 & \frac{13}{12} & 1 & \frac{13}{12} & -1 \\ 1 & 0 & \frac{1}{12} & 1 & \frac{13}{12} & 0 \\ 1 & 1 & \frac{13}{12} & 1 & \frac{13}{12} & 1 \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \\ F\Delta x\Delta y \end{bmatrix} . \tag{17.24}$$

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

$$\begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{1}{12} & -1 & \frac{13}{12} \\ 1 & -1 & \frac{13}{12} & 0 & \frac{1}{12} \\ 1 & 0 & \frac{1}{12} & 0 & \frac{1}{12} \\ 1 & 1 & \frac{13}{12} & 0 & \frac{1}{12} \\ 1 & 0 & \frac{1}{12} & 1 & \frac{13}{12} \end{bmatrix} \begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \end{bmatrix} . \tag{17.25}$$

Inverting gives

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & -\frac{1}{24} & \frac{7}{6} & -\frac{1}{24} & -\frac{1}{24} \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix} . \tag{17.26}$$

Similarly, in three dimensions, the interpolation function is

$$f(x,y,z) = A + B(x-x_i) + C(x-x_i)^2 + D(y-y_j) + E(y-y_j)^2 + F(z-z_k) + G(z-z_k)^2 \ . \tag{17.27}$$

and we have

$$\begin{bmatrix} A \\ B\Delta x \\ C\Delta x^2 \\ D\Delta y \\ E\Delta y^2 \\ F\Delta z \\ G\Delta z^2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} & \frac{5}{4} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{24} \\ 0 & 0 & -\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -1 & 0 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & -1 & 0 & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{i,j,k-1} \\ f_{i,j-1,k} \\ f_{-i,j,k} \\ f_{i,j,k} \\ f_{i+1,j,k} \\ f_{i,j+1,k} \\ f_{i,j,k+1} \end{bmatrix} . \tag{17.28}$$

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

$$\begin{aligned} f(r,z) = \ & \\ & A + B(r-r_i) + C(r-r_i)^2 + D(z-z_j) \\ & + E(z-z_j)^2 + F(r-r_i)(z-z_j) \ . \end{aligned} \tag{17.29}$$

and

$$\begin{bmatrix} A \\ B\Delta r \\ C\Delta r^{\frac{2}{6}} \\ D\Delta z \\ E\Delta z^2 \end{bmatrix} =$$

$$\begin{bmatrix} -\frac{1}{24} & -\frac{h_1-1}{24h_1} & \frac{7}{6} & -\frac{h_1-1}{24h1} & -\frac{1}{24} \\ 0 & -\frac{(7+6h_1)(h_1-1)}{3h_2} & \frac{2h_1}{3h2} & \frac{(7+6h_1)(h_1-1)}{3h_2} & 0 \\ 0 & \frac{\left(12h_1^2+12h_1-1\right)(h_1-1)}{h_1h_2} & -2\frac{12h_1^2-13}{h_2} & -\frac{\left(12h_1^2+12h_1-1\right)(h_1-1)}{h_1h_2} & 0 \\ -\frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -1 & \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} f_{i,j-1} \\ f_{i-1,j} \\ f_{i,j} \\ f_{i+1,j} \\ f_{i,j+1} \end{bmatrix} . \quad (17.30)$$

17.2.2 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 `Flash-X` 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.

17.3 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 `Flash-X`. Particle types must be specified in the `Config` file of the Simulations unit setup directory for the application, and the syntax is explained in **??**. At configuration time, the setup script parses the `PARTICLETYPE` specifications in the Config files, and generates an F90 file [[api reference]]`.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 [[api reference]]. The users of active particles typically have their own custom initialization. The following two implementations of initialization techniques are included in the `Flash-X` distribution (they are more likely to used with the passive tracer particles):

`Lattice` distributes particles regularly along the axes directions throughout a subsection of the physical grid.

`WithDensity` distributes particles randomly, with particle density being proportional to the grid gas density.

Users have two options for implementing custom initialization methods. The two files involved in the process are: [[api reference]] 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 **??**.

- The `ParticlesMapping` subunit controls the mapping of particle properties to and from the grid. Flash-X 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 **??** 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 `Flash-X`. 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 [[api reference]] 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 [[api reference]]. 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 **??**. 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 (**??**) 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-X. 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-X. However, note that the sink particle implementation described below in **??** 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 **??** 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 **??** below as well as **??** 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 `Flash-X`, 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-X will abort if the criterion is too demanding. To overcome the abort, specify a less demanding criterion, or increase the value of `lrefine_max`.

17.3.1 Particles Runtime Parameters

There are several general runtime parameters applicable to the `Particles` unit, which affect every implementation. The variable [[rpi reference]] obviously must be set equal to `.true.` to utilize the Particles unit. The time stepping is controlled with [[rpi reference]]; 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 [[rpi reference]] and similar variables in Y and Z. The physical range of initialization is controlled by [[rpi reference]] 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 **??** for more details.

17.3.2 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 **??**). 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-X 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 [[api reference]] is invoked by Flash-X 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
[[rpi reference]], [[rpi reference]], etc. These specified attributes are collected in an array by the [[api reference]] routine. This array in turn is used by [[api reference]] to calculate the values of the specified attributes from the corresponding mesh quantities before they are output.

17.3.3 Particle I/O

Particle data are written to and read from checkpoint files by the I/O modules (**??**). For more information on the format of particle data written to output files, see **??** and **??**.

Particle data can also be written out to the `flash.dat` file. The user should include a local copy of [[api reference]] 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 [[api reference]] which can be used to dump particle output to a plain text file. An example of usage can be found in [[api reference]]. Output from this routine can be read using the fidlr routine `particles_dump.pro`.

17.3.4 Unit Tests

The unit tests provided for `Particles` exercise the [[api reference]] methods for tracer particles. Tests under `Simulation/SimulationMain/unitTest/ParticlesAdvance` can be used to examine and compare convergence behavior of various time integration schemes. The tests compare numerical and analytic solutions for a problem (with a given velocity field) where analytic solutions can be computed.

Currently only one `ParticlesAdvance` test is provided. It is designed to be easily modified by replacing a few source files that contain implementations of the equation and the analytic solution. The use the test, configure it with a command like

```
./setup -auto -1d unitTest/ParticlesAdvance/HomologousPassive \
                  -unit=Particles/ParticlesMain/passive/EstiMidpoint2
```

and replace `EstiMidpoint2` with one of the other available methods (or omit the option to get the default method), see **??**. 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 [[rpi reference]] INTEGER 2` — The order of the integration scheme. This should probably always be either 1 or 2.

`PARAMETER [[rpi reference]] REAL 1.0e-8` — Zero-th order error coefficient C_0, used for convergence criterion if `sim_schemeOrder`= 0.

`PARAMETER [[rpi reference]] REAL 0.0001` — First order error coefficient C_1, used for convergence criterion if `sim_schemeOrder`= 1.

`PARAMETER [[rpi reference]] 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:

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

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 Δt 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 [[rpi reference]] by several orders of magnitude (over a range where it directly determines `maxActualDt`) for a given test in order to examine convergence behavior.

Part VI

Monitor Units

Chapter 18

Logfile Unit

Flash-X supplies the `Logfile` unit to manage an output log during a Flash-X simulation. The logfile contains various types of useful information, warnings, and error messages produced by a Flash-X 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 Flash-X2 and early releases of `Flash-X`, 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 causes 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-X simulations because it is required by the `Driver/DriverMain Config`. As with all the other units in Flash-X, 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 [[api reference]].

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 [[rpi reference]] in the `flash.par`.

```
# names of files
basenm   = "cellular_"
log_file = "cellular.log"
```

18.1 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-X 3.0.
 Build directory: /home/kantypas/Flash-X3/trunk/Sod
 Setup syntax:
 /home/kantypas/Flash-X3/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
==============================================================================
```

18.2 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 `Flash-X` logfile keeps track of whether a runtime parameter is a default value or whether its value has been redefined in the `flash.par`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-X Units used:
   Driver
   Driver/DriverMain
   Driver/DriverMain/TimeDep
   Grid
   Grid/GridMain
   Grid/GridMain/paramesh
   Grid/GridMain/paramesh/paramesh4
   ...
   Multispecies
   Particles
   PhysicalConstants
   PhysicalConstants/PhysicalConstantsMain
   RuntimeParameters
   RuntimeParameters/RuntimeParametersMain
   ...
   physics/utilities/solvers/LinearAlgebra
 ==============================================================================
 RuntimeParameters:

 ==============================================================================
algebra                     =          2 [CHANGED]
bndpriorityone              =          1
```

```
bndprioritythree            =          3
...
cfl                         =                0.800E+00
checkpointfileintervaltime  =                0.100E-08 [CHANGED]
cvisc                       =                0.100E+00
derefine_cutoff_1           =                0.200E+00
derefine_cutoff_2           =                0.200E+00
...
zmax                        =                0.128E+02 [CHANGED]
zmin                        =                0.000E+00
basenm                      = cellular_                      [CHANGED]
eosmode                     = dens_ie
eosmodeinit                 = dens_ie
geometry                    = cartesian
log_file                    = cellular.log                   [CHANGED]
output_directory            =
pc_unitsbase                = CGS
plot_grid_var_1             = none
plot_grid_var_10            = none
plot_grid_var_11            = none
plot_grid_var_12            = none
plot_grid_var_2             = none
...
yr_boundary_type            = periodic
zl_boundary_type            = periodic
zr_boundary_type            = periodic
bytepack                    =  F
chkguardcells               =  F
converttoconsvdformeshcalls =  F
converttoconsvdinmeshinterp =  F
...
useburn                     =  T [CHANGED]
useburntable                =  F

 ================================================================================

 Known units of measurement:

             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
 ================================================================================
```

18.3 Accessor Functions and Timestep Data

Other units within Flash-X may make calls to write information, or stamp, the logfile. For example, the `Driver` unit calls the API routine [[api reference]] after each timestep. The `Grid` unit calls [[api reference]] whenever refinement occurs in an adaptive grid simulation. If there is an error that is caught in the code the API routine [[api reference]] stamps the `logfile` before aborting the code. Any unit can stamp the logfile with one of two routines [[api reference]] which includes a data and time stamp along with a logfile message, or [[api reference]] which simply writes a string to the `logfile`.

The routine [[api reference]] is overloaded so the user must use the interface file `Logfile_interface.F90` in the calling routine. The next snippit shows logfile output during the evolution loop of a Flash-X 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
 ================================================================================
```

18.4 Performance Data

Finally, the `logfile` records performance data for the simulation. The `Timers` unit (see **??**) is responsible for storing, collecting and interpreting the performance data. The `Timers` unit calls the API routine [[api reference]] 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-X run complete.
```

18.5 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 19

Timer and Profiler Units

19.1 Timers

19.1.1 MPINative

Flash-X 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-X 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 [[api reference]] 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 [[api reference]] once at the beginning of a run, call [[api reference]] and [[api reference]] around sections of code, and call [[api reference]] at the end of the run to report the timing summary at the end of the logfile. However, it is possible to call [[api reference]] 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 [[api reference]] does not reset the integer key mappings, it is safe to obtain a key through [[api reference]] once in a saved variable, and continue to use it after calling [[api reference]].

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

Table 19.1: Timer Unit runtime parameters.

Parameter	Type	Default value	Description
`eachProcWritesSummary`	`LOGICAL`	`TRUE`	Should each process write its summary to its own file? If true, each process will write its summary to a file named timer_summary_<process id>

Table 19.1: Timers parameters (continued).

Parameter	Type	Default value	Description
writeStatSummary	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 ?? for an example of this type of output. The max, min, and avg summary can be disabled by setting the runtime parameter [[rpi reference]] 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 [[rpi reference]] to false.

19.1.2 Tau

In Flash-X3.1 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-X labeled code sections (marked by Timers_start and Timers_stop). After running the simulation, the Tau profile contains timing information for both Flash-X 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-X. 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-X labeled code section. Another reason to use the Tau version is that the MPINative version (See ??) is implemented using recursion, and so incurs significant overhead for fine grain measurements.

To use this implementation we must compile the Flash-X source code with the Tau compiler wrapper scripts. These are set as the default compilers automatically whenever we specify the -tau option (see ??) 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.

19.2 Profiler

In addition to an interface for simple timers, Flash-X includes a generic interface for third-party profiling or tracing libraries. This interface is defined in the monitors/Profiler unit.

In Flash-X we created an interface to the IBM profiling libraries libmpihpm.a and libmpihpm_smp.a and also to HPCToolkit http://hpctoolkit.org/ (Rice University). We make use of this interface to profile Flash-X evolution only, i.e. not initialization. To use this style of profiling add -unit=monitors/Profiler/ProfilerMain/mpihpm or -unit=monitors/Profiler/ProfilerMain/hpctoolkit to your setup line and also set the Flash-X runtime parameter profileEvolutionOnly = .true.

For the IBM profiling library (mpihpm) you need to add LIB_MPIHPM and LIB_MPIHPM_SMP macros to your Makefile.h to link Flash-X to the profiling libraries. The actual macro used in the link line depends on whether you setup Flash-X with multithreading support (LIB_MPIHPM for MPI-only Flash-X and LIB_MPIHPM_SMP for multithreaded Flash-X). 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 -lmpihpm $(HPM_COUNTERS) $(LIB_MPI)
LIB_MPIHPM_SMP = -L/soft/perftools/hpctw -lmpihpm_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

Tools

Chapter 20

VisIt

The developers of `Flash-X` 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 `Flash-X2.5` and `Flash-X`. Version 1.10 and higher natively support `Flash-X`. For VisIt versions 1.8 or less, `Flash-X` support can be obtained by installing a tarball patch available at http://flash.uchicago.edu/site/flashcode/user_support/visit/. Full instructions are also available at that site.

Chapter 21

Serial Flash-X 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-X version 2.0 as a replacement for focu.

Sfocu is a serial utility which examines two Flash-X checkpoint files and decides whether or not they are "equal" to ensure that any changes made to Flash-X 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-X 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.

21.1 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.

21.2 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        |
mfrc | 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        |
mfrc | 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 22

Drift

22.1 Introduction

Drift is a debugging tool added to Flash-X 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-X, 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
```

22.2 Enabling drift

In Flash-X, 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-X 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
`drift_trunc_mantissa`	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.
`drift_verbose_inst`	0	The instance index at which drift should start logging checksums per call to `Grid_releaseBlkPtr`. Before this instance is hit, only user calls to `Driver_driftUnk` will generate log data. A value of zero means never log checksums per block. Instance counting is described below.
`drift_tuples`	.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 `driftDee` script found in the `tools/` directory. Generally the "non-tuples" format is a better choice to use with tools such as `diff`.

22.3 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 (`dirft_verbse_inst`=0). The main `Driver_evolveFlash` implementations have calls to `Driver_driftUnk` between all calls to Flash-X 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.

22.4 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

`#include "Flash.h"`
`subroutine Flash_subroutine()`
`  use Grid_interface, only: Grid_releaseBlkPtr`
`  implicit none`
`  [...]`
`  call Grid_releaseBlkPtr(...)`
`  [...]`
`end subroutine Flash_subroutine`
	`subroutine Flash_subroutine()`
	`  use Grid_interface, only: Grid_releaseBlkPtr`
	`  implicit none`
	`#include "Flash.h"`
	`  [...]`
	`  call Grid_releaseBlkPtr(...)`
	`  [...]`
	`end subroutine Flash_subroutine`

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 forceably 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`.